A Modular Flow for Rapid FPGA Design Implementation

Andrew R. Love

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

Mark T. Jones

Leyla Nazhandali

February 11, 2015

Blacksburg, Virginia

Keywords: FPGA Productivity, Modular Design,

Instant Gratification, Attention Span, Design Assembly Flow

Copyright 2015, Andrew R. Love

A Modular Flow for Rapid FPGA Design Implementation

Andrew R. Love

(ABSTRACT)

This dissertation proposes an alternative FPGA design compilation flow to reduce the back-end time required to implement an FPGA design to below the level at which the user's attention is lost. To do so, this flow focuses on enforcing modular design for both productivity and code reuse, while minimizing reliance on standard tools. This can be achieved by using a library of precompiled modules and associated meta-data to enable bitstream-level assembly of desired designs. In so doing, assembly would occur in a fraction of the time of traditional back-end tools. Modules could be bound, placed, and routed using custom bitstream assembly with the primary objective of rapid compilation while preserving performance. This turbo flow *(TFlow) aims to enable software-like turn-around time for faster prototyping by leveraging precompiled components. As a result, large device compilations would be assembled in seconds, within the deadline imposed by the human attention span.*

Contents

1 Introduction **1**

1.1 Machine Tools and Modular Design 3

1.2 Software Design and Reuse 4

1.3 Software Development Cycle 6

1.4 Productivity and Attention Span 8

1.5 FPGA Design and Reuse 10

1.6 FPGA Design Approach 13

1.7 Contributions 16

1.7.1 Purpose 19

1.8 Dissertation Organization 20

2 Prior Work **22**

2.1 Chapter Organization 24

2.2 Analogous Use Case 24

2.3 Placement 26

2.3.1 Constraint Generation 29

2.4 Modular Design 29

2.5 Flow Comparison 35

2.5.1 Flow Capabilities 37

2.6 Meta-data Description 39

2.7 Bitstream Relocation 40

2.8 Summary 42

3 Implementation **44**

3.1 Flow Motivation 46

3.2 Flow Design 46

3.3 Flow Model 51

3.3.1 Model Application 53

3.3.2 Model Preprocessing Trade-Offs 57

3.3.3 Module Reuse Model 59

3.3.4 Module Parallelism Model 60

3.3.5 Strategy to Meet a Hard Placement Deadline 60

3.3.6 Strategy to Meet a Hard Routing Deadline 62

3.4 TORC 62

3.5 Module Relocation 63

3.6 TFlow Phases 65

3.7 Module Creation Phase 65

3.7.1 Module Creation 66

3.7.2 Module Shaping 67

3.7.3 Module Compilation 68

3.7.4 XML Meta-data 72

3.8 Static Creation Phase 74

3.9 Design Assembly Phase 76

3.9.1 Design Entry 77

3.9.2 Module Placement 78

3.9.3 Inter-module Routing 83

3.9.4 Clock Routing 84

3.9.5 Bitstream Stitching 84

3.10 Debugging 85

3.11 Summary 86

4 Results **88**

4.1 Flow Optimization 88

4.1.1 Router Optimization 89

4.1.2 I/O Optimization 89

4.1.3 Placer Optimizations 91

4.2 TFlow on Virtex 5 96

4.2.1 Design Assembly on the Virtex 5 Architecture 99

4.2.2 Further Virtex 5 Design Assembly Exploration 101

4.2.3 Attention Span Comparison 106

4.3 TFlow on the Xilinx 7 Series 107

4.3.1 Placement Overhead 110

4.3.2 Area Overhead 112

4.3.3 Area Penalty Amelioration 113

4.3.4 Area Overhead Versus Placement Time 114

4.3.5 Zynq 7 Static Design 115

4.3.6 Design Assembly on the Zynq 7 Architecture 117

4.4 Summary 124

5 Conclusion **126**

5.1 Contributions 126

5.2 Ongoing and Future Work 131

Bibliography **135**

List of Figures

1.1 Evolutionary Prototyping Software Development Cycle [1] 7

1.2 TFlow's Gajski-Kuhn Y-chart 16

1.3 Human Attention Span Time Constraints 17

2.1 Xilinx XC5VLX20T Frame Layout 41

3.1 TFlow Use Model 47

3.2 Meta-data Annotation Process 69

3.3 XML Port Information 73

3.4 XML Net Information 74

3.5 Design Connectivity Example 78

4.1 Sequential Placement (In Windows Adobe Reader, click for video) 95

4.2 Flow Assembly Time Relative to Attention Span 107

List of Tables

2.1 Flow Goals Comparison 37

2.2 Flow Capabilities 38

3.1 Model Terms and Impact 58

3.2 Vertical expansion of clock region size for modern Xilinx FPGA devices. . . . 64

4.1 Router Run-time Improvements 89

4.2 Placement Results, 15 Streaming Blocks 92

4.3 Placement Results, 10x10 Grid 93

4.4 Placement Results, 7x7x3 3D Grid 93

4.5 Placement Results, ZigBee Radio Design 94

4.6 Virtex 5 Module Run-time and Timing Analysis 97

4.7 Virtex 5 Module Resource and Placement Analysis 98

4.8 Virtex 5 Static Timing Analysis 99

4.9 Assembly Time for BPSK Radio (s) 101

4.10 Virtex 5 Design Assembly Comparison 103

4.11 Virtex 5 TFlow I/O Comparison 104

4.12 Virtex 5 HMFlow Overhead (seconds) 105

4.13 Zynq Module Run-time and Timing Analysis 109

4.14 Zynq Module Resource and Placement Analysis 110

4.15 Zynq Large Module vs Small Modules 111

4.16 Zynq BPSK Design Area Penalty 112

4.17 Zynq BPSK Design Area Penalty With Sub-frame Modules 114

4.18 Minimum Granularity Vs Placement Time 115

4.19 Zynq-7000 Static Timing Analysis 117

4.20 Zynq Design Build Time 119

4.21 Total Design Time including Precompilation (s) 120

4.22 Zed Design Build Time 122

4.23 GReasy Zynq Overhead 125

Chapter 1

Introduction

The path from idea to implementation can be a long and arduous one. This path can go by many routes. Choosing the best path to follow depends on the user's requirements. For example, what drives the user; what priorities must be met? Is it cost, efficiency, time, or some other factor? If cost is the driver, then minimizing the most costly portions is appropriate. In industrial production, skilled labor can be a driving cost; thus, the skilled can make tools that the unskilled can supervise [2]. This reduces the number of necessary skilled laborers, cutting costs. If design efficiency is most important, then each component should be created with exacting standards and optimization. This leads to time-consuming handcrafted results. Lastly, if time is the driver, then creating a working solution as quickly as possible is best. Wartime projects are normally done under this constraint. The Manhattan Project, the SAGE air defense system, and the V-2 Rocket all qualify [3]. In practice, each of these components and many others are weighed against one another to determine the best

approach. Very rarely are any of them disregarded.

Another consideration is to determine how many times the design or its components will be used. If the design is a one-off where it will be tested and then either used as-is or improved, then performance requirements can be less stringent. There is a trade-off between the number of uses and production time. Many cars can be built from the same design; thus, small improvements to the design will be multiplied considerably. Prototype vehicles, on the other hand, are limited in number. Time spent on small improvements may be better spent on analyzing the prototype and generating a final design. Small tweaks may be inapplicable to the final design and thus a waste of resources at this juncture.

Even with this trade-off, building multiple smaller modules that can connect to one another has its advantages. The International Space Station (ISS) was built using multiple smaller modules and then assembled in orbit. Additional modules were added on over time, expanding the station's capabilities. There are many advantages that accrued due to this assembly process. For one, since the modules can be independent, failure of one module does not cause failure of the station. More capabilities could be added over time, as additional modules were put into orbit, without changing the existing station. The modules could be built separately, by teams all over the world, and then combined using standard connections. This distribution of labor improves productivity when divided into appropriately sized modules. Again, the balance of priorities had its influence on the design, as redundancy became an additional requirement.

This work focuses on balancing these requirements, with the focus on maintaining the users

focus, or 'flow' [4]. The goal is to maximize 'flow' and minimize wait time. This will reduce wasted time, increase designer efficiency, and improve productivity. To do so, this work plans to leverage the benefits of modular design.

1.1 Machine Tools and Modular Design

One field that has embraced modular design is that of machine tools [5]. These are the tools that are used in production environments. For example, car factories use automated machine tools to create vehicles. Car designs are modified often to refresh and update the models. If new tools were needed each time the design changed, it would be expensive and inefficient. Instead, the tools are built using modular components, so the factory only needs to swap out or create a few new components to get the desired functionality. One machine tool center at Opel has an eighty percent reuse rate when updating products because of its modularity [5]. To ensure that these modules are interchangeable, the International Organization for Standardization (ISO) has standards governing these tools [6]. Modular design thus combines standardization with flexible configuration to create something better then both.

Modular design has four main principles, separability, standardization, connectivity, and adaptability [5] [7] . Separability describes the determination of the size of a module. A large and complex module can be split up into multiple smaller modules. This yields increased design flexibility; however, the additional principles will create a trade-off where smaller

modules are not always better. Standardization deals with standard sizes and shapes of modules, such that interchangeability is enhanced. With these standard sizes, splitting up modules may create overhead in the form of wasted space and resources. Connectivity deals with the interfaces between modules. A standard connection method may again create overhead, were the smallest modules used. Additionally, standard interfaces allow modules to interface with one another to create arbitrary designs. This leads to adaptable modules, that can be reused in new and interesting ways. When dealing with machine tools, these principles were first presented in the 1960s, and have maintained their importance as they evolved into the current modular design flows.

Additional modular design techniques have been put forward to better improve the design process, including improvements of the graphical interface and modular structure [8]. Defining modules properly lets designers create the desired modules, resulting in interchangeability and a paper trail of documentation. One such language is the Unified Modeling Language (UML) [9]. This allows for the module connectivity and relationships to be well-defined.

1.2 Software Design and Reuse

Hardware and software labor productivity have taken different paths as technology improved. In the electronics industry, hardware labor productivity has improved markedly, while software labor costs has grown to more than 90% of the total system installation price [10]. This is partially due to the high amounts of reuse of hardware.

To generate solutions quickly and effectively, one methodology is to use preexisting components as much as possible. Code reuse has a long history, and the idea can be traced back to the start of software design [11]. In software design, many benefits accrue from reuse: quality improvements, productivity, and cost reduction [12]. Depending on the application, this can be referred to as COTS (Commercial Off the Shelf) or as a component library.

The benefit of using existing sub-solutions is that their functionality has already been tested and optimized. As a software example, sorting can be easily implemented by a designer. However, the standard software sorting algorithm has not only been tested for functional correctness, but guarantees minimum performance. The creation and analysis of sorting algorithms is a complex field, and should more stringent constraints be required for a specific design, different algorithms could be explored. For a quick and effective solution, the standard methodology is sufficient.

A software library example is for Digital Signal Processing (DSP), with the Liquid DSP library [13]. This lightweight C library contains a full set of DSP functions that can be inserted as necessary into a design. A DSP application can be rapidly built using these abstract functional blocks. Implementation and testing of these blocks has already been completed by the library designer.

Another example are the C++ boost and standard libraries. Software programmers do not need to handcraft the lowest level of their functions and can instead build upon prior work. One reason these libraries can have widespread acceptance is that they have useful functionality and standard interfaces.

For best design principles for a digital tool, Bürdek states ”Deep complexity requires surface simplicity” [10]. The front end of a tool should be simple in nature, but the implementation can be highly complex. TFlow, a modular assembly tool presented in this dissertation, has this property. The front end is a standardized design description, while implementation involves complex low-level manipulations. Additionally, by enforcing modular design, block reuse becomes an integral part of TFlow.

1.3 Software Development Cycle

The software development cycle is iterative in nature and modular if possible [14]. There are a wide variety of techniques, but each of them require a feedback loop linking design, implementation, and testing. Figure 1.1 shows an example of the process. Initially, the requirements are given to the programmer. These requirements are used to build a design. This design must be implemented and tested. Once this is complete, feedback is necessary to verify that the design properly implements the requirements and that these requirements have not changed. This process continues until the design is acceptable. One situation where this loop is most evident is when dealing with prototyping.

For rapid prototyping, quick feedback is essential. This is especially true when dealing with throwaway prototyping [15], where quick implementation and modification are prioritized. The quicker a throwaway prototype can be analyzed, the quicker an evolutionary prototype or full design can begin production. With a quicker prototyping stage, the requirements can

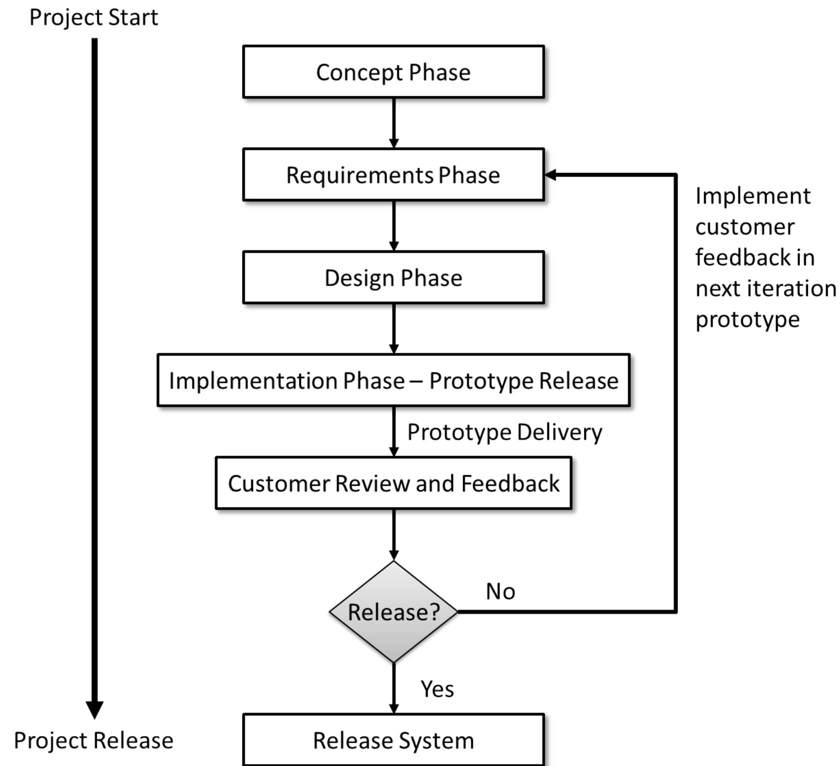

Figure 1.1: Evolutionary Prototyping Software Development Cycle [1]

be corrected and the design documents updated faster. Dead-end design space exploration can be determined quicker, yielding a much more efficient design process.

When performing evolutionary prototyping, a simple version of the design is programmed and tested [1]. If issues are found, the programmer will fix these issues. Should the prototype meet specifications, the next iteration will evolve further capabilities, until the full requirements are met and a final design created.

These techniques can be applied to Field Programmable Gate Arrays (FPGAs), but one drawback is that the time necessary to go from design to test can be very long. This would increase the size of the feedback loop and reduce productivity. Shortening this loop would

allow for more design space exploration and/or additional testing and verification time.

1.4 Productivity and Attention Span

Improving productivity is an important issue facing designers in a wide range of applications. With increased productivity, designers are able to produce more and better results within the same amount of time. The design flow methodology is an important factor influencing productivity. Although there are many variants, design flows are normally iterative in nature.

An iterative design process loops through a number of phases before a final result is generated. A high-level view of this process can be seen in Figure 1.1. Before an iterative design loop, a design concept is generated. Next, the design requirements are specified. After this, a design is built to meet these requirements. The design is then implemented, and the result is evaluated against the requirements. The result of this evaluation will be used as feedback for the next iteration. Each of the phases has significant user interaction except for the automated implementation phase.

The user's response to an automated computation phase follows one of two possible paths. Either the process finishes quickly, and the user continues on with their task, or there is a long delay and the user switches to another task. These mental context switches have a cost in time and cognitive load, and thus are undesirable [16] [17]. A design process that can remove the need for context switches will be better streamlined and have increased productivity.

In order to remove the need for context switches, the process must complete before losing the attention of the user. This attention span is a finite amount of time after which there is a mental discontinuity as attention is lost [18] [19] [20]. Should the user sit and wait without performing a context switch, the ordeal reduces both user motivation and productivity [18]. This effect can be ameliorated by adding a percent complete indicator [21]. This indicator enables a user to determine whether they can safely switch to secondary activities.

Analysis of attention span has shown that people are willing to wait for a response for about 15 seconds [18]. After this wait time, the user will begin filling the wait time with secondary activities and will switch metal contexts [18]. Card [19] expands this wait time to around 5 to 30 seconds for task completion, while Nielsen [20] places the limits of user's attention at 10 seconds. Giving a response within this window removes the need to switch to other tasks.

The effect of switching to other activities has been studied by O'Conaill [22]. In this study, interruptions sometimes caused the task to be discontinued entirely. After an interruption, approximately 40% of the time the waiting person will go on to other work instead of returning to their original task. Should a task take some time to complete, the user will most likely have switched to another task. There is only a 60% chance that the user will return to the first task once it completes [22]. At this point the user has moved on, and the flow will need to wait indefinitely for user input. Removing these interruptions by having tasks complete before losing a user's attention therefore has significant advantages in productivity and mental focus, and losing the user's attention has a productivity penalty.

1.5 FPGA Design and Reuse

While there are a number of different tools used to generate designs for FPGAs, they all have a bottleneck when creating the physical design. Partially, this is due to the highly device-specific nature of physical back-end implementation. Back-end implementation for FPGAs consists of the post-synthesis design phases, including device mapping, routing, and bitstream generation. With the advent of frameworks like the Tools for Open Reconfigurable Computing (TORC) [23], physical device information is exposed. Unlike generic tools such as Versatile Place and Route (VPR) [24], real-world designs can be built and implemented. This gives the capability to modify the back-end. Modifications must be chosen that yield the desired improvements.

The existing tools focus on a slew of competing requirements when generating a final design. These factors include resource utilization, packing, routing, and timing closure. The total run-time to optimize these requirements is less important than the quality of the result. For the current target audience, maximum performance and resource packing are the focus. Larger designs can fit into the same physical device with global optimization and better resource utilization. Timing closure is necessary to wring out the fastest designs. While the traditional tools aim to minimize the time necessary to run these optimizations, this is secondary to improving the results.

There are other use cases for FPGAs that do not require optimal results. Two of these use cases are productivity and rapid prototyping. Focusing on mature software design ap-

proaches can maximize productivity. These are normally predicated on quick turn-around times for generating a design so that it can be promptly evaluated. Another use case is rapid prototyping. Prototypes can be used for tests on real hardware. Currently, simulations are a good way to test designs without needing to run through the time-consuming tool flow, which may take hours to finish. Rapid prototyping can replace some of these simulations.

An untapped audience for FPGAs is software designers. To enable the use of FPGAs, one method is to abstract away the physical details and use a higher level approach. Graphical tools such as LabVIEW can perform this sort of FPGA design. Unfortunately, these tools do not ameliorate the long back-end compilation time required. This time is on the order of minutes or hours. Software designers accustomed to compilation taking seconds may not be willing to make this sacrifice. However, if a faster way to generate FPGA designs from high-level representations are available, software designers could gain the advantages of FPGAs without some of the shortcomings. Tight integration has been used successfully for hardware/software codesign [25]. With recent advances, acceleration using Graphical Processing Units (GPUs) can be seamlessly integrated into software designs as well [26]. FPGA usage should follow suit.

In the FPGA design compilation process, 90% of the compile time is spent doing FPGA place and route [27]. Amdahl's Law thus guides any effort to reduce compile time to first focusing on reducing the compile time for these steps.

Modular design is a principle where a design with performance and functionality specifications can be designed and built by picking and combining the necessary modules from a

preexisting library [5]. With well-defined interfaces, these modules can be worked on independently. Changes to one module do not necessitate changes to the rest of the design. This approach fits in well with speeding up compilation time. For software designers, the modules abstract away the hardware details and can be treated as building blocks. These building blocks can be pre-built, reducing the time required to create the design. Since the modules are independent, rapid prototyping of a new component will not require changing the rest of the design.

While prior attempts have been made at implementing modular design for FPGAs, they are not without their drawbacks. They rely on licensed tools to generate the final designs. This limits the environments that these solutions can be deployed and restricts the modifications that can be made. A few examples of prior modular design flows are HMFlow [28], QFlow [29], and Xilinx Partitions [30]. While design compilation time is reduced - in some cases significantly - the results are still in the realm of minutes to generate a design. Software compile time normally completes in seconds, so ideally FPGA compilation would be the same. These example flows rely on some stages of the standard pipeline to generate the final results. While this does generate more optimized FPGA designs, the goal of this work is to meet the speed requirements. For example, QFlow and Partitions require the Xilinx router [29] [30], HMFlow requires the Xilinx Design Language (XDL) conversion utility [28], and all three rely on the Xilinx bitstream generation tool. Custom tools can be designed with a focus on speed, so reducing the reliance on vendor tools will help achieve this goal.

1.6 FPGA Design Approach

In the design process from Figure 1.1, the implementation phase is automated. Both the design and evaluation phase require user feedback. To keep the user on task, the implementation phase needs to complete before losing the user's attention. This places a hard time limit on completing the implementation phase if the resulting productivity gains are to be achieved. Current FPGA design flows do not finish within this narrow window, as implementation takes minutes or hours to complete.

The goal of this FPGA design flow is thus to implement a design within the narrow window of human attention span. To meet this goal of second long compilation times, the conventional algorithms need to be reformulated. Modeling the flow has shown that an effective approach is to perform as much work as possible during precompilation. For FPGAs, compilation ends with a bitstream, and so the closer a design is to this final bitstream, the faster the flow. If map is performed ahead of time, it is no longer necessary for it to run at design time. The same holds true for placement and routing. This leads to the logical extension that if modules can be precompiled into their end state - bitstreams - maximal performance can be wrung out of the assembly flow.

To enable this capability, a full design flow needs to be built. The module bitstreams must be created, and a method of storing information about these designs is necessary. The module bitstreams need to be added to a library for later assembly. A static design must be created and compiled into the bitstream library as well. Routing for each of these library blocks

needs to be constrained to remain inside each block. A method to fetch these library blocks at assembly time needed to be created. These blocks need to be placed and routed at the bitstream level, and so a module placer and a inter-module router are necessary. To put it all together into a usable bitstream, these components must all be stitched together. This whole process is predicated on running the modules through the entire standard flow, so that assembly no longer needs to run any of the standard flow tools. As such, assembly can be done as a custom, standalone process. By performing as much work as possible during library creation, design assembly can occur rapidly. Requiring only seconds for design assembly greatly increases the number of designs that can be built and can keep the user's attention from wandering. This gives more time for the remainder of the design process - design, testing, and verification.

This work presents a turbo flow, *TFlow*, that implements this rapid design implementation flow. This flow is a novel method whereby modules are run through the entire Xilinx flow and are stored as bitstreams for assembly at design time. Other techniques do not fully compile the modules and require additional processing before a design bitstream could be used. Nor do other techniques meet the strict time requirements necessary.

Module creation relies on the standard tools, where optimizations such as timing closure and resource allocation can be done on a per-module basis. With the appropriate safeguards in place to prevent module-module collisions or timing issues, these modules can thus be used in combination to create a full design.

This technique is not without its drawbacks. Additional information describing the fine-

grained module structure is necessary. Additional micro-bitstreams are needed to perform inter-module connectivity. The proposed methodology will generate these modules into a library and later - independent of the vendor tools - assemble them into a full design in seconds.

TFlow consists of multiple stages, each of which combine to produce a significant contribution to the field. TFlow is predicated on stitching module bitstreams together to create a full design. The data inside the bitstreams is unknown, so meta-data describing the module is necessary. Currently, no one source contains the full picture of the module. This includes physical information, consisting of port locations and routing, the logical port structure, and additional shaping and placement information. This big picture module meta-data is just one contribution of this flow.

For module packing to occur optimally, the modules should be created in such a way that no excess resources are included within their boundaries. Module bitstreams cannot overlap. To enable packing, the module requirements are analyzed and a minimum sized block is reserved on the device. Module relocation is taken into account, so that these blocks can be rearranged appropriately. Module shaping reduces the area overhead of each module.

From a user perspective, TFlow abstracts away the implementation details. The Gajski-Kuhn Y-chart [31], seen in Figure 1.2, deals with the abstraction levels of design. In the structural domain, TFlow's modules are seen as subsystems. Behaviorally, the user sees TFlow's modules as algorithms to be used, while internally the register-transfer information is needed. With respect to the physical domain, TFlow needs to consider the floor plan of

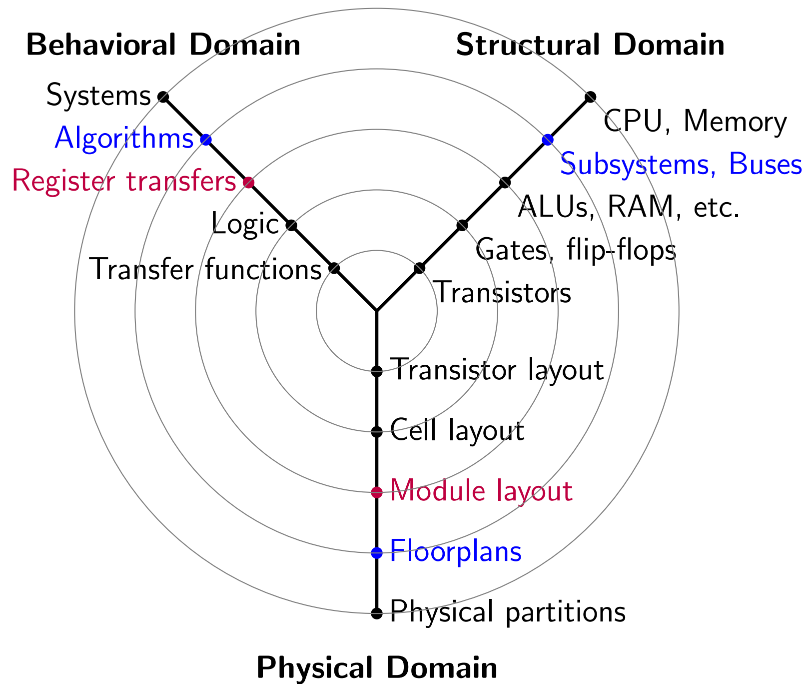

Figure 1.2: TFlow's Gajski-Kuhn Y-chart

the device during assembly, and the module layout during module creation. This puts TFlow at a mid to high level of abstraction with respect to the device.

1.7 Contributions

Improving productivity remains an import goal for designers. Productivity can be enhanced by improving design experience. One key factor in the design experience is producing immediate feedback [4], within the limitations of the human attention span [18]. This work improves the FPGA design experience through the following contributions.

1. The primary contribution of this work is proof that FPGA design implementation can

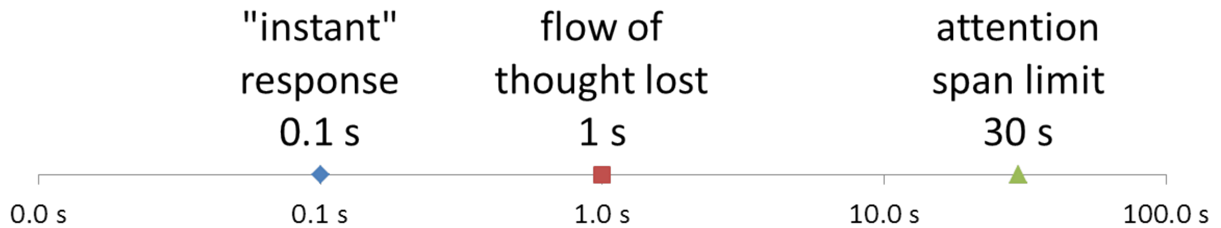

Figure 1.3: Human Attention Span Time Constraints

finish within the deadline imposed by the human attention span. This proof takes the form of the presented modular design flow, TFlow. Meeting this deadline allows for increased design productivity and enables new approaches to FPGA design.

Figure 1.3 shows the various time limits to the human attention span [18] [19] [20]. Other applications might focus on the response times necessary for instant response or maintaining flow of though, but the goal of this work is to complete implementation before reaching the attention span limit. As soon as this limit is reached the user's attention will be lost and productivity will drop [18].

TFlow provides proof that FPGA design assembly can comfortably fit within the window of the human attention span. This is a significant improvement on the state of the art, as all other techniques remain outside this narrow window. With this advance, user focus can be maintained while traversing the design-implement-test process flow.

All other contributions are in service to meeting the hard deadline imposed by the limitations of the user's attention span. Implementing and automating a flow that can meet this deadline

required the creation of a number of capabilities, many of which are contributions in their own right.

2. To maintain performance while speeding up design implementation, the conventional design process needs to be reformulated. The presented approach uses modular design techniques to split the conventional algorithms into compile time and design time components. Compile time computation involves building the component library ahead of time. This moves a significant amount of computational effort out of the critical path. The design time complexity is thus reduced significantly, enabling the creation of a flow that can complete within the allotted time.

    (a) TFlow includes a fast and efficient module placer designed to meet the time deadline imposed by the overall flow. Placement completes within two seconds with the most complicated test design running on a 2.83 GHz Intel Core 2 Quad with 3 GB of DRAM.

    (b) TFlow includes a quick router that is designed to meet the time budget allocated for routing. Routing using this inter-module router produces results 7.8 times faster than the standard ISE tools on an Intel i7-2600 with 8 GB of DRAM.

    (c) TFlow assembly occurs at the lowest possible level to reduce any temporal overhead. For FPGAs, this lowest level is the bitstream. Bitstream level manipulation tools were created to perform bitstream relocation and bitstream routing.

    (d) A metadata description methodology was created which which contains all of the

necessary information that TFlow will require for design assembly. The module blocks could not be used for this purpose because they are bitstreams, which are opaque to the design tools.

3. Existing tools are built for a different problem space than TFlow. Forcing TFlow to complete within the given timespan has required a simplified custom assembly process. These simplifications have given TFlow the added capability to run on embedded devices where it would otherwise be impossible to create a design. This embedded design assembly toolflow is a secondary contribution of this work.

1.7.1 Purpose

TFlow is intended as a way to build designs such that the user transitions from design to test without their attention wandering during implementation. This forces a hard time constraint on the implementation flow that can be best met through the use of modular design and precompilation. This seamless transition enables design space exploration, as well as rapid prototyping.

There are always trade-offs, however, and so TFlow is not a design panacea. Because of its focus on deadline completion, TFlow trades away the ability to achieve the best design density and the fastest circuit. It cannot be used for implementation space exploration, where different implementations are analyzed to determine optimality. Lastly, TFlow is not a partial reconfiguration flow - the modules are not intended for on-line replacement.

In addition, there are costs to performing precomputation. The total amount of computation is not reduced overall, but may in fact increase. These increases can be due to the loss of global optimization as well as the overhead incurred by building the modules independently. Since computation is shifted, at design time most computations are already complete. As such, some design flexibility is lost - the precomputed modules are now static constraints.

The concept also has limitations. When dealing with modular design, determining the appropriate way to split up functionality is difficult. Choosing too large of a module loses flexibility, while the overhead incurred from small modules in both space and time can cause the flow to no longer complete within the allotted time. In addition, determining the capabilities of the non-modular area - the static region - will impact what sort of designs can be built. Changing this static region uses the slow, standard method of design implementation, with all its drawbacks.

TFlow shows that it is possible to implement designs within the time constraints for user 'flow' [4], enabling a paradigm shift in FPGA design.

1.8 Dissertation Organization

The rest of the dissertation is organized in the following manner. Chapter 2 discusses prior work in FPGA modular design and some background on the many facets of TFlow. Chapter 3 discusses TFlow's model and implementation. It covers how TFlow works and its driving motivation. Chapter 4 shows the results from implementing TFlow on real FPGAs and

how it compares to other compilation flows. Chapter 5 discusses the impact of TFlow, its contributions, and avenues for future work.

Chapter 2

Prior Work

As the size of FPGAs increase, Parkinson's law of design complexity implies that the size and complexity of designs will increase to fill the space [32]. The time required to build these larger and more complex designs also grows unless FPGA productivity keeps pace. Code reuse and high-level design can help to improve productivity without a significant impact on the end result. Current tools have the capability to utilize these techniques, but do not enforce their use. As with modular design [5], there is some additional work that must be done to have reusable code. Interfaces need to be standardized and documentation of the block must be maintained. Without enforcement, many designers will skimp on these stages to get a viable one-off product out the door. While this does work for the first design, the lack of reusable blocks means that design needs to start from the beginning each time. Thus, despite the set-up overhead, well documented and structured code can give significant productivity gains. General-purpose programming libraries, such as the C++

boost library, or the Xilinx IP CORE library [33] are powerful productivity tools. However, application-specific libraries can yield these same improvements, targeted at the required design. This type of library can be built by the programmer, if they followed stringent code reuse guidelines.

One such guideline for FPGAs is modular design. Modular design is a principle where a design with performance and functionality specifications can be designed and built by picking and combining the necessary modules from a preexisting library [5]. These modules have standard interfaces to ensure connectivity. By building a design as a series of modules, mixing and matching these components can allow for the creation of different end results. In addition, the blocks can be combined and used without needing to know how they are implemented.

As FPGAs continue to grow, the amount of time necessary to implement a synthesized design remains considerable. A significant factor is the complexity of the desired designs. If the design goal is to meet the attention span deadline, then trade-offs must be made. One way to obtain the desired time reduction is to leverage the benefits of modular design. Modular design has been used to good effect in software [11], machine tools [5], and construction [34], among others. Modules can be precompiled to be as complex as necessary. Assembling these modules into a final design is a simpler problem then building the final design all at once. This work will demonstrate the significant time reduction possible when assembling modular designs. With this modular design flow, the attention span deadline can be met.

2.1 Chapter Organization

This chapter covers prior work that has relevance to the creation of a sub-attention span toolflow. Section 2.2 will cover a prior case where there were significant wait times and long feedback loops when creating a design, and how this problem was solved. Section 2.3 discusses the placement problem and how it applies to TFlow's placement strategy. Section 2.4 takes a look back at how modular design has been built and used in the past to improve the performance of FPGA design, as well these methods advantages and disadvantages. The next section, Section 2.5 compares and contrasts different FPGA design flows to emphasize the benefits of TFlow. Section 2.6 discusses how previous work represented the necessary information to create a cohesive flow, while Section 2.7 covers the history of bitstream relocation and how it works, which will be applied to TFlow. Lastly, Section 2.8 summarizes the lessons learned from this prior work and discusses how this information will inform the implementation design of TFlow.

2.2 Analogous Use Case

Prior to the advent of personal computers, shared mainframes were used for compiling and running software programs. Programming and data entry on these shared mainframes were done with the use of punch cards. Punch cards were physical cards that had holes which could be punched out to indicate the data stored on them; the standard IBM card could store as many as 80 characters per card [35].

Building a design using punch cards took multiple phases, each one of which could have day-long turn around times. Initially, the program would be written and the appropriate punch cards created. This process required users to submit their code for entry into a keypunch machine which would create the cards.

This set of punch cards would then need to be compiled to create the program, which was also stored on punch cards. Compilation jobs were submitted to a mainframe. This process was manual, and a user would have to wait until it was their turn for compilation. Compilation errors or any other problems would require changing the code and then resubmitting these bug fixes to the keypunch machine. The turn around time on these jobs was such that only two or three compilation jobs could be run in a day [36]. With a dedicated keypunch machine, the number of turns per day could double. Once a program was compiled, it would need to be resubmitted to the mainframe in order to run. The results would be returned to the programmer, who would begin the process all over again. Each step in this process had a long wait time for completion, reducing the number of times a program could be compiled and run.

The next bottleneck occurred once programs could be stored and run on the mainframe. Since submitting jobs no longer required an additional person in the loop, more jobs could be done in a day. Sharing the mainframe limited the number of people that could compile and run their program at once. As the number of people and projects increased, the load on these systems would slow down the number of turns-per-day. This problem was solved by adding multiple smaller machines to the computer pool. The process of splitting work up and

running it in parallel is analogous to the module and static creation processes performed by TFlow. This yields a similar result, with the speed and number of turns-per-day improving considerably.

2.3 Placement

In the CAD placement field, placement algorithms can be categorized into two types, (a) constructive placement and (b) iterative placement improvement [37]. Additionally, the metrics to evaluate a placement depend on the desired goal. For example, timing, packing, and wire-length are all valid metrics to determine the efficacy of a placement strategy. These placement techniques each have advantages and disadvantages, with research ongoing to improve them.

Constructive placement is a rule-based method used to generate a constructed placement [37]. This method, in turn, has two main approaches, partitioning-based placement and analytic placement. Partitioning-based placement involves splitting up the design such that the number of nets crossing partition boundaries is minimized. This leaves areas with dense networks in the same partition. Once the partitions are small enough, they are placed on the FPGA. This is a speedy method of placement, but it revolves around minimizing the number of cut nets without regard to wire length. This can yield some un-optimized solutions [38].

Analytic placement strategies treat placement as a top-down problem. Design connectiv-

ity is used for deciding placement optimization. Solving the generated system of linear equations is best done when this connectivity is represented by a continuously differentiable value. One such placer, StarPlace [39], uses linear wire-length as its objective function. This generates a system of non-linear equations that must be minimized to obtain an optimal placement. However, this placement is unlikely to be valid. Another pass is necessary to remove collisions, and the placements need to be adjusted to integer locations that represent the physical structure of the FPGA. StarPlace's final placement is thus no longer guaranteed to have minimal linear wire-length.

Constraint-based placement is a method of analytic placement where the problem space is drastically reduced. This significantly reduced problem space allows for faster run-times, at the expense of a much lower placement granularity [40] [41].

Iterative placement improvement repeatedly performs operations such as placement swaps and moves to obtain a better result. This requires a seed placement as a starting point for these optimizations. This seed placement is normally generated by a constructive placement technique. One iterative placement algorithm is simulated annealing. Simulated annealing is a technique based on metallurgical cooling. It iteratively improves an initial placement to find the optimal solution. These small iterations take a significant amount of time before settling into the final placement. Versatile place-and-route (VPR) [24] uses a simulated annealing placement algorithm that is used as a metric for comparing placement algorithms. VPR can find good solutions, but run-time is long. As such, there are analytic placers [42] that achieve competitive results against VPR's simulated annealing methods, while reducing

run-time.

Prior placement strategies for modular design included random and simulated annealing techniques [43]. However, one drawback of these techniques is that they are heavily platform specific, and thus, are locked into the Virtex 5 device. This is despite the fact that this modular flow otherwise uses the TORC framework [23], which should allow for platform flexibility.

Modular design lowers the granularity of the placement space due to module collision avoidance. Therefore, device utilization will be poorer than when global optimization is possible [44]. Thus, properly designed modules and a good placement strategy are necessary to overcome this hurdle.

A deterministic placer will ensure that a design that meets the time requirements will continue to do so no matter how many times it is run. QFlow's [43] placer is non-deterministic and can take differing amounts of time for the same design. In addition, QFlow does not support recent devices. HMFlow [28] is not designed with hard time requirements in mind.

As iterative placement requires a significant time investment, the presented module placer will use a constructive placement algorithm. As this problem is well suited to the fast placement times and high granularity of constraint-based placement, this work will present a constraint-based solution for rapid assembly within a time budget. In addition, the algorithm will be deterministic, to ensure consistent results.

2.3.1 Constraint Generation

Generating many of the necessary constraints for the placer can be done during modular design. These constraints will include the set of valid placements for a module on the target device. One prior approach generated the valid placements using the Xilinx Relatively Placed Macros (RPM) grid [43]. This grid is different for every design, but Frangieh created a method for generating this coordinate system for the Virtex 5 architecture. Unfortunately, building an RPM grid for other architectures is a manual process. Instead, a different placement coordinate system, based on Xilinx tiles, is built into the TORC framework and thus works with all TORC-supported devices. This tile-based coordinate system will therefore be used for this work.

By creating these constraints prior to assembly, less computation will be needed at assembly time, increasing the speed at which assembly can occur. Not all of the constraints can be precompiled; these additional constraints will be generated during assembly time. For example, the number of valid placements will be reduced, because some regions of the device will be unavailable for module placement at assembly time.

2.4 Modular Design

If a progression of the densest FPGA devices over the past two decades is considered, backend compile time has remained nearly constant. There have been notable improvements

in EDA algorithms over this period, yet these have not kept pace with device densities. In recent years, much work has been done to improve the efficiency of the front-end processes of design entry and synthesis. Xilinx has developed System Generator for DSP [45] to capture a design and convert it to Hardware Description Language (HDL). Impulse Accelerated Technologies [46] created a C-to-HDL flow for FPGAs. Instead of directly coding in HDL, high level abstractions like C or system block diagrams are successfully being used to reduce the time for design entry. Research into incremental synthesis started as early as the 1990s [47]. Commercial synthesis tools like Xilinx XST and Synopsis Symplify have long supported incremental compilation, which sharply decreases the time required to synthesize a modular FPGA design.

Back-end post-synthesis processing consumes a large portion of the full FPGA development flow; with 90% of the compile time spent on FPGA place and route [27]. Therefore, reduction in the computation time for the back-end flow would result in the largest gains. Early work on improving this computation time was done using VPR [24], but this tool was not implemented on real devices. Incremental techniques have been exploited for single back-end steps. [48] and [49] investigate incremental techniques for the mapping stage of lookup table (LUT) based FPGAs. [50] and [51] explore incremental placing algorithms. [52] and [53] develop algorithms for incremental routing. While these techniques are effective for improving portions of the back-end flow, TFlow is focused on deadline assembly for the entire back-end. Another approach is to reuse precompiled modules, a technique that is used in [54] and [28]. Hortal and Lockwood [54] propose the idea of Bitstream Intellectual Property (BIP) cores. BIP cores

are precompiled Intellectual Property (IP) modules that are represented as relocatable partial bitstreams. HMFlow [28] creates precompiled modules that are represented as hard macros in the Xilinx Design Language (XDL), a human readable format for physical level information. As in [54], the precompiled modules in TFlow are represented as bitstreams. However, [54] is essentially a Xilinx Partial Reconfiguration (PR) flow and thus has limited flexibility. The modules only fit inside a few pre-defined regions. These regions are specifically for modules and are known as sandboxes. Inter-module connections must match specific bus macro interfaces with fixed routes. If a design needs a new module, the full vendor tool flow needs to be run on the whole design again, although other modules in the design may not have changed. By contrast, TFlow does not use the vendor's partial reconfiguration model; hence, it does not require fixed-location sandboxes or fixed-location bus macros. Modules can be relocated to wherever there are enough resources available and can dynamically route the connections. New modules are compiled independently, improving parallelism.

This contrasts with the Xilinx PR flow [55], where a module is compiled with respect to a single design framework. This framework is the static design. For a module to be used in a different design framework, it must be recompiled. TFlow can reuse modules between static designs. Xilinx PR uses an island-style approach to modules, with each sandbox only permitting the placement of one module at a time. TFlow has no such restriction. Additionally, Xilinx PR is intended for run-time reconfiguration, while TFlow assembles full bitstreams off-line.

OpenPR [56] is an open-source run-time reconfiguration tool that follows the same approach

as the Xilinx PR flow, and has many of the same drawbacks. It has an island style approach, with only a single module permitted in each designated static sandbox region. As such, this is not a design assembly tool, but a run-time reconfiguration tool.

Another approach, *Wires-on-Demand* [57], is also designed for run-time reconfiguration. It has a slot-less model with reserved routing channels between modules. TFlow is not a run-time reconfiguration tool, and it is also a more general solution to modular design, as it does not require reserved routing channels. Instead, TFlow has its own inter-module router.

GoAhead [58] is a partial reconfiguration framework designed to efficiently build run-time reconfigurable systems. It is built as a newer, upgraded version of ReCoBus-Builder [59]. As such, it has some similarities with modular design flows. The design process is split between a static design and modules. These components can be built in parallel to speed up design generation. GoAhead does not have a placer or a router. Instead, selecting placements is done manually. For routing, GoAhead uses a route blocker. Instead of blocking all routes, it blocks all but one for a specific connection. When the Xilinx router is run, these blocking routes force the router to use the only remaining path for each route. In so doing, GoAhead can constrain an exact path for each route in the sandbox. The same holds true for the clock lines; the clocks are prerouted into the static for integration with the modules.

As with other flows, placements need to match the resource pattern of the original module to be valid. In addition GoAhead requires that the static design has specific existing routes that pass through the desired placement slot. This is due to the fact that routing is not performed at design time. Instead, routing is static, and the modules interrupt this existing path. This

is effective, so long as no changes to routing are desired. In addition, every sandbox slot must have these routes passing through them to be valid placements for a module.

GoAhead's modules and static design are thus tightly coupled due to these shared routing prerequisites. While the order of modules can be changed, the inter-module connectivity cannot be changed without recompilation of either the module or the static. Since the routing is predefined, if new fan-out or different connectivity is desired, recompilation is required. In contrast, other modular design flows can reroute connectivity without recompilation; inter-module connectivity is performed at design time, not during precompilation.

Due to the tight coupling of the modules and the static in GoAhead, they are not truly independent. These components must be built to a routing standard if they are to be compatible. In comparison, other modular flows can build modules that are compatible with a generic static.

The Dynamically Reconfigurable Emdedded Platforms for Networked Context-Aware Multimedia Systems (DREAMS) design flow builds and assembles modules for partial reconfiguration [60]. The goal of DREAMS is to minimize the need for human intervention in the design process. Modules and static designs can be built independently, and module relocation can occur. Modules are built such that their interfaces are directly compatible when placed adjacent to one another. This is ensured through the use of the DREAMS router. Modules have specific interfaces at each of their four boundaries for connectivity. So long as these interfaces are compatible and the modules properly align, they can be relocated within the placement grid. Connectivity is therefore limited; modules cannot connect to more than

four others. Modules must be built to be compatible with one another and the static. Any changes to the module interfaces will cause recompilation of all compatible blocks. On the static side, the reconfigurable region is manually specified by the designer.

QFlow [29] uses modular design like TFlow, but is not as well-optimized for speed. Whereas TFlow generates modules for the library at the bitstream level, QFlow stops after the map stage. These modules must then be routed at run-time, slowing down design assembly. By using unrouted modules, QFlow gains routing quality at the cost of speed.

Both HMFlow [28] and TFlow make use of XDL (the Xilinx Design Language). HMFlow stores all of the module information in XDL, including logical instances, their placements, and their routing. TFlow, however, mainly uses XDL to as an input mechanism for its meta-data, extracting physical and net information. More importantly, to create a full design, HMFlow must convert the final design XDL it produces into a Netlist Circuit Description (NCD), a Xilinx physical description file, before it is able a bitstream for use on a device. This XDL-to-NCD conversion and subsequent bitstream generation takes considerable time and scales with the size of the design. Tests will show that this overhead exceeds the total TFlow runtime, acting as a performance bottleneck (see Section 4.2.2). This makes meeting the performance deadline impossible when using HMFlow's approach. Instead, TFlow, like QFlow, uses a different approach that does not require this costly XDL-to-NCD conversion process and thereby speeds up bitstream creation.

Building higher level design flows requires significant low-level knowledge of the device architecture. TORC [23] and RapidSmith [61] are two open-source tools that have been built to

enable this capability. These two tools are analogous; some capabilities differ, but the main distinction is that RapidSmith is a Java framework and TORC is C++. HMFlow [28] and DREAMS [60] both build on the RapidSmith framework [61]. QFlow [29] and TFlow [40] build on TORC [23].

GoAhead [58] also builds on top of prior work in the form of the ReCoBus-Builder tool [59]. However, while this tool is publicly available, it is not open-source, and thus is not available as a framework for additional work.

2.5 Flow Comparison

Table 2.1 compares the goals of different design flows. Xilinx ISE aims for the highest quality results. The design can be tightly packed and have high utilization, global optimizations can consolidate or improve the design, and clock rates can be pushed to their limits. These are all important considerations, but other goals may have precedence depending on the application. Xilinx ISE also supports partial reconfiguration.

QFlow [29] uses modular design to attempt to speed up the back-end design flow. The modules it uses are unrouted, but are otherwise locked to a specific resource pattern. While QFlow is faster than ISE, it does not meet the necessary time requirements for attention span assembly. This is because, post-placement, QFlow uses ISE for routing and bitstream assembly.

HMFlow [28] also aims for rapid assembly. In fact, it is faster than QFlow at generating a

design because its modules are pre-routed. However, design restrictions mean that HMFlow cannot meet the attention span deadline either.

Conceptually, TFlow treats modules as immutable objects. Once created, their resources are reserved and their shape is locked. In contrast, HMFlow does not have this constraint. Instead, it manipulates internal module logic; in [62] it does so to obtain better timing results. Conceptually, HMFlow treats modules as convenient logic groupings, not immutable blocks.

HMFlow's module XDL must also include internal logic and resource utilization. In comparison, TFlow's module meta-data only needs the shape of the module for assembly. This can be an asset when dealing with proprietary modules, because in TFlow the internal logic does not exist in the meta-data, whereas HMFlow's approach requires this information to generate the final bitstream. Conceptually, TFlow's design approach is better suited for proprietary assembly.

GoAhead [58] is a PR flow, and is design accordingly. The modules are immutable; they are stored as bitstreams. However, GoAhead's goal is to improve on Xilinx PR, not the Xilinx ISE tool. As such, its features are designed for more capable and flexible partial reconfiguration and not for back-end acceleration or deadline assembly.

TFlow is designed from the ground up towards meeting the attention span deadline and has succeeded. To do so, it uses immutable modules and assembles them with an eye towards meeting these time constraints. Device utilization and packing are sacrificed for time.

Table 2.1: Flow Goals Comparison

	Quality	Partial Reconfiguration	Immutable Modules	Speed	Meets Attention Span Deadline
ISE	x	x			
QFlow			x	x	
HMFlow				x	
GoAhead		x	x		
TFlow			x	x	**x**

2.5.1 Flow Capabilities

Table 2.2 compares the capabilities of different design flows. Xilinx ISE is the reference, as it has full functionality, except for running in an embedded environment. To duplicate this functionality while speeding up execution time is the goal of these flows. QFlow [29] has its own placer, but routing and bitstream generation are done by reintegrating with the Xilinx toolchain. As such, it is not a standalone environment. HMFlow [28] goes further and has a router as well, but must return to the Xilinx tools for bitstream generation of the design.

GoAhead [58] is an interesting case, because it is designed as a PR flow and not a modular flow. It does not have a placer or a router, but it does create a library of partial bitstreams for use during run-time for partial reconfiguration. As such, these partial bitstreams could be used in an embedded environment, although the GoAhead tool would not run. To contrast, TFlow [40] implements placement, routing, and bitstream generation. TFlow assembly is a standalone process that can occur in an embedded environment.

To look at these flows from a different perspective, placement is a constant for modular flows. Only GoAhead does not implement one, but it is designed to be a PR flow first and

Table 2.2: Flow Capabilities

	Placer	Router	Bitstream*	Standalone	Embedded
ISE	x	x	x	x	
QFlow	x				
HMFlow	x	x			
GoAhead			x		x
TFlow	x	x	x	x	x

*Generates a bitstream at design time without using outside tools

foremost. Routing the design without reliance on the Xilinx tools is done by HMFlow and TFlow. Having this capability removes the need for either Xilinx to perform the routing, as in QFlow, or for there to be pre-built static routes for connectivity, as in GoAhead. When looking at bitstream generation, only ISE and TFlow generate bitstreams at assembly time. GoAhead is marked as having bitstream support because the partial bitstreams it generated when creating the modules are drop-in components to the flow. It does bitstream-level module relocation, but no bitstream-level routing.

Only TFlow offers a non-Xilinx standalone design assembly flow. This removes Xilinx from the equation and gives significant control back to the flow. As assembly is not reliant on the closed-source Xilinx tools, no license is necessary. In addition, this is what enables TFlow to run on embedded platforms. GoAhead's library of partial bitstreams can also be run on an embedded environment, although the actual GoAhead tools might be incompatible.

2.6 Meta-data Description

To best describe a full design, information regarding the logical and physical properties is necessary. In most current representations, only a subset of this information is in any one place. In [63], a module-based design strategy with core reuse and abstracted linking between modules is presented. To describe these cores in an abstracted manner, a set of meta-data is created using the IP-XACT standard. However, this information solely describes the system at the logical level.

A standard netlist also represents a module or design at the logical level; this information is then passed to another tool where physical information is created. For Xilinx tools, physical information can be read in the form of XDL. XDL contains a full picture of the physical device, but in so doing it simplifies away much of the logical level information. Doing so can optimize the solution and remove unnecessary information. However, when attempting to reuse cores at the bitstream level, both physical and logical level information is necessary.

One representation technique that combines both physical and logical level information was presented by Tavaragiri [64]. This technique meets all of the requirements for describing a design, and so will be adopted in an updated form by TFlow. However, automatically generating this information was not supported. A logical-level import tool was built [65] to initialize the meta-data from a netlist. Automatically back-annotating the meta-data with the full design data is implemented in this work.

2.7 Bitstream Relocation

Xilinx device programming is done through a file called a bitstream. This bitstream contains information on how every part of the FPGA will be configured, from the routing resources to the values in look-up tables. The mapping for a specific logical value is proprietary, but the general structure of the bitstream is not [66].

Bitstreams for Xilinx devices are split up into frames. Each frame is one clock region high. Clock regions are named because the clock routing resources have a horizontal spine that serves all of the resources in a clock region. The height of a clock region is dependent on the device architecture. For the Virtex 5, a clock region is 20 tiles high.

Each frame has a 32-bit address that uniquely identifies its position on the device. Figure 2.1 shows the frame addressing scheme for the smallest Virtex 5 device, the XC5VLX20T. The clock region is represented as the Major Row, and the tiles are ordered by column. A single frame is insufficient to represent all of the data inside a tile, so a set of minor addresses are used to determine which internal column of the tile this frame represents. From the figure, a full frame tile is circled. This tile contains Configurable Logic Blocks (CLBs), and so requires 36 frames to fully describe the configuration.

Since these frame addresses are known, they can be changed to move the frame contents to a new location. With the circled set of CLB tile frames of Figure 2.1, if the ”Bottom Bit” value in the frame address is changed from a ’1’ to a ’0’, the CLB frame contents would move to the top of the device. In this way, module relocation can occur. Of course, since

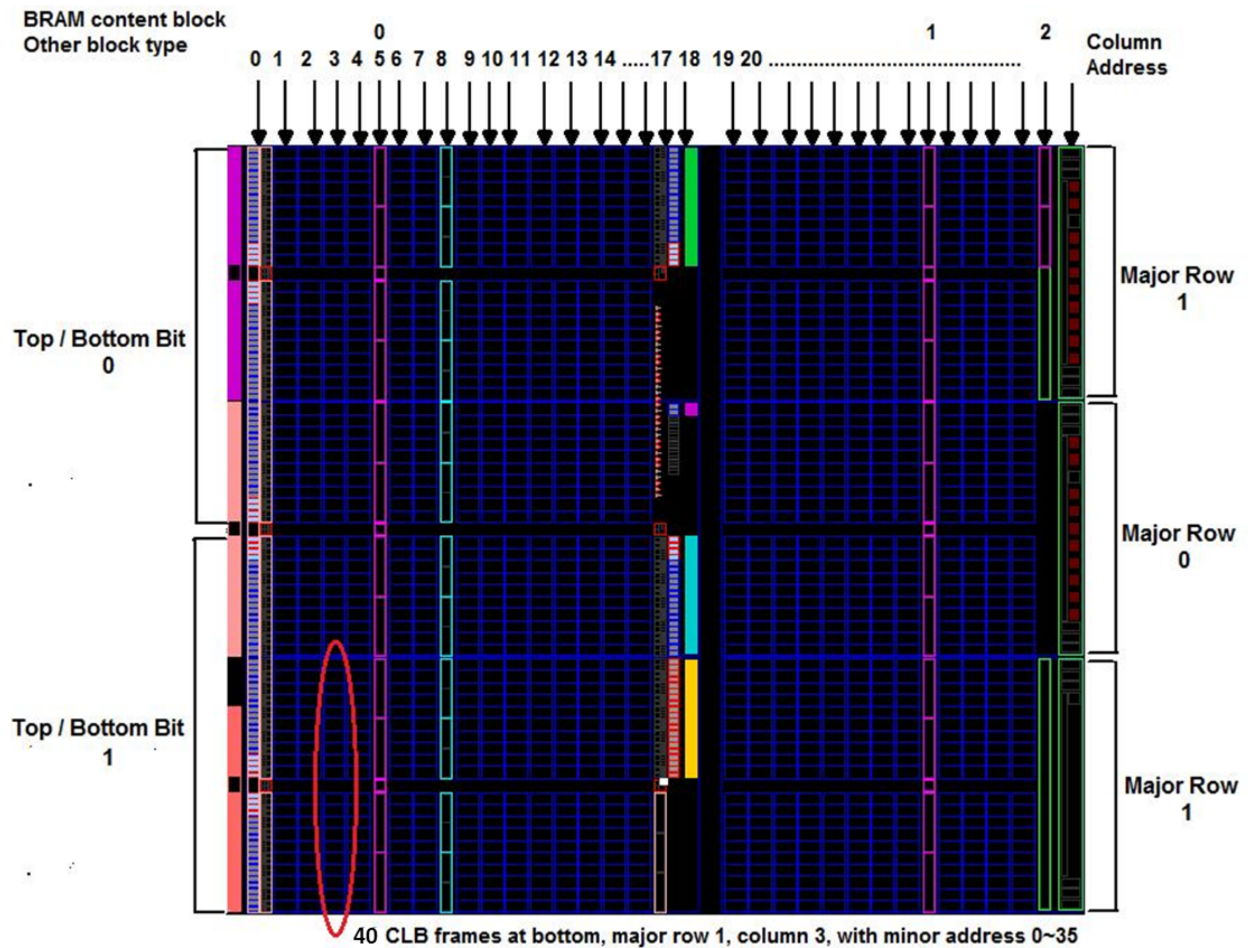

Figure 2.1: Xilinx XC5VLX20T Frame Layout

these frames represent a specific type of tile, in this case CLBs, the new frame location must match or else the behavior is unknown.

This process is known as bitstream relocation, which is a well-explored capability for Xilinx devices [67] [68]. Since bitstream relocation only changes the frame address, it does not require any proprietary Xilinx knowledge of the contents of the frame. As such, any information about what the frame represents would need to be extracted before bitstream generation, which is the purpose of meta-data. One important note about relocating a frame

to a new location is that the new location should be empty. If it has logic or routing, this information will be lost when the new frame overwrites it.

Bitstream relocation only moves frames around. Should additional routing be required, as it is in TFlow, a method to add this to the bitstream is necessary. [69] presents a method of generating routing micro-bitstreams. Each micro-bitstream represents a specific wire segment on the FPGA. When multiple micro-bitstreams are logically OR-ed together, they can create a full route. Knowing which wire segments to select is a router's job. These micro-bitstreams can be logically OR-ed with an existing design to add routes to this design. This capability does not allow for removing routes, so the desired path must be free of routing.

By combining bitstream relocation with micro-bitstreams, new bitstream designs can be created from pre-built pieces. Without micro-bitstreams, routing between relocatable modules would need to occur on pre-defined interfaces, as occurs with DREAMS [60] and GoAhead [58]. This new capability makes relocated bitstreams much more flexible, since communication is no longer a bottleneck and dynamic routes can be produced. Leveraging this work into a larger productivity flow will enable full design assembly without the need to return to the standard tool flow.

2.8 Summary

TFlow builds on many of the successes and lessons of prior work. To improve the compilation speed of design assembly, prebuilding modules and static designs are necessary, as seen in

QFlow and HMFlow.

The Dreams flow impressed the importance of minimizing human intervention in the design process, and so automation of placement footprint selection was implemented.

HMFlow shows the drawback to relying on the Xilinx tools for bitstream generation, in that it adds significantly to the assembly overhead. Therefore, methods of bitstream manipulation were explored.

Bitstream manipulation techniques for both relocation and routing are needed to generate the final design for programming the physical FPGA. These techniques need to be seamlessly integrated into the flow.

Once the designs are represented as unreadable bitstreams, a meta-data description of the design is necessary. This meta-data must contain all of the necessary information for the design assembly process. Prior work has shown what sorts of information are necessary and how best to represent it.

Each of these lessons are necessary to meet the attention span deadline assembly time requirement.

Chapter 3

Implementation

Modular design is an approach that subdivides a system into smaller parts – or modules – that can be independently created and then combined and recombined into different systems. Designs for FPGAs typically consist of a hierarchy of primitive elements combining to form larger components until eventually resulting in a full system. The benefits of modular FPGA design become apparent when an incremental change or expansion of a design is required [70] [71].

To gain the most benefit from modular design speedup, as much work as possible should be completed prior to the iterative phase of the design process (e.g., during module creation). The motivating principle behind this work is to meet the deadline imposed by the human attention span. As such, it aims to complete as much back-end computation as possible ahead of time, even if this makes module preparation more computationally expensive. Full

compilation of modules yields module-level bitstreams for later iterative assembly.

TFlow moves computation earlier by precompiling modules, in many permutations, into a library. These precompiled modules can then be stitched together during design assembly. Use of precompiled components dramatically decreases design assembly time. These modules are analogous to software libraries, where precompiled functions are used to reduce compile time. This analogy must be extended when applied to FPGAs since the additional steps of module placement/relocation and inter-module routing are required. TFlow's precompiled modules can be reused in different designs, a capability not common to all modular flows. This modular reuse mimics the technique of code reuse in software development, a method proven to increase productivity [72] [73].

Another productivity technique for software design is a rapid feedback loop. A user can change a design, compile it, and run it within a very short time frame (on the order of seconds). This positively impacts the number of turns-per-day. Additionally, this capability provides a psychological change in behavior where the user may make many changes incrementally and interactively, encouraging the user to explore more design alternatives. Standard FPGA design flows can take a considerable time to complete, which reduces the feedback loop to the order of minutes to hours. In some cases, small changes to the design require full re-compilation. This reduces the number of turns-per-day, restricting the time available for prototyping. TFlow aims to reduce FPGA compilation times down to the software speed of seconds. This increase in the speed of compilation will improve the number of turns-per-day. In addition, TFlow's driving goal is to complete design implementation within

the narrow window of the user's attention span and thereby increase productivity [19].

3.1 Flow Motivation

Implementing the design within the window of human attention span enables significant productivity enhancements to the design process. TFlow must be able to meet this requirement, and it does so by splitting up the design flow into work that can be done beforehand and work that must be done at assembly time. This design split will significantly speed up assembly time. The majority of the computation time is thus front-loaded into *library creation*. While other techniques have used this same strategy, none of them have met the strict time requirements necessary to maintain the user's attention. To meet this deadline and improve upon the other attempts, the modules are compiled completely. This consists of running the modules through the full compilation flow, from synthesis through bitstream generation. These bitstreams are pushed into the module library. TFlow's maximally compiled modules can be assembled to create a full bistream design in seconds, well within the thirty second attention span window.

3.2 Flow Design

TFlow gets its large productivity boost by splitting the flow into two distinct phases, as seen in Figure 3.1. The first phase, referred to as the *module creation phase*, occurs when new

functionality is needed. This phase is intended for an HDL programmer, or a *librarian* [74]. This librarian designs a module that supplies this capability using a front-end tool, much like a dynamically loadable library in software development. This module design is then passed to TFlow, which shapes it and passes it into the vendor flow. This creates a bitstream and meta-data that is stored in a component library for later use.

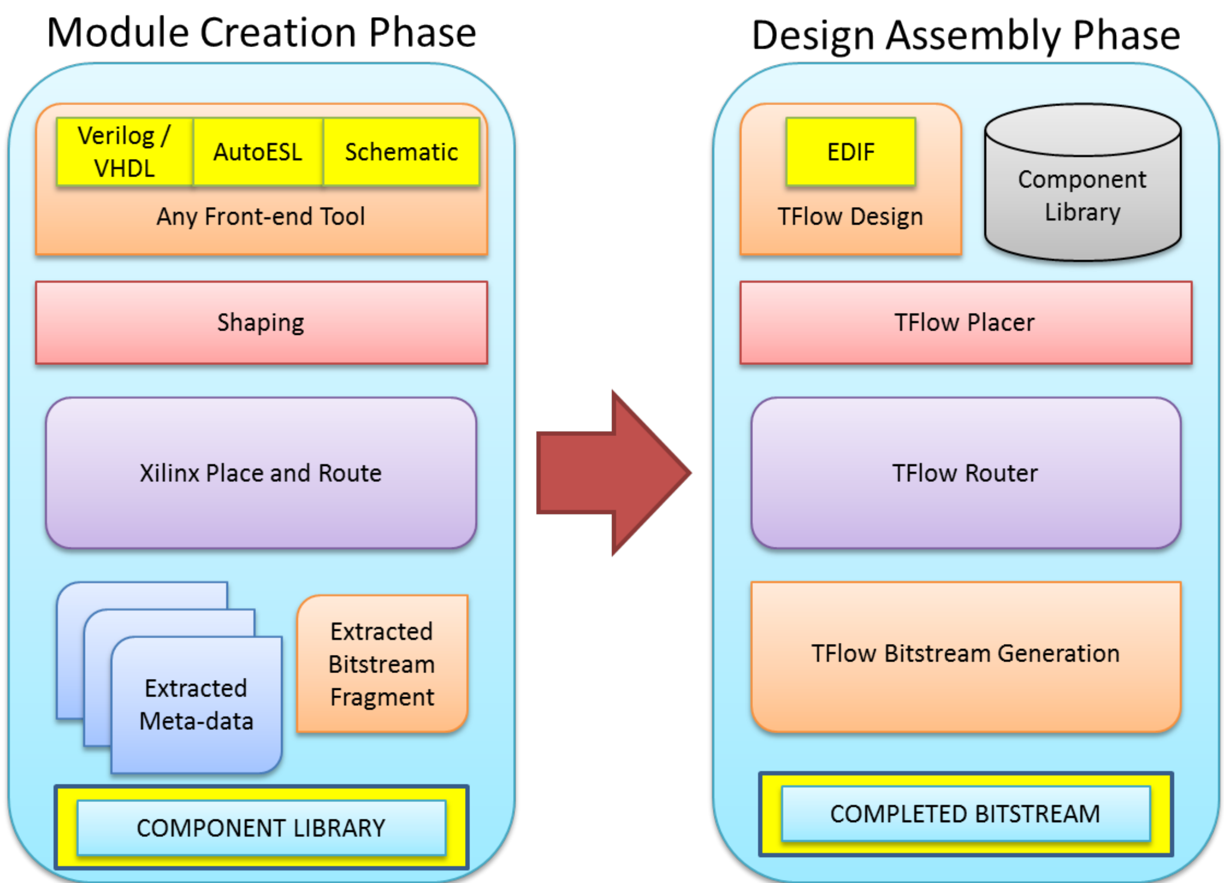

Figure 3.1: TFlow Use Model

The second phase, referred to as the *design assembly phase*, is performed by the engineer implementing the design. This designer creates high-level plans consisting of a set of modules representing a final design. TFlow will fetch the pre-created modules and assemble them

using the TFlow design assembly process. The resulting bitstream can then be loaded onto an FPGA. So long as the necessary components are in the library and the library components are sufficiently parameterizable, the design loop can be traversed quickly for rapid prototyping. These phases will be discussed in further detail.

The decision as to where design ends and implementation begins is a mutable one and depends on perspective. The design (or front-end) process can be the requirements documentation, with the rest of the process considered implementation details. On the other hand, it can also consist of everything prior to actually setting the bits on a device, where this programming is considered the implementation (or back-end).

This work takes the more common middle ground, where the back-end process begins after synthesis. Synthesis consists of translating a design into a set of primitive blocks and their logical connections. This process can be device-agnostic, although in most cases device-specific optimizations are used. The result from synthesis can be represented as an Electronic Design Interchange Format (EDIF) file, which is a standard netlist format [75]. This standard is front-end and vendor agnostic. The first phase of TFlow begins with an EDIF netlist.

Any front-end tool that ultimately produces an EDIF netlist can be used with TFlow. This includes standard HDL flows, C-to-gates flows, and graphical front-end flows. The vendor implementation flow, including the mapper, placer, router, and configuration bit generator, is run once for each module, incorporating user constraints. Key attributes of the module, such as Input/Output (I/O) interface, anchor point, and resource usage, are extracted from the EDIF using custom tools based on the *Tools for Open Reconfigurable Computing* (TORC) [23]

and stored in a meta-data file as Extensible Markup Language (XML). The vendor flow, including the appropriate bit generation tool, is used to create a module bit file. The configuration bit file and the meta-data for each module are stored in the module library.

The second phase of TFlow is design assembly. To build a design, the user only has to specify which modules to use and how to connect them. Using the same EDIF format to create a design description, the assembly flow then fetches these modules to create the design. This assembly flow goes through TFlow's placer, router, and bitfile generation phases. No time-consuming vendor tools need to be run, enhancing platform flexibility. A modified TORC router is run to make the inter-module connections. These connections are done at the bitstream level.

The compilation time saved by TFlow can be seen by comparing it to the traditional Xilinx back-end compilation flow. Starting from the initial synthesized netlist, the traditional flow has four phases. Initially, the design is mapped, which converts the post-synthesis logic gates to FPGA primitives. Then, these primitives are laid out onto the target FPGA device in the placement phase. Next, the primitives are connected together with wires in the routing phase. Finally, the bit generation phase converts the connected primitives into a final bitstream that will be used to program the FPGA. TFlow reduces the time required for each of these phases.

The mapping of logic to slices and physical components can be skipped entirely since the modules in the library have already been compiled. No additional mapping is required during assembly. The placing time is significantly reduced due to the coarser granularity of module

placement versus the slice or logic placement of the standard flow. These modules have a selection of specific sizes and shapes, but are still restricted to a limited number of possible locations due to the properties of the module. This does mean that some excess area is used due to the inability to do cross-module or global optimizations. No overlap is permitted between modules. As modules were internally routed during module compilation – using vendor tools – the routing phase is reduced to inter-module connectivity. Lastly, bitstream generation time is reduced since a modified bitstream merge is performed, replacing the full vendor-provided generation process.

Five aspects of TFlow distinguish it from conventional back-end compilation:

1. It boosts the productivity of FPGA assembly by significantly reducing compilation time. This increases the number of turns-per-day possible for designers;

2. It explores the possibility of applying software engineering practices to FPGA development, in this case a library of precompiled components;

3. It demonstrates the power of using TORC to augment or enhance vendor-supplied compilation flows;

4. It broadens the applicability of FPGAs to a wider selection of applications; and

5. It further boots productivity by completing compilation within the user's attention span.

By speeding up the number of turns-per-day and reducing the complexity of the design pro-

cess, FPGA design can begin to attract users accustomed to software design. Consequently, FPGAs can expand into fields that would otherwise choose to remain pure software, such as emerging applications like GNU Radio [76].

TFlow is not without some drawbacks. Timing for the routes that TFlow creates can be longer than that of the vendor tools, as there is no global optimization. Similarly, there may be resource optimization issues, because modules cannot share resources. Additionally, since modules cannot overlap, any unused resources in a modules footprint are unavailable to the design.

3.3 Flow Model

A model of the computational effort required for the full design flow is necessary to best determine how to meet the assembly deadline. With this model in hand, TFlow can properly perform modular design and determine what work can be performed ahead of time - during *Module Creation* or *Static Creation* - and what the remaining run-time effort will be for *Design Assembly*. This model will then be applied to multiple modular design approaches to determine suitability.

For the model, D is the total design computation effort. Computational effort is the amount of work necessary to complete a computation. In other fields, the joule represents the amount of effort required to move an object one meter using one newton of force. For this work, computational effort is defined as the running time needed to reach a solution [77].

The computational effort required can be divided into any number Z of stages, T, depending on the properties of the flow. T_n is the computational effort necessary for the nth stage.

Equation 3.1 represents the baseline modular design flow which does not perform any precomputation. While it is likely that there are some savings possible by combining stages together, it is assumed at this point that any benefit is minimal.

$$D = T_1 + T_2 + \cdots + T_Z \tag{3.1}$$

To obtain the appropriate complexity reduction, the design process is split into three different portions. *Static Creation* S and *Module Creation* M both occur prior to design time. *Design Assembly* A consists of the computations that must be performed at run-time. Each stage T will contain S, M, and A. Equation 3.2 represents how any stage T_n can be split into these precomputed and run-time portions.

$$T_n = T_{nS} + T_{nM} + T_{nA} \tag{3.2}$$

One additional factor that can impact the gains from splitting the design process are global optimizations, G. This influences Equation 3.2, transforming it into Equation 3.3.

$$T_n = T_{nS} + T_{nM} + T_{nA} - G_n \tag{3.3}$$

This means that there is some overhead added when splitting the design stages, as some optimization cannot be performed. The gains from precompilation must therefore be judged against the total time requirements to determine the comparative size of G_n. Splitting the stages is contraindicated only when $T_{nA} \geq T_n$. Otherwise, extra computation in S or M is acceptable, as it occurs during precompilation. This adds another factor when performing minimization of A.

Design time thus consists only of performing those computations that occur during assembly, A. The equation for this computational effort is shown in Equation 3.4.

$$A = T_{1A} + T_{2A} + \cdots + T_{ZA} \tag{3.4}$$

From this, it can be seen that the more computations that can be moved into S and M, the less that will be necessary at design time. Maximizing S and M while minimizing A without removing the desired flow capabilities is the desired goal.

3.3.1 Model Application

The model presented in Section 3.3 can be applied to the various existing flows to determine how well they meet the requirements of reducing computational effort and thus enabling hard deadline-based implementation.

To apply the model properly, the number of stages needs to be determined. The best fit to the

existing flows are three stages. The first stage is P, which represents module logic mapping and placement. The next step, R, is the stage responsible for connecting the modules to one another. The last step, B, compiles the design into the final device-usable form - the bitstream.

Equation 3.5 represents the required computations necessary to compile a post-synthesized netlist into a bitstream. This ignores precompilation, and is a representation of the Xilinx standard design flow, ISE, or I.

$$D_I = P_I + R_I + B_I \tag{3.5}$$

Equations 3.6, 3.7, and 3.8 break down the design time still further, into three distinct portions: static creation, module creation, and design assembly. For this flow, all of these steps occur at design time; no preprocessing is done.

Each stage is split between three different portions of the design: the static design S, module creation M, and design assembly A. When compiling an ISE flow, each step is performed atomically. Global optimization occurs, but this is not visible at the scale of the model.

$$P_I = P_S + P_M + P_A \tag{3.6}$$

$$R_I = R_S + R_M + R_A \tag{3.7}$$

$$B_I = B_S + B_M + B_A \tag{3.8}$$

The computational effort required to compile a design in ISE can therefore be represented as shown in Equation 3.9.

$$D_I = P_S + P_M + P_A + R_S + R_M + R_A + B_S + B_M + B_A \tag{3.9}$$

Contrast this with the way that TFlow, T, is structured. TFlow performs many calculations prior to design time. This preprocessing can be seen in Equations 3.10 and 3.11. In these preprocessing steps, all of the static and module calculations are performed.

$$S_T = P_S + R_S + B_S \tag{3.10}$$

$$M_T = P_M + R_M + B_M \tag{3.11}$$

This leaves few calculations to occur during design assembly, as seen in Equation 3.12.

$$D_T = P_A + R_A + B_A \tag{3.12}$$

The difference in the number of calculations required between ISE and TFlow can be seen in Equation 3.13. As seen, since many of the TFlow calculations occur prior to assembly time, there are significant savings.

$$D_I - D_T = P_S + P_M + R_S + R_M + B_S + B_M \tag{3.13}$$

QFlow [29], Q, prebuilds unrouted hard macros as its modules and then assembles the design and performs routing and bitgen with the Xilinx tools. Equations 3.14 and 3.15 show the work done prior to design time, while Equation 3.16 shows the work necessary at design time.

$$S_Q = P_S + R_S \tag{3.14}$$

$$M_Q = P_M \tag{3.15}$$

$$D_Q = P_A + R_M + R_A + B_S + B_M + B_A \tag{3.16}$$

Comparing QFlow with TFlow, it can be seen that routing and bitsteam generation are additional QFlow computations; thus QFlow will take longer to complete than TFlow.

$$D_Q - D_T = R_M + B_S + B_M \tag{3.17}$$

HMFlow [28], H, builds both placed and routed modules, but it populates its library with XDL. The static and module work can be seen in Equations 3.18 and 3.19.

$$S_H = P_S + R_S \tag{3.18}$$

$$M_H + P_M + R_M \tag{3.19}$$

Assembling these modules also occurs in XDL. The result must then be converted to the correct format for Xilinx Bitgen. To do so, additional conversion overhead, O_H, represents running Xilinx `xdl2ncd`. This is represented in Equation 3.20.

$$D_H = P_A + R_A + O_H + B_S + B_M + B_A \tag{3.20}$$

TFlow does not have O_H, and already has prebuilt bitstreams. Equation 3.21 has the difference in computational requirements between these flows. So long as Bitgen and O_H remain computationally intensive, TFlow will require less computational effort than HMFlow and thus complete faster.

$$D_H - D_T = O_H + B_S + B_M \tag{3.21}$$

3.3.2 Model Preprocessing Trade-Offs

This model can be analyzed to determine the preprocessing trade-offs. Table 3.1 covers each term in the model from Equation 3.9 and the impact from preprocessing. This table summarizes the trade-offs from preprocessing. For example, TFlow precompiles every term except for P_A, R_A, and B_A. This reduced the computational effort below the attention span deadline while maintaining flexibility. The effect of precompiling these terms can be seen throughout this work. However, the precompiling the remaining three terms also trade-offs to consider. OpenPR [56] performs P_A during preprocessing by having a designated slot

Table 3.1: Model Terms and Impact

	Preprocess Effect	Design Time Computations	Preprocess Evaluation: Pros	Preprocess Evaluation: Cons
P_S	Preplace Static	Map / Place Static	Less computational effort	Static locations and resources locked; Lacks global optimization
P_M	Preplace Modules	Map / Place Modules	Less computational effort	Module footprint locked; Resources reserved; Lacks global optimization
P_A	Preselect Design Placement; Slot-based design	Flexible Placement	Less computational effort; Known module location	Inflexible; Module takes full area despite size; One-to-one mapping
R_S	Preroute Static	Route Static	Less computational effort; Internal timing can be guaranteed	Routes may cause collisions
R_M	Preroute Modules	Route Modules	Less computational effort; Internal timing can be guaranteed	Routes may cause collisions; Module relocation restricted to locations with appropriate routes
R_A	Preroute Design; Route blocking; Preset route interfaces	Route Design	Less computational effort; Timing can be guaranteed since routes are known	Less flexible connectivity; Limited number of prebuilt connections
B_S	Precompile Static Bitstream	Compile Static Bitstream	Less computational effort	Metadata description of static required
B_M	Precompile Module Bitstreams	Compile Module Bitstreams	Less computational effort	Metadata description of modules required
B_A	Precompile Design Bitstream; Select from library of full bitstreams	Compile Design Bitstream	Minimal computational effort; High quality design	No flexibility; Only prebuilt designs available

for the module. Only one module is permitted in each slot, so excess area is wasted but placement can proceed rapidly. The DREAMS flow [60] performs R_A during preprocessing, sacrificing routing flexibility for simplified connectivity. Performing B_A during preprocessing is equivalent to precompiling full designs and having a selection of bitstreams to load at design time. Changing the design requires recompilation, but if the design space is predictable, bitstreams can be supplied instantly. Depending on the design philosophy and goals for a flow, preprocessing any or all of these terms can be desirable. Flow designers can use Table 3.1 to make an informed decision.

3.3.3 Module Reuse Model

To model the improvement that a modular flow has over a standard flow with respect to reuse, consider Equation 3.22, where N is the total number of modules. Each of the modules must be built. Now compare this with Equation 3.23, a modular flow, where U is the number of unique modules. The improvement given is due to modular reuse, even within the same design. From Equation 3.24, the modular flow can be seen to perform at least as well as the standard flow. As the number of duplicated modules grows, the modular flow's advantage grows likewise.

$$M_I = \sum_{i=1}^{N} M_i \tag{3.22}$$

$$M_F = \sum_{i=1}^{U} M_i \tag{3.23}$$

$$U \leq N, M_F \leq M_I \tag{3.24}$$

3.3.4 Module Parallelism Model

Equation 3.25 shows the behavior of a modular flow when each module is independent and can be built in parallel. Since this is a perfectly parallel process, the amount of time spent on modular design is equal to the maximum amount time it takes to build any single module. From this, it can be seen that the more modules that are built in parallel, the larger the gain that a modular flow has over the standard flow, as represented by Equation 3.26.

$$M_{F|||} = MAX(M_1, M_2, \ldots, M_U) \tag{3.25}$$

$$U \gg 1 \iff M_{F|||} \ll M_I \tag{3.26}$$

3.3.5 Strategy to Meet a Hard Placement Deadline

Applying strict time constraints to TFlow restricts the time allotment for each step of the process. One way to meet this requirement is to simplify each stage. For placement, using larger, more heavily constrained blocks simplifies the placement problem. Algorithmically, this larger granularity enables significant gains relative to computational effort.

The drawback to this approach is that, when performing fully granular placement, simplification and optimization between the different components can be performed. This can lead

to resource sharing and a higher density design, which can simplify the number of required computations during placement. TFlow's modular design approach cannot take advantage of these options, but the gains in design assembly time supersede this benefit when attempting to meet the assembly deadline.

To best determine how this modular design process is helpful when attempting to meet a deadline, consider the following thought experiment. Assume that the algorithm for placement between a highly granular approach and a low granularity approach have the same computational complexity $O(m, n)$, where m is the number of modules and n is the number of placements for a module.

A module m can contain many submodules s, where $s \geq m$. Since $s \subseteq m$, the number of placements for s, p, must also be defined as $p \geq n$. This is because each placement for m is also a valid placement for the set of all submodules, s. Therefore, $O(m, n) \leq O(s, p)$. Increasing the granularity for a given design will also increase the computational complexity of that design, and reducing the granularity likewise will reduce the computational complexity.

When dealing with a hard time constraint for placement, speeding up the placement process can thus be done by decreasing the granularity. This is true regardless of the algorithm used.

Since TFlow has a strict time limit on execution, efforts must be made to reduce the time necessary for each stage of the flow. These significant gains in runtime can be obtained through the use of this strategy for reducing the computational complexity of the placement process. To increase the size of a block to be bigger than primitives, modules need to be

available. This is done by creating them ahead of time so that internal placement computations are already complete before entering the critical path. This informs the modular structure to which TFlow must adhere. In this way, TFlow's placer can complete within its time allotment.

3.3.6 Strategy to Meet a Hard Routing Deadline

Another step of the TFlow process is routing the design. This step must also complete within its time budget. Reducing the number of routes necessary simplifies the routing problem. One approach to reducing the number of routes is to use prerouted modules. Every route inside these modules is a route that will not need to be computed during implementation. The more routes that are moved into these modules, the less flexibility in routing that the final design will have. The hard deadline in routing time informs the amount of routing that should be pushed into the modules and how much can be left to occur during implementation. This informs the size and complexity of TFlow's modular structure. With these prerouted modules, TFlow's inter-module router can complete within the allotted time.

3.4 TORC

TFlow relies heavily on TORC [23], an open-source C++ infrastructure and tool-set for reconfigurable computing. The TORC infrastructure is able to read, write, and manipulate EDIF, Berkeley Logic Interchange Format (BLIF), and XDL netlists, as well as Xilinx bit-

stream frames. The TORC tools include placing and routing capabilities for full or partial designs, along with additional capabilities to facilitate design manipulation and analysis.

Many of the TORC APIs and tools are used by TFlow. The EDIF importer extracts a module's logical level information for use in creating TFlow meta-data. The XDL importer extracts physical information from the module's XDL, including anchor point, shape, and routing information. TORC also contains a device database (DDB) that can track wire and logic resource usage information for a wide range of target devices. Importantly, TORC also includes a router that can treat previously used wires as constraints to avoid contention. The bitstream parser can map from the frame indices of a bitstream file to the frame addresses on a device.

3.5 Module Relocation

Module relocation is an important component of TFlow. FPGAs consist of different types of tiles, such as Configurable Logic Blocks (CLBs), Block RAM (BRAM), and Digital Signal Processors (DSPs). These tiles are arranged in a regular pattern throughout the device. TFlow leverages this regular structure for module bitstream relocation.

A Xilinx FPGA is configured by loading a bitstream file. This file is organized into frames, the smallest addressable segment of the Virtex-5 configuration memory space [66]. A frame address maps to a tile on the FPGA, and is represented as 32 bits. Multiple frames may point to different portions of the same tile. Each frame is one clock region tall, so frame

Table 3.2: Vertical expansion of clock region size for modern Xilinx FPGA devices.

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

height and clock region height can be used interchangeably.

The size of the frame has expanded with the newer and larger FPGA architectures. Table 3.2 shows how the frame size has increased. For example, the Virtex 5 has a frame height of twenty CLBs while the Xilinx 7 Series of devices has a frame height of fifty CLBs. More importantly, while Virtex 5 has four BRAMs and eight DSPs, the Virtex 7 frame has ten BRAMs and twenty DSPs (These BRAM36s can also be used as two BRAM18s).

By manipulating the frame address, the frame data can be moved around the device. Knowledge of the contents of a frame is unnecessary for this bitstream level relocation.

Other Xilinx FPGA families, such as the Virtex 4, have frame addresses that work in a similar way. Several research teams have demonstrated methods for module bitstream relocation. [67] uses frame relocation as part of its fault tolerance tool for the Virtex II Pro. Becker [68] discusses a way to do more flexible bitstream relocation on the Virtex 4. Becker's work allows module relocation onto regions with different resources at the expense of underutilizing the region.

3.6 TFlow Phases

To best understand how the desired flow works, an in-depth analysis of each phase is necessary. These phases are *Module Creation*, *Static Creation*, and *Design Assembly*. The phases are split such that as much computational work as possible is offloaded to the Module and Static Creation phases to minimize the work done during Design Assembly. This will minimize the time necessary for Design Assembly.

3.7 Module Creation Phase

The Module Creation step is predicated on the idea that the more work that can be done prior to assembly time, the faster assembly can occur. Other modular flows have different approaches to how much precompilation should occur when creating modules. QFlow [29] uses unrouted modules that were represented as Xilinx hard macros. These hard macros would need both routing and bitstream generation at assembly time. HMFlow [28] goes further, and uses both placed and routed hard macros. However, these files are represented as Xilinx XDL, and thus need to be converted back to a Xilinx NCD prior to bitstream generation. These approaches do not offload all of the available computations from assembly. The maximum amount of processing that can occur on a module, without knowledge of the eventual use case, results in a module bitstream. As such, all possible computations are moved into this stage, leaving a minimal amount of work for assembly. This enables assembly to complete quickly.

One of the key contributions of TFlow is the library of precompiled components for later assembly. The hardware designer creates and synthesizes a module to pass to the flow. This module design has some minor constraints to make it properly fit into a modular design strategy. To reduce arbitrarily long combinatorial paths during assembly, all ports for a module must be registered. This constraint adds an additional clock cycle whenever entering or leaving a module. Timing closure for each module is thus guaranteed. During assembly, inter-module timing closure will only require routes that complete within a clock cycle. Additionally, the module should be selected such that it does not contain highly unique components, such as Input/Output Blocks (IOBs). A module with an IOB would be locked to a specific location on the device, and thus would be a much better candidate for use in the static design. Should direct IOB connectivity be desired, a better approach would be to add additional static ports that interface directly with these IOBs. This would allow design placement to have more flexibility while maintaining connectivity.

3.7.1 Module Creation

The entry point for TFlow module creation is a post-synthesis EDIF. EDIF files are an open standard [75] that can be automatically created by most front-end tools [78] [79] [80]. EDIF contains a logical level representation of a module, including connectivity, logic, ports, and other information. TFlow compiles the module through an enhanced version of the Xilinx Partition Flow [30]. Once this flow is complete, the module and its associated meta-data are added to the library.

3.7.2 Module Shaping

Before compiling a module, an appropriate shape must be selected. The shape and resource utilization of a module will decide how it can be integrated into a final design. To choose a shape, an estimation of the number and type of resources is required. TFlow uses PlanAhead for resource estimation. This yields an estimate of the number of CLBs, DSPs, and BRAMs required for the module. TFlow's custom shaping tool then creates a minimum footprint for the module that meets both resource and TFlow-specific requirements. TFlow has additional shaping rules that improve area utilization during design assembly.

When shaping a module for the Xilinx 7 Series devices, its shared clocking mechanism must be addressed. As will be discussed in the next section, TFlow relies on Xilinx Partitions to create modules. Xilinx Partitions requires every module to have an even number of columns because each clock line is shared between two columns. This further reduces the number of unique shapes that are available for a module, and reduces the placement granularity still further.

The bitstream for each module will be used during assembly. As such, no overlap between modules will be possible. Any unused resources within a module are wasted, so finding a minimal shape will reduce these wasted resources and improve module packing is important.

The resulting region is given in the form of a Xilinx User Constraints File (UCF). This file will be used as a constraint for the Xilinx Partition Flow that TFlow uses for module compilation.

3.7.3 Module Compilation

TFlow uses an enhanced version of Xilinx Partitions for compilation. Unlike the standard Xilinx tools, Xilinx Partitions will enforce routing constraints. Module routing will therefore remain inside the shaped region. This results in a reduced footprint for each module, improving module packing.

For Xilinx Partitions to work, there must be a specified hierarchy. An additional EDIF manipulation program is run, creating a seamless top level wrapper for the module. This top level wrapper has the appropriate structure required by Xilinx Partitions.

Additionally, Xilinx Partitions automatically performs port consolidation. Port consolidation takes a large set of inputs to a module and reduces them to a single port using bus macros. In the literature, it refers to these bus macros as proxy logic. This proxy logic decreases the fan-out required during the final assembly stage, allowing the assembly-time router to do less work. This reduction in routing complexity results in increased speed when routing the design, at the cost of a slightly longer combinatorial path.

Metadata Creation

In addition to the module bitstream, TFlow will also generate meta-data describing both the physical and logical properties of the module. The initial logical-level data extraction occurs from the module EDIF. Physical-level information is obtained from the module XDL. The flow for creating meta-data for a module can be seen in Figure 3.2.

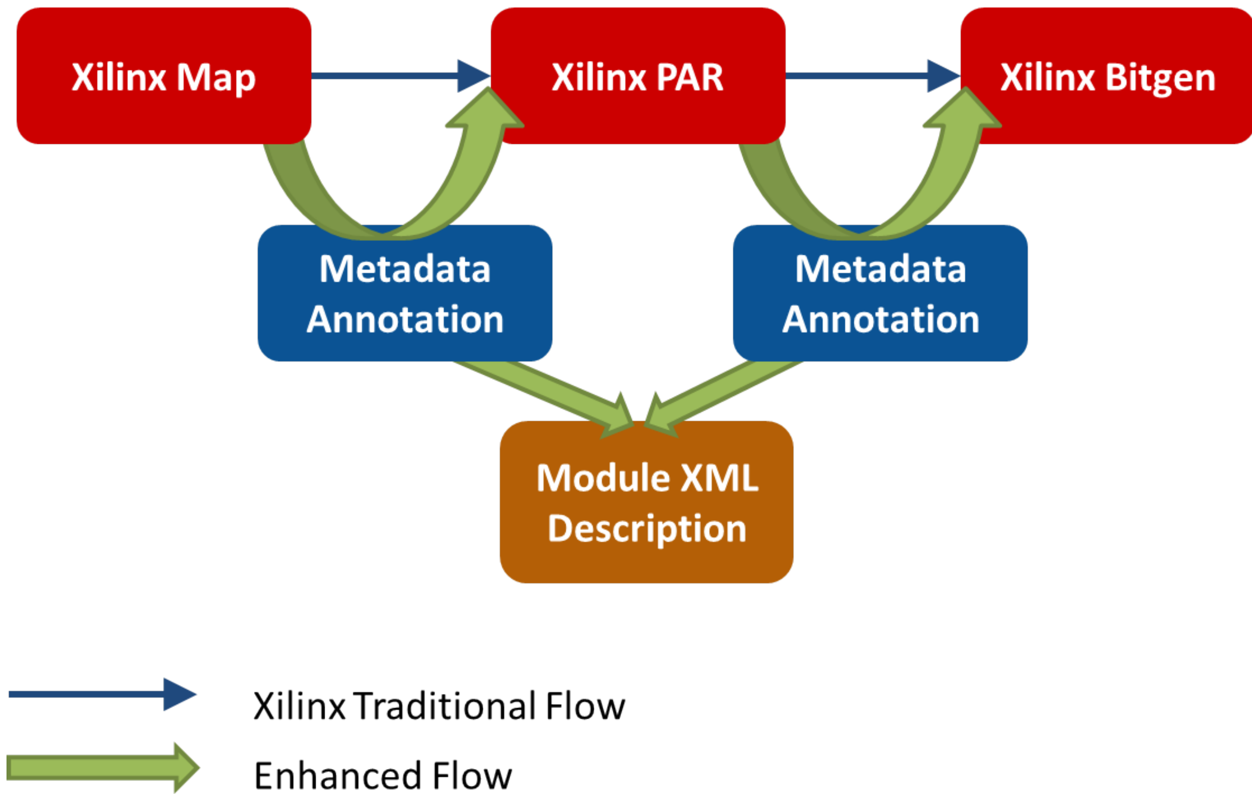

Figure 3.2: Meta-data Annotation Process

The physical-level information includes the location of each port, the used routes (denoted as PIPs), and the physical boundaries of the module. The boundaries are needed because no other logic utilization information is extracted.

Module Pre-Placement

Additional placement information is also generated and added to the module meta-data. This information consists of a list of all possible placements for the module to be located on the device. To generate this information, the southwest and northeast tiles of the module are used to generate a bounding box. This bounding box is used to generate the tile resource pattern

for the module. For example, *'CLB—DSP—CLB—CLB—BRAM'* might be the horizontal resource pattern. The tile patterns are compared against the TORC FPGA database for this device to determine where this pattern exists. Subsequently, the rows are also compared. Once all the fully matching placements are discovered, a final routing check is performed. Each of the routes in the module are tested to determine if a placement has identical routing resources. This behavior will normally occur along the edges of the device or in other irregular regions. Those locations that do not pass this test are removed from the set of viable placements. These results will be used as constraints during the placement step of design assembly. More information about module pre-placement can be found in [81].

By generating this pre-placement data during compile time, less effort is necessary during assembly time. These additional constraints reduce the granularity. This pre-placement data leverages to TORC framework to determine the device structure. In so doing, pre-placements can be generated for all TORC-supported architectures. Contrast this with the RPM grid mechanism mentioned in Section 2.3.1, where each new device family would need to have its structure be recreated by hand.

Module Clock Analysis

Another required step is to unroute the clock nets from the module. Because the Xilinx Partitions flow is run without a standard clock input, Xilinx does not distinguish between clock nets and standard nets. All nets are routed as though they were standard nets. Therefore, these 'clock' nets must be unrouted, so that the clock ports are available for clock routing

during design assembly.

TFlow Module Requirements

As mentioned, modules created for TFlow have a few additional requirements. The foremost requirement is that modules register all input and output signals. This removes inter-module timing issues that may otherwise occur. One property of modular design is that timing closure is most often an issue within the module, and does not span across modules [82]. The vendor place-and-route tools are used for this intra-module routing to ensure timing is met.

Module Library

Once module generation is complete, the module is added to the FPGA device library, where it can be used by any design for this device. Should changes to the module be required, or if it will be targeted at another device, the process can be rerun with the new information.

Different versions of the same module, distinguished by differing shapes or resource utilization, can also be constructed. However, integrating these additional versions into the flow remains in the realm of future work.

3.7.4 XML Meta-data

Since the modules are stored in the library as bitstreams, meta-data describing these modules must contain all of the necessary information for implementation. This meta-data describes the module at both the logical level and the physical level. Neither XDL nor EDIF contain a complete description of the module, but by combining information from both, a full picture can be obtained. Some of the information contained within the meta-data includes port information, clock names, pre-placement locations, and utilization boundaries. Port information consists of the physical and logical names of the ports for later translation of high-level connectivity into physical-level routes. Clock names are necessary for design- and static-level meta-data. The position and utilization boundaries are required for module placement during design assembly. By performing this data extraction prior to design assembly, design compilation time can be reduced.

An example of how physical port information is defined in XML can be seen in Figure 3.3. This information defines the exact location of the port for routing purposes. The name is necessary to translate from the logical net defined by the assembler to the physical net desired by the router. The coordinate system used is based on the tile framework of the device. This tile coordinate system is cohesive between different types of tiles. In contrast, a site coordinate system uses a different coordinate system for each type of resource. As an example using the site coordinate system, SLICE_X24Y3 is adjacent to DSP_X0Y3. As tiles, however, they are CLB_X12Y3 and DSP_X13Y3 respectively. This cohesive coordinate system allows for a single modular reference point for all resources. This permits easy

```
<mName>G</mName>
<xml_anchor>
        <mType>CLBLM</mType>
        <mX>0</mX>
        <mY>1</mY>
        <mIndex>0</mIndex>
</xml_anchor>
<mPin>A</mPin>
```

Figure 3.3: XML Port Information

relocation of the module.

To uniquely define a site, the type of the tile, the relative location, and the index are necessary. The index is necessary to specify which site on the tile is desired. For some components, such as DSPs, only one site is associated with each tile. However, for Slices, the Virtex 5 architecture has two per CLB tile. The index defines which one is requested. Lastly, the pin is needed to route the net to its destination.

A simplified version of the net representation in XML can be seen in Figure 3.4. The net name and each of the ports are described. These ports are uniquely defined as a port name and a module instance. In this case, the reset net connects the reset port of instance ZB1 with the reset port of instance BT0. Additional information about the direction of the net is inferred by the direction of each of the ports. This net and port information combine to give the router the exact physical locations of each of the ports and their desired connectivity.

```
<mName>reset</mName>
<mPins>
        <item>
                <mPort>reset</mPort>
                <mInstance>ZB1</mInstance>
        </item>
        <item>
                <mPort>reset</mPort>
                <mInstance>BT0</mInstance>
        </item>
</mPins>
```


Figure 3.4: XML Net Information

3.8 Static Creation Phase

Static creation is done under the same premise as module creation, except instead of looking for modules that can be combined, duplicated, and connected to form working designs, static design looks at those parts of a design that should not change. These include I/O ports, internal interfaces, and other static logic, such as memory controllers or Ethernet interfaces. Again, to reduce the number of computations, and thus time, required during assembly as much work as possible should occur beforehand. The logic that holds true for module creation is applicable to static creation - a bitstream consists of the maximum amount of work possible prior to assembly. Thus, the static design stage completes all of the sandbox creation, placement, routing, and bitstream generation steps. The resulting bitstream has completed as much processing as possible prior to knowledge of the requested design. This top-level static design bitstream is then added to the library.

A static design requires additional bus macros – to interface with the modules – and a module

sandbox at both logical and physical levels. The physical-level sandbox is created during the static compilation process. This physical sandbox will be completely void of any logic or routing; thus providing a clean region for module placement.

To create this clean region, the static design must be constrained such that neither its logic nor any routes cross into the module sandbox region. This is necessary because the module's routes and logic cannot be changed once the module is built, so any logic or routing already existing in the sandbox could conflict with resources reserved by the module. Constraining the logic can be done through the Xilinx User Constraints File (UCF), but constraining routing is more difficult. GoAhead [58] forces the Xilinx router to follow a set of predefined routes by marking all of the other wires as occupied. This would be an acceptable approach for TFlow, but it overconstrains the problem. OpenPR [56] has another method for constraining routing. It uses a Route Blocker, which marks wires entering or leaving a region as unavailable. Routing within or outside the region can be done as normal. This method can be used to constrain routing to either stay inside a region or to keep out. TFlow has adapted the OpenPR Route Blocker to keep the static design from routing into the module sandbox.

Compilation of the static is performed using the normal Xilinx tool-chain and results in a bitstream for use by the design assembly tools. The XDL and EDIF representations of the static design are processed and their meta-data is created. This meta-data contains information about the port interfaces to the assembly sandbox. It also contains boundary information. For the static, this boundary information describes what areas are in use, and

thus what areas will be available for placement during assembly. Routing information is also extracted, as the assembly router will need to avoid used nets when routing a final design.

The clock nets in the static design are also analyzed for eventual clock routing. Since the clocks already exist in the static, clock routing is done independently of standard routing. Additionally, clock nets use their own clock network, different from the general purpose PIPs, and will therefore have their own clock router during assembly.

Additional general information about the static design, such as port information, is exported as well. This information can be used to interface with the later design creation tools. Currently, the GReasy framework [76] imports this information for use when creating TFlow designs. GReasy will be discussed in more detail later.

3.9 Design Assembly Phase

Design assembly consists of those computations that cannot be precomputed because they require knowledge of the requested design. Each step is analyzed to determine if any part of the assembly process is redundant or could be moved earlier in the process. For example, clock analysis to determine the sources of all of the clock nets is performed during assembly in QFlow. In TFlow, this analysis process has been moved into the static creation phase, as all of the required prerequisites can be met there. By pushing as much work as possible into the previous steps, design assembly is left with little to do and thus takes little time. Meeting the time budget of TFlow requires these kinds of optimizations, and doing so is one

of the major contributions of this work.

The design assembly process can proceed quickly due to the pre-built library of components. This section describes the major steps in the iterative design creation phase.

3.9.1 Design Entry

Any front-end tool that generates an EDIF file can be used to create a design for TFlow assembly. One such tool is GReasy [76]. GReasy is a TFlow enhanced version of the GNU Radio [83] environment, an open-source environment for software-defined radios. Many radio applications could be improved by using FPGAs, but the target audience is software designers. By having a librarian with FPGA knowledge create the radio components, these software designers can use TFlow to enhance their designs with FPGAs without programming HDL. To keep the librarian/designer divide, GReasy automatically creates the design entry EDIF from its graphical user interface.

The design assembly EDIF file specifies the modules and their connectivity. The connectivity information details how the modules connect to one another and to the static design.

Figure 3.5 shows the graphical user interface for a GReasy radio design. The connectivity information is shown as multi-bit wires. The static interfaces are seen at the edges of the design. This design implements a Binary Phase-Shift Keying (BPSK) radio from library components. The data path goes from the static, through three components, and then back into the static interface. This high-level description of a design will use TFlow to

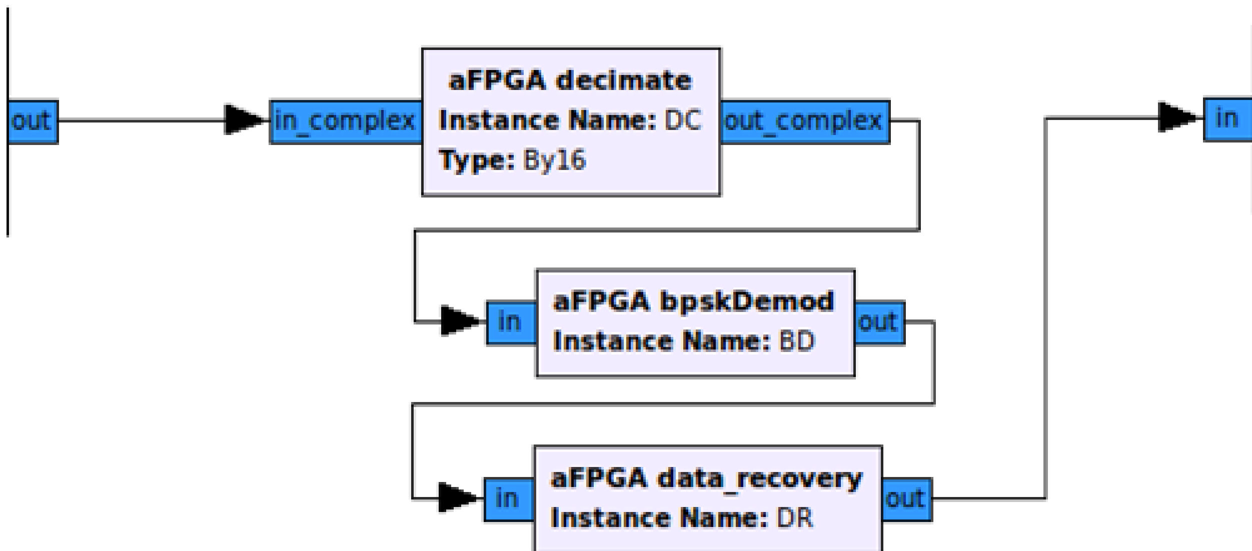

Figure 3.5: Design Connectivity Example

implement itself on an FPGA, without the user needing to know FPGA design or module implementation details.

To process this EDIF, TFlow fetches the meta-data for each of the components from TFlow's pre-built library. This information includes the ports and resource requirements. The static design information, including available space for placement, is also fetched.

3.9.2 Module Placement

Once the modules have been fetched from the library, they must be assigned a location inside the static's sandbox region. Prior work on placement was done using an extended version of TORC as a framework [43]. These extensions required additional device-specific information to be added to TORC. This information was created and added for the Virtex

5 device family. However, as it was an extension, any conversion to another device family would require the recreation of this database. Therefore, a new generic placer was designed that only required information already existing in TORC.

Many of the placement computations can be performed during the module creation phase to reduce the amount of time required during assembly. First, TORC's databases are queried, as they contain information about the device resources and the structure of the FPGA. From this, the placer finds valid module locations. Modules have placement restrictions because each tile, such as BRAMs or DSPs, must be properly matched, since the module bitstream cannot be changed; thus, a module's placement must match in both resources and routing. Additionally, modules have restricted movement within a frame, due to the requirements for bitstream relocation as discussed in Section 3.5. This forces the module to maintain the same alignment within a clock region as when it was built. Modules can only relocate vertically in steps equal to the size of the clock region on the FPGA. Horizontal relocation does not have this additional constraint. These additional constraints reduce the granularity of the placement space. Precompiling several versions of a module, each with a different shape, can provide more flexibility during module placement.

Additionally, the module height does not need to fill the full frame. Sub-frame modules can also be placed, with the same alignment constraints. Precompiling modules with the same shape but different frame offsets can counteract the resource wastage from larger clock regions at the cost of dramatically increasing the number of placements. For example, the new minimum granularity would be based on the size of a BRAM block. For a Virtex 7

device, this would increase the number of placements by a factor of ten. Table 3.2 in Section 3.5 has more information about the differing sizes of the Xilinx FPGA architectures.

Since valid placement computation [81] takes place during module creation (Section 3.7.3), module placement at assembly time can occur much faster. Thus, during module assembly, placement consists of fetching the pre-placements and bounding boxes for each module. Bounding boxes are necessary because modules may not overlap to share resources. This draw-back may increase the size of the final design. Additionally, the module placements are evaluated, and those locations which overlap with the static design are removed from the pool of possible placements. This constraint could not be created earlier, as modules only become associated with a static design during design assembly. Additional information about connectivity is also fetched, so that optimizations can be made to minimize wire length. This constraints-based placement methodology can give rapid results.

Placement Complexity

When picking a placement algorithm, both the quality of the results and the computational complexity must be analyzed. For modular design, the placer must quickly yield a result, or else it will bottleneck the rest of the flow. In turn, this means that the quality of the result can be compromised. A balance between these two factors will determine the algorithm that is best suited. Thus, an analysis of the computation complexity of different algorithms is necessary.

The placement process will involve m modules each with n placements. The number of placements may vary between modules, depending on the pre-placement results.

A straight-forward approach for solving the placement problem is to use a brute force algorithm. Brute force has a computational complexity of $T(m,n) \in O(m^n)$. This will scale poorly with respect to either m or n.

Another placement algorithm simply attempts to find the first valid placement for the full design using a depth-first search. This approach has a best case performance of $T(m,n) \in O(m)$ when the first choice is a valid one, but a worst case performance equivalent to brute force. Thus, as the placement problem scales, there is no guarantee that the placer will return a result within a reasonable time frame.

[41] uses an a variant of this simplified placement algorithm. It places one module at a time in the first valid location it finds. When adding additional modules, prior modules can block placement; there is no multi-module optimization. Thus, it is possible for a valid solution to exist, but this algorithm may not discover it. On the other hand, the result - either a placement or failure to place - will occur rapidly. In either event, the quality of the result is not used to guide placement in any way.

An approach that sequentially places each module without backtracking has complexity of $T(m,n) \in O(m * n)$, but for best performance requires a sorted list of modules; sorting requires $T(m) \in O(m \log m)$. The modules are sorted by ascending number of placements. Thus, those modules with the most restrictive footprint will have their placements determined

first, based on placement quality. Despite not allowing for backtracking, this approach still yields a much better complexity than the other options. Although global optimization is still not performed, choosing an appropriately sorted module list can yield near-optimal results with minimal time requirements. This is a better approach than [41] both because the best placement for a module is selected based on the placement quality, and because the order the modules are placed can be optimized for best results.

To determine the quality of the results, the modules' connectivity information is fetched from the design. This information tells how each of the modules connect to one another and to static modules outside of the placer. This connectivity is weighted by the number of connections between each module. The final calculation uses Manhattan distance, such that a placement has a value equal to that in Equation 3.27.

$$\sum_{i=1}^{m} \sum_{j=i+1}^{m} w_{i,j} \Big(|m_{i_x} - m_{j_x}| + |m_{i_y} - m_{j_y}| \Big) \tag{3.27}$$

In this equation, m is the number of modules, $w_{i,j}$ is the weight of the connectivity between module i and module j, and m_{i_x} and m_{i_y} are the x and y coordinates, respectively, of the centroid of module i. This will preferentially choose module positions aligned in the vertical or horizontal plane; this can simplify the routing step. This calculation is run on each placement for a module to determine its best valid placement. Once selected, the next module is placed, until all the modules have been placed or placement fails. Either result will occur quickly, as desired to meet the deadline.

Finalizing Placement

The completed module placement is then added to the meta-data, which is then used during final assembly to (a) relocate the component in the bitstream, and (b) identify the exact position of the module's terminals for subsequent routing.

This placement step can thus be completed in seconds due to the large granularity of the placement problem and the considerable precomputation performed during module creation.

3.9.3 Inter-module Routing

Once the modules are placed, the next step is to route the desired inter-module connectivity. TORC's routing capability [23] was expanded into a router for TFlow [84]. The terminals of the precompiled modules and their inter-module connectivity form a routing task list. The pre-existing routes inside the static and each module are also imported into the routing task list as constraints. These routes cannot be modified, because they already exist in the bitstream. With this information, the custom router can route through the static and the modules without impacting existing connectivity.

TFlow's custom router then generates a list of the Programmable Interconnect Points (PIPs) necessary to route the design. This custom router is designed primarily for execution speed, routability, and lastly, timing performance. As mentioned in Section 3.7, the I/O signals of the modules are registered, so the inter-module timing constraints are lessened. The PIP listing is then passed to the next phase of TFlow for transformation into bits.

3.9.4 Clock Routing

The clock information is extracted from the static meta-data and incorporated into the design. Due to the different routing resources needed for clock routing, the TORC router is insufficient. A separate clock router extracts the desired module clocks from the meta-data and routes them using the FPGA clock tree. The required PIPs are passed on to the next stage for their micro-bitstreams to be added to the design.

3.9.5 Bitstream Stitching

The final step creates a bitstream that implements this design. The static bitstream is fetched from the library as a starting point. As mentioned in Section 3.8, this static bitstream has clean regions that have no logic or routing. These are the sandbox regions where the modules are to be placed.

The meta-data specifies the module bitstream frames for relocation into the static bitstream. The new location for these frames is given by the placement meta-data. This overwrites the region, which is why the sandbox region must be empty; otherwise, existing logic will be erased. The contents of the frames are not changed for relocation.

Lastly, the connectivity PIPs for routing are translated into micro-bitstreams. Writing these bits readies the bitstream for transit onto the physical device. See [69] and [85] for more details about bitstream generation. When dealing with assembly at the bitstream level, no additional processing, such as XDL-to-NCD conversion or bit generation, is necessary, in

contrast to flows like HMFlow [28] or QFlow [29].

3.10 Debugging

One of the drawbacks to skipping directly to a bitstream is that debugging becomes difficult. Of course, physical prototyping allows for the design itself to be analyzed by testing the actual inputs and outputs to a design. This yields the true behavior of the system and can give results immediately.

Were a problem to arise, there are a few possible approaches. As mentioned, the physical response of the design may be sufficient to determine the behavior. However, if further analysis is required, a tool such as ChipScope [86] can be used. ChipScope can be implemented as part of the static design, and either would probe the normal inputs and outputs of the sandbox, or it could have its own set of ports. These ChipScope ports would be treated like any other port by TFlow, and thus any signal internal to the sandbox could be extracted.

Another approach to debugging is to use the standard Xilinx tools to analyze the design. These tools require an NCD, and the output of TFlow is a bitstream. To overcome this issue, TFlow includes a debugging toolflow extension. Just before bitstream stitching, but after placement and routing, TFlow can build an XDL representation of the design. This effectively turns TFlow into a variant of HMFlow; as with HMFlow, this XDL then completes the time consuming `xdl2ncd` process. Because of the additional time requirements for running the debugger, it is only run when specifically requested to keep the normal TFlow

bitstream generation time fast. Once the NCD is created, the flow can be analyzed however the designer prefers. Thus, using TFlow does not lock a designer from performing design analysis.

3.11 Summary

This flow uses multiple stages as a cohesive whole to create a rapid modular assembly design process. Modules are automatically shaped, built, and added to the library, with significant meta-data, including a list of valid possible placements.

The static design has both logical constraints through the Xilinx tools and routing constraints through the custom route blocker to create a clean sandbox for design assembly.

With this library creation phase complete, design assembly can begin. The desired modules are fetched from the design and their meta-data is used as an input to the fast module placer. Combined with the sandbox information stored in the static meta-data, this placer rapidly generates high-quality placements for the design.

With the new placements known, connectivity between the modules can be implemented by the TFlow router. Using the included meta-data as a guide to existing routes, this router generates new valid paths to connect all of the components. Since this routing is built during assembly, changing the connectivity can be done almost instantly should a new design request it.

At this point, a full picture of what the desired design should look like has been built. The last step is to implement it. Module bitstreams are fetched and relocated to the desired location, and routing micro-bitstreams are added to the design.

This full assembly process takes only seconds to complete, and can immediately be deployed onto the FPGA for use. As can be seen, each component of TFlow is necessary to build a cohesive flow that can meet the hard deadline imposed by the human attention span.

Chapter 4

Results

To determine if TFlow manages to succeed in its goal of deadline design assembly, it is necessary to ascertain the efficacy of each component of the flow as well as the overall behavior.

4.1 Flow Optimization

While precompiling modules should already enable a significant compilation time improvement over other flows, the design assembly process should also be optimized. As such, flow analysis was performed to determine how the design assembly process performed and where it could be improved. The router, the placer, and the metadata were all found to have inefficiencies, and the following sections will discuss what these problems were and how they were overcome.

Table 4.1: Router Run-time Improvements

	Iteration				
	1	2	3	4	5
Time	1:19.541	55.963	53.647	39.050	38.602

4.1.1 Router Optimization

Router run-time analysis revealed that there was significant room for improvement. This router [87] implements a pathfinder algorithm using A*, and extends the capabilities of the TORC router. For a more in-depth discussion of the router, refer to [84].

The router performance was analyzed through tools such as *kcachegrind* and it was found that when attempting rip-up and retry, the information gained from the prior attempts were deleted and the router started over. This issue only began to reveal itself with more complicated designs, since if the first routing attempt was successful, it did not occur. These tests were run on a 2.83 GHz Intel Core 2 Quad with 3 GB of DRAM.

The same test bench design was run through the router, and the impact of each iteration of code improvements can be seen in Table 4.1. These optimizations halved the necessary run-time for this test case.

4.1.2 I/O Optimization

Next, I/O performance for accessing the module and static meta-data was analyzed and long time requirements were discovered. I/O performance scaled poorly with the size of the XML meta-data. The initial test cases used small designs where this I/O time was overshadowed

by the rest of the assembly time. However, when run on larger designs or less capable CPUs, benchmarking the flow revealed the issue.

To solve this problem, the serialization process, which uses the standard C++ boost library, needed to be studied.

Accessing the module meta-data involved reading the XML files and importing them into the TFlow data structures through the standard C++ boost serialization function. It was found that although the human-readable XML files were quite useful for debugging, the C++ boost libraries also supported a binary format. The advantages of the XML format are that it is human-readable and platform-independent. As such, transferring modules from one architecture to another required no additional overhead. Examples of different architectures are 32-bit vs 64-bit x86 processors, as well as the ARM architecture. The binary format is not human-readable, and is no longer platform-independent. A binary metadata file is not guaranteed to be transferable to another machine. When looking at the size of these files, the binary file was found to be almost one-third smaller. For example, one large design was 72 MB in XML and only 25 MB as binary. The run-time gains from using a binary format are significant, as will be shown later.

To gain the advantages of both formats of metadata, a *metadata_converter* was built to switch between the different formats. This is necessary if a module library is to be platform-independent. When this library is on the assembly machine, the metadata will be converted into the binary format for fast assembly.

These metadata improvements are demonstrated in Section 4.2.1 and Section 4.3.6.

4.1.3 Placer Optimizations

The placer was also analyzed to determine its performance with respect to different placement algorithms. The sequential placement algorithm consists of stepping through a sorted list of the modules and searching for the best location for that module before proceeding to the next module. There is no backwards traversal allowed. The first valid algorithm uses a depth-first search to find a valid placement. It stops once it discovers a valid design placement. The brute force approach uses the same depth-first search, but it searches the entire placement space for the best result. The random approach creates N placements for the design, which are then evaluated to determine if they are valid; the best one, if any, is selected. This gives a better view of the expected quality of the results than the first valid algorithm, because that approach is heavily reliant on the order the search tree is traversed.

Each test case uses modules that have already had the pre-placement process completed and are awaiting final placement. This may involve multiple instances of the same module, or a mix of modules. The design also includes connectivity information describing how the modules are to connect to one another and to any existing parts of the design.

The timing information has a resolution of 0.01 seconds. Tests were run on a 2.83 GHz Intel Core 2 Quad with 3 GB of DRAM. Tests were allowed to run for a maximum of 200 hours before being marked as incomplete. The modules were placed on the Xilinx Zynq 7 Series,

Table 4.2: Placement Results, 15 Streaming Blocks

Streaming Design	15 modules	62 placements
Algorithm	Time (s)	Quality
Sequential Placement	0.01	4228
Random 10000	0.06	7556
Random 100000	0.52	7150
Random 1000000	5.16	6814
First Valid	<0.01	9072
Brute Force	>200 hours	N/A

XC7Z045 architecture. Placement on other architectures has been performed, but this only influences the number of placements available for each module and thus is not included. The placements were graded on both time and quality of results. The quality of the results is the Manhattan distance for all the connections, but the actual value should only be used to judge placement efficacy within a test case. In the designs where brute force results are available, they are used as an optimal reference for determining the quality of the results.

Table 4.2 describes a placement design which consists of $m = 15$ modules each with $n = 62$ valid placements. These modules are streaming, in that each one connects only to the next one in the sequence. This streaming design is an appropriate use case for many real world applications, such as radio designs [76]. In this case, the brute force approach did not run to completion. As seen, the sequential placement algorithm delivered the highest quality results while completing within a hundredth of a second.

Table 4.3 involves a placement design that consists of 100 modules, each with 62 valid placements. These modules represent a 10 x 10 grid with connectivity from each module to its eight neighbors. With this large number of modules and placements, brute force again

Table 4.3: Placement Results, 10x10 Grid

Compact Grid Design	100 modules	62 placements
Algorithm	Time (s)	Quality
Sequential Placement	0.4	14846
Random 10000	0.15	111016
Random 100000	1.48	105360
Random 1000000	14.8	102704
First Valid	<0.01	48202
Brute Force	>200 hours	N/A

Table 4.4: Placement Results, 7x7x3 3D Grid

Compact 3D Design	147 modules	62 placements
Algorithm	Time (s)	Quality
Sequential Placement	1.56	74819
Random 1000000	-	Failed
First Valid	<0.01	299473
Brute Force	>200 hours	N/A
Hand Optimized	N/A	102089

could not run to completion. The sequential placement algorithm yielded results more than three times better then its nearest competitor.

Table 4.4 involves a design that consists of a 7x7x3 array, where each neighbor is connected to one another. For this case, the random approach did not yield any valid placement results. An additional hand optimized manual placement was performed, but this still did not yield better results than the sequential placement algorithm. As seen in the prior examples, the brute force design could not complete with the large placement space, and finding the first valid placement, while fast, yields suboptimal results.

Figure 4.1 is a video representation of the 7x7x3 array, using the sequential placement algorithm. Note that each module goes through the full set of placements before choosing the current best option based on the quality of the current results.

Table 4.5: Placement Results, ZigBee Radio Design

Streaming Radio Design	5 modules	32 placements
Algorithm	Time (s)	Quality
Sequential Placement	<0.01	19415
Random 10000	0.2	19699
Random 100000	1.97	19481
First Valid	<0.01	24987
Brute Force	2.39	19407

The final set of results can be seen in Table 4.5. This design consists of five modules, three of which have 32 placements and the remainder with only two placements. This yields a total number of placements of 131,072, which is small enough that brute force can finish in a reasonable amount of time. As expected, the brute force approach has the best quality of results; however, the sequential placement algorithm has an almost identical quality, takes less time and, as seen previously, scales much better.

A number of possible placement algorithms were explored. The initial algorithm was a simple brute force design. Unfortunately, while the quality of the results is optimal, this method scales poorly and may not complete. To overcome this problem, the first valid placement was selected. This ran quickly when a valid solution was found, but would yield poor quality results. In addition, when no valid solution exists, this technique would run for the same length of time as brute force and so might not complete. The next set of algorithms involved a random placement strategy. Some number of random placements were generated, and these placements were evaluated for validity and quality. The time requirements for this algorithm were static, in that the time increased with the number of random placements. The quality of the results could be good, run time could be kept to a manageable level, and best of all,

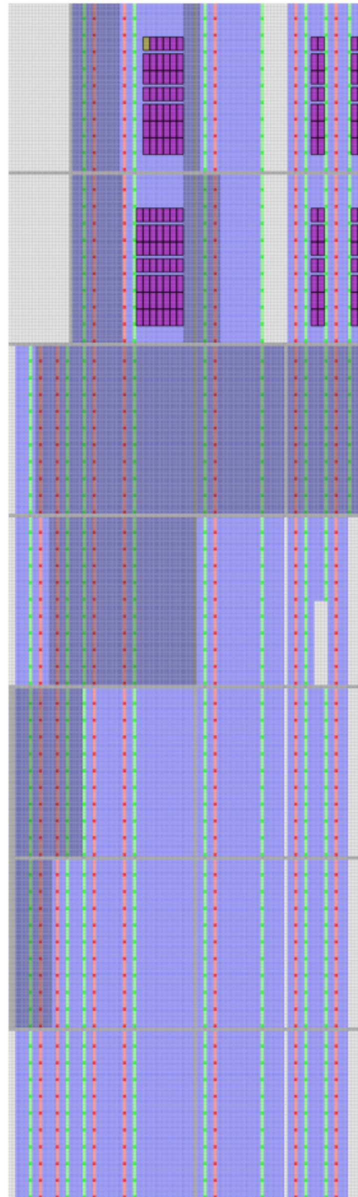

Figure 4.1: Sequential Placement (In Windows Adobe Reader, click for video)

it was guaranteed to complete one way or the other. However, with much more complex designs, it was found that the random placer would spend most of its time generating invalid placements. Thus, the sequential placement algorithm was explored. This algorithm also is

guaranteed to complete quickly, and the quality of its results are comparable with that of brute force.

Thus, in situations where m or n are sufficiently small, the brute force algorithm runs for an acceptable amount of time and the resulting placement is guaranteed to be optimal. Brute force and the first valid approach are also guaranteed to find a valid placement if one exists. This guarantee comes at the price of time, quality of results, or both. Thus, once m or n begin to grow, faster algorithms must be used. For TFlow, the sequential placer algorithm is used due to its high quality and rapid completion time.

4.2 TFlow on Virtex 5

Performing module creation ahead of time is one of the main ways that TFlow can complete so rapidly at assembly time. To get some perspective on how much work this step will save, and how large the M_T term is in Equation 3.11, analysis of module creation is necessary. The following tests deal with how long it takes to create each module and some of the properties of these modules.

For this set of tests, modules were compiled on a 4th generation Intel i5 processor with 32 GB of ram. This set of modules was compiled for the Virtex 5 XC5VLX110T device. Table 4.6 has the resulting maximum clock frequency and build time requirements.

The run-time for compiling a module is a good indication of how much computation is occurring during this preprocessing step. This will tell the complexity of module preprocessing,

Table 4.6: Virtex 5 Module Run-time and Timing Analysis

Module	**Frequency (MHz)**	**Run-time (s)**
add	503	105.5
bpskDemod	54.2	172.6
complex_conjugate	517.87	101.2
data_recovery	371.89	107.5
decimateBy16	344	136.6
decimateBy64	358.29	128.2
passthrough	588.58	101.5
rms	117	130.0

M_T, as mentioned in Equation 3.11 and determine the amount of work that has been pre-computed. The larger M_T, the more work that has been done ahead of time and the less that will need to occur at assembly time. In addition, by splitting the design process into small, independent pieces, the process can easily occur in parallel. This set of modules could be created and added to the library independently, taking only 172.6 seconds to generate them all.

One drawback of this independent module creation approach is that the modules are built without knowing the eventual timing constraints of the final design. As such, each module has a different supported maximum clock rate. Should a module not meet timing during assembly, the module specification can be updated and a new version of the module can be built. Modules are built such that they take up a minimal footprint. If this new timing cannot be met, this area constraint can be relaxed to create a larger module that achieves the desired clock rate.

Another drawback of precompiling modules is that area reserved by a module is reserved for only that module. No other part of the design can use those resources. This restriction can

Table 4.7: Virtex 5 Module Resource and Placement Analysis

Module	**Requested Resources**			**Allocated Resources**			**Placements**
	CLBs	BRAMs	DSPs	CLBs	BRAMs	DSPs	
Add	26	0	0	40	0	0	156
BPSK Demod	272	3	7	280	4	8	6
Complex Conjugate	18	0	0	20	0	0	168
Data Recovery	24	0	0	40	0	0	156
Decimate By 16	186	0	4	200	0	8	6
Decimate By 64	187	0	4	200	0	8	6
Passthrough	18	0	0	20	0	0	168
Root Mean Square	273	0	0	280	0	0	20

be exceptionally onerous when dealing with the sparser BRAM or DSP resources. Table 4.7 shows the resource overhead for modules that fill the Virtex 5 clock region. The important resource overhead occurs when requesting excess BRAMs or DSPs. This will be contrasted with the Xilinx 7 Series devices in a later section. The 7 Series has a much larger clock region that is fifty CLBs high instead of the twenty in the Virtex 5 architecture.

The area reservation of a module also impacts the number of possible different placements for that module. From these examples, the number of placements for these modules span from as few as six to over one hundred and sixty. This relatively small number of available placements for a module will allow the placement process to occur much more rapidly than standard high-granularity flows.

Static designs are also included in TFlow's component library. The static contains I/O, static logic, and interfaces into the sandbox. Static logic can grow large when including blocks like Ethernet or memory controllers. However, the amount of static logic must be balanced against the need for flexibility during design assembly. If all of the logic is moved

Table 4.8: Virtex 5 Static Timing Analysis

Clock	Frequency (MHz)
sys_clk	134.9

into the static, design assembly will consist of simply loading the static design, as no modular assembly will be necessary. However, this comes at the cost of being unable to rapidly change the design. Therefore, selection of the static to determine the maximal static design that does not compromise flexibility is an important task when building the TFlow library.

The following static design is used for GReasy radio designs [76]. As this static is fairly complex, it takes 318 seconds to compile. This consists of work that has been offloaded from design assembly, S_T, as represented in Equation 3.10. Table 4.8 has the timing information for the clock(s) that feed into the blacktop. This design only connects a single clock into the module sandbox. This is the fastest supported clock rate for this static design. The clock in this static runs at 60 MHz, well within this range.

4.2.1 Design Assembly on the Virtex 5 Architecture

TFlow's assembly run-time must be compared against other techniques and against the attention span window. The comparison flows are Xilinx ISE and QFlow [29]. Xilinx ISE does not perform modular design, while QFlow only processes modules prior to routing. The most important metric for TFlow success is the speed of the assembly process, as meeting the time requirement is TFlow's primary contribution.

The following test results, shown in Table 4.9, uses the GReasy BPSK radio design given

in Figure 3.5, which consists of a decimate-by-16 block, a BPSK demodulator block, and a data recovery block. This test case was run using three different flows, the standard ISE flow, the hard-macro based QFlow, and the bitstream assembly flow TFlow. It was run on an Intel i7-2600 with 8 GB of DRAM. Each step of the process has been split apart so that the stages can be compared. The first step, map, has results that reflect the granularity difference between the tool flows. Where ISE has no reduction in granularity, QFlow places pre-mapped modules into the sandbox based on resource constraints. TFlow's modules are both pre-mapped and pre-routed, so they have a reduced number of possible valid locations for placement. TFlow also requires module alignment with the clock region, which reduces the granularity still further, speeding up placement. For the routing stage, both ISE and QFlow use the Xilinx Place and Route tool, yielding similar results. TFlow only routes the inter-module connectivity during assembly due to its pre-routed modules. Lastly, bitstream generation for both ISE and QFlow use the Xilinx Bitgen tool, whereas TFlow has integrated bitstream generation into its routing stage. While QFlow has a speed advantage over the Xilinx flow, TFlow has the clear speed advantage at every step and completes within the allotted time.

TFlow Model Validation

These results can be compared against the flow model in Section 3.3. The terms D_I, D_Q, and D_T can be found from Equations 3.5, 3.16, and 3.12 respectively. From this, it can be seen that the modular and static components of these equations are significant; for ISE, $D_I - D_T$

Table 4.9: Assembly Time for BPSK Radio (s)

	Map	Route	Bit gen	**Total**
ISE	109	47	34	190
QFlow	43	49	42	134
TFlow	11	6	-	17

from Equation 3.13 clocks in at approximately 173 seconds. The R_M term for QFlow is also large at 43 seconds, while $D_Q - D_T$ from Equation 3.17 evaluates to 85 seconds. In addition, these results reveal the simplification of the model. QFlow and TFlow have different times required to complete the Map (Place) step. For the model, these two are treated identically because both of them have precompiled modules, but in practice the time requirements can be different. Still treating them as identical yields Equation 3.17, which gives a reasonable model for determining how the complexity of the flows differ at design time. The current results show that this underestimates TFlow's speed advantage. Still, the model accurately reflects the gains possible from precompilation and will enable deadline assembly.

4.2.2 Further Virtex 5 Design Assembly Exploration

To expand on the comparison between ISE, QFlow, and TFlow, the next set of tests will further investigate TFlow's implementation of the modular flow model to achieve deadline assembly. The model predicts that the more computations that occur prior to assembly, the faster the flow. TFlow should be the fastest as it precomputes bitstreams, followed by QFlow's placed but unrouted hard macros, and last should be ISE, while computes everything at assembly time. Only TFlow should be able to complete within the desired time window.

The following test cases compare the Xilinx ISE compilation time with that of both QFlow and TFlow. These test cases were run on a 2.83 GHz Intel Core 2 Quad with 3 GB of DRAM. For consistency, both QFlow and TFlow use the same XDL RPM grid placer from [43]. TFlow uses the binary metadata format; a run-time comparison with the XML metadata will follow.

The first three test cases in the next example target the Xilinx Virtex-5 XC5VLX110T FPGA board. The first design is for a video edge filter. As can be seen in Table 4.10, TFlow runs approximately twelve times faster than the other flows. The second design swaps out the edge filter for a video Gaussian filter. Assembly of the Gaussian filter also has a 15x speedup over the other methods. The time required to run TFlow is the total time from having an edge filter design to having a Gaussian filter design, because both of these designs share the same static. The third design has a different static and uses modules for a ZigBee Radio. This static is more complex, resulting in a more pronounced difference between QFlow and ISE. However, TFlow maintains its lead in all three cases and meets the assembly deadline. This lead is due to the significant pre-processing of TFlow's components, reducing the computation necessary for assembly.

The fourth test case was run targeting a Xilinx Virtex-5 XC5VLX330 board. This board is considerably larger and the static design is for the more complicated Convey environment [88]. A vector-add module was used for this test. With this more complex design, the differences between the flows are emphasized. While QFlow has some significant gains over ISE, TFlow completes assembly in forty-two seconds. This exceeds the attention span limit,

Table 4.10: Virtex 5 Design Assembly Comparison

	Time (s)			TFlow Speedup	
	ISE	QFlow	TFlow	Over ISE	Over QFlow
Edge Filter	184.6	170.8	14.8	12.5x	11.5x
Gaussian Filter	159.8	156.7	10.2	15.7x	15.3x
ZigBee Radio	236.2	157.7	15.2	15.5x	10.4x
Vector Add	3891.7	805.1	42.2	92.2x	19.1x

so a thorough analysis was done to determine what improvements could be made. Analysis showed that more computations could be moved into the static creation process at no penalty. Doing so reduces the run time to twenty-five seconds, meeting the deadline. This optimization is incorporated into the 7 Series test cases. Even without this change, TFlow performs over ninety times faster than ISE, as shown in 4.10. The static and the routing for this design are complicated, but TFlow's use of precompiled modules allows for quick and flexible modular design.

The next set of results show the improvement possible due to the I/O optimization mentioned previously. As the functionality does not change, any speed improvement will validate the new metadata representation.

Table 4.11 shows the performance difference obtained for the two different metadata formats, XML and binary. The binary files are approximately three times smaller than the XML files, and the parsing is faster. From this data, it can be seen that using binary metadata runs approximately twice as fast as XML. The gain is proportional to the size of the module and static metadata. The edge and Gaussian filters both use the same static, so their speedup is about the same. On the other end of the spectrum, the Vector Add example uses the large

Table 4.11: Virtex 5 TFlow I/O Comparison

	TFlow (s)		TFlow Speedup
	XML	BIN	
Edge Filter	26.2	14.8	1.8x
Gaussian Filter	19.7	10.2	1.9x
ZigBee Radio	19.8	15.2	1.3x
Vector Add	98.7	42.2	2.3x

Convey static design, so a significant gain can be obtained.

HMFlow Comparison

To compare these results against HMFlow [28], one approach is to look at its best-case end-to-end solution. As mentioned, HMFlow performs hard macro placement and routing at the XDL level. HMFlow can generate results quickly, but since FPGAs require a bitstream for use, the conversion process must be taken into account. Assuming HMFlow's placement and routing require zero time, the XDL design must still be converted to a bitstream. Should this best-case scenario still not meet the deadline, HMFlow can be removed from contention as a solution to sub-attention span assembly.

Conversion of the XDL to a bitstream is performed through the use of the Xilinx `xdl2ncd` tool, which converts the XDL file to the Xilinx NCD format. This is the required input to the next Xilinx tool, Bitgen. Bitgen takes this NCD and compiles a useable bitstream file. HMFlow's additional requirements over TFlow were shown in Equation 3.21, and consists of the HMFlow overhead, O_H, and the bitstream generation time for the static design and the modules, $B_S + B_M$. Evaluating $D_H - D_T$ for its time requirement can indicate how

Table 4.12: Virtex 5 HMFlow Overhead (seconds)

Module	XDL to NCD	Bit gen	Total Overhead	TFlow Reference
Edge Filter	105.6	41.7	147.3	14.8
Gaussian Filter	101.2	38.2	139.4	10.2
ZigBee Radio	17.8	45.8	63.6	15.2
Vector Add	86.6	213.2	299.8	42.2

HMFlow's approach performs. This set of tests was run on a 2.83 GHz Intel Core 2 Quad with 3 GB of DRAM.

According to the simplified model in Equation 3.21, the difference between HMFlow and TFlow is represented by the overhead incurred from running `xdl2ncd` plus the bitstream generation time. This time can be determined by taking the XDL versions of completed TFlow designs and running them through these processes. This controls for any difference in the prior steps for TFlow or HMFlow. Were this difference to be small, it would mean that pushing bitstream generation into the library creation phase may be unwarranted.

Fortunately, this is not the case; Table 4.12 shows that regardless of the speed of HMFlow, just the overhead to completing assembly requires more than four times longer to finish than the time it takes to complete the full TFlow design and can be thirteen times longer. This overhead exceeds the desired completion window. In practice, HMFlow's placement and routing will also require non-zero time to complete as well, extending TFlow's lead still further.

Bitstream generation time is highly dependent on the size of the target FPGA. Since the first three designs are each on the Virtex-5 XC5VLX110T, Bitgen takes approximately the

same amount of time. The fourth design is on the much larger XC5VLX330, and takes a proportionally longer time to complete. As device size continues to grow with newer devices, HMFlow's overhead should remain a bottleneck during assembly.

Thus, HMFlow only solves part of the problem of putting the design on a device within the allotted time. While their XDL generation time may be excellent, the bottleneck from converting XDL into a useable bitstream means that this approach is unable to complete within the attention span deadline.

4.2.3 Attention Span Comparison

Figure 4.2 shows the attention span time constraints relative to the best completion time for each flow. Note that the time scale is logarithmic. Only TFlow completes within the critical time limit. HMFlow, QFlow, and ISE each require additional time to assemble a design. Any excess time required will cause the user's attention to be lost and productivity to drop. TFlow is able to complete within the deadline using precomputed modules and rapid assembly. Since the other flows are unable to do so, TFlow's modular approach is validated.

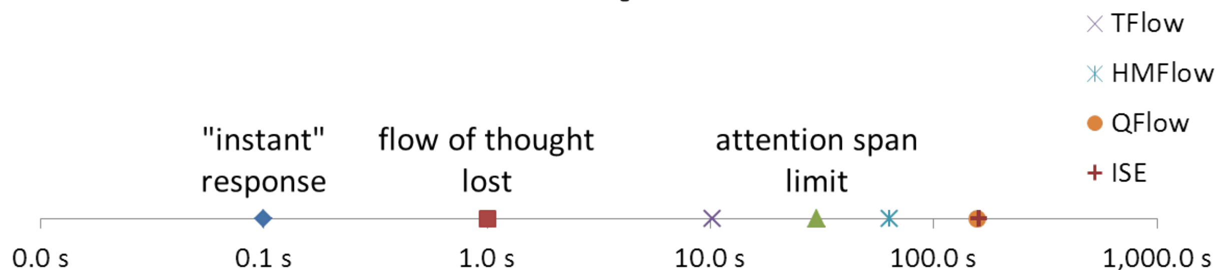

Figure 4.2: Flow Assembly Time Relative to Attention Span

4.3 TFlow on the Xilinx 7 Series

The previous examples used the older Xilinx Virtex 5 architecture. A more recent Xilinx generation is the Xilinx 7 Series, which includes the Virtex-7, Kintex-7, and Artix-7 devices as the high-end, mid-range, and low-end respectively. It also includes the Zynq-7000 devices, which have an embedded ARM processor.

This newer architecture is larger and more complicated than the Virtex 5. An analysis of each of the components of the flow is necessary to ascertain if these complications will help or hinder TFlow's modular approach.

When updating the code base to the Xilinx 7 Series, some significant architectural differences were found. These devices have a paired tile structure, which impacts module shaping, tile names, and a different clocking structure. Additionally, new micro-bitstreams were necessary, as well as incorporating an updated version of TORC [23]. Each of these changes were successful, and the following examples demonstrate the performance of TFlow for the 7 Series devices.

Again, performing prior module creation is a key component to the performance gains of TFlow. The time necessary to build these modules is a good indicator of how much work would otherwise be necessary in a standard flow. Additionally, the overhead and properties of these modules will inform the types and speed of the final design.

The following examples include designs built for the Zynq-7000, as the on-chip ARM CPU enables additional application opportunities. TFlow is fully compatible with the full 7 Series portfolio. The device used in these examples is the Zynq XC7Z045 and the modules were built on an Intel i7-2600 with 8 GB of DRAM.

Table 4.13 shows the module creation times and the maximum clock frequency. Generating these Zynq modules takes longer than those for the Virtex 5; this will be a consistent theme when comparing the different generations. As with Virtex 5, this is a fully parallel process; multiple modules can be built independently. With this longer compile time for the modules, TFlow should reduce assembly times drastically, as these computations will already be complete at assembly time.

These modules are also built without regard for the eventual design. As such, no target clock rate is specified. Nevertheless, most of these modules support reasonable clock rates. Should a faster rate be desired, the tool can add in clocking constraints or relax the footprint so that module creation can meet these more stringent restrictions. When generating a module in this manner, the flow will inform the user if these constraints cannot be met.

These modules are designed to take up the minimum amount of resources necessary. Table

Table 4.13: Zynq Module Run-time and Timing Analysis

Module	Frequency (MHz)	Run-time (s)
Decimate By 16	428.082	480.3
BPSK Demod	57.2	490.5
Data Recovery	433.7	502.1
Frequency Tuner	358.9	461.7
DES Encryption	288.4	523.2
Zigbee Receiver	149.4	455.9

4.14 shows the resource overhead for generating a design using TFlow. The clock region is larger for the Zynq device family, and the clock architecture uses a shared clock between pairs of columns. Therefore, the number of columns in a module will always be an even number. This overhead is part of the cost of precomputing independent modules. Modules cannot overlap, because their bitstream representation cannot be modified at assembly time.

As anticipated, the overhead when dealing with this larger block size on the Zynq as compared to the prior Virtex 5 example in Table 4.7 is considerably worse. The larger minimum module size for newer devices is a serious detriment to efficient resource utilization, and needs to be addressed.

This overhead can be ameliorated by building sub-frame blocks. Sub-frame blocks no longer need to fill the full clock region, and thus the sparser BRAM and DSP resources can be better utilized. While this process is supported by the tools, this example demonstrates the overhead required by using modules of clock region height. Additionally, sub-frame placement increases the number of Zynq placements by a factor of ten, increasing the complexity of the placement problem. Thus, the use of sub-frame modules should involve an analysis of the resource overhead versus placement complexity.

Table 4.14: Zynq Module Resource and Placement Analysis

Module	Requested Resources			Allocated Resources			Placements
	CLBs	BRAMs	DSPs	CLBs	BRAMs	DSPs	
BPSK Demod	302	3	7	350	10	20	7
Data Recovery	24	0	0	50	0	0	55
Decimate By 16	168	0	4	200	0	20	7
DES Encryption	495	1	0	500	10	0	7
Frequency Tuner	50	2	3	100	10	20	12
Zigbee Receiver	281	0	10	300	0	20	7

The number of placements a module can have will influence both the flexibility of the placer as well as the complexity. This is a careful balancing act. Too many placements will slow down module placement, while too few may preclude placement altogether. Looking at these modules gives some perspective on the expected number of placements for a range of modules. Placements for these designs vary, with some modules limited to only a few options, while others can migrate throughout the board.

4.3.1 Placement Overhead

Tables 4.7 and 4.14 show the area overhead incurred from building modules instead of using a single build stage. To delve deeper into this issue, Table 4.15 shows the area required for a design that consists of three modules. The Full BPSK Design represents the area required if these three modules are built independently and assembled at design time. From this, it can be seen that there is a considerable number of reserved but unused resources.

The Full BPSK Design looks at the resource overhead for a combination of three modules from Table 4.14: Decimate By 16, BPSK Demod, and Data Recovery. Because there can be

Table 4.15: Zynq Large Module vs Small Modules

Module	Requested Resources			Allocated Resources			Placements
	CLBs	BRAMs	DSPs	CLBs	BRAMs	DSPs	
Full BPSK Design	494	3	10	600	10	40	2695*
Full BPSK Module	415	3	11	500	10	20	5

no shared resources between these blocks, the overhead is significant. For example, although only 10 DSPs are required, 40 are allocated to this design. The number of placements for this design is calculated by assuming each of the placements for the submodules are independent. Placement for this will be the responsibility of the design assembly module placer. The best case is that there are $55*7*7 = 2695$ different ways to place this design, but in practice some of these will conflict with one another and the static will restrict the number still further.

The second Full BPSK Module entry shows the area requirements for a single module that contains the full functionality of the previous three modules. Note that the number of requested resources changes between these two methods. The number of placements for this combined module is significantly less than for the separate modules: 5 vs 2695. This results in a significant loss of flexibility while placing. In addition, there is a reduction of design flexibility. Should one of the submodules need to change, the full module must be recompiled. If, instead, the three module version requires the same change, substitution only requires fetching the new module from the library. If this changed module is not in the library, only the smaller submodule would need to be built and recompiled, instead of the larger combined module.

Table 4.16: Zynq BPSK Design Area Penalty

	Resource Usage			Percent Overhead		
	CLBs	BRAMs	DSPs	CLBs	BRAMs	DSPs
ISE Reference	415	3	11	-	-	-
1 Combined TFlow Module	500	10	20	20%	333%	81%
3 Separated TFlow Modules	600	10	40	44%	333%	363%

4.3.2 Area Overhead

Building designs using TFlow adds area overhead. Module size is locked during assembly, and modules can reserve unused resources due to their size and shape. This is one of the prices that TFlow pays to complete assembly within the strict time requirement.

Table 4.16 describes three different module design possibilities and their area utilization. The first option is to build the module using ISE, the second option is to build it as a single TFlow module, and the third option is to build it as three separate TFlow modules. Functionality will remain identical between these three options. As seen in the table, creating a TFlow module adds significant overhead compared to the ISE reference module, reserving over 300% more BRAMs than the module requires. Separating the design into multiple modules adds even more overhead. Therefore, the logic density for a TFlow design will be less than ideal. However, this loss of design density is one of the trade-offs necessary to meet TFlow's assembly deadline. As such, TFlow is not an appropriate tool when attempting to maximize design density. For this application, ISE would be a better choice.

4.3.3 Area Penalty Amelioration

To ameliorate some of TFlow's area overhead, it is possible to reduce the size of the modules by no longer adhering to clock region boundaries during module shaping and placement. Instead, the smallest granularity is constrained by the size of the largest physical resource. For the Zynq, this is the BRAM36, which is five tiles high. In addition, the Zynq has a paired clocking structure. Thus, the minimum granularity is a height of five tiles and a width of two tiles. Table 4.17 shows the reduced overhead when dealing with the same reference modules as in Table 4.16. While this achieves a significant reduction in area overhead, it also adds considerable complexity to the placement problem. The penalty is due to the significantly larger number of placements possible for each module. With the new five tile granularity instead of the clock region's fifty tile size, at least ten times as many placements are possible. Therefore, if n is the number of placements normally, $n_{max} = 10 * n$. The selected placement algorithm, as mentioned in Section 3.9.2, has complexity $O(m * n)$, with m as the number of modules. The placement problem is thus $O(m * n_{max}) = O(10 * m * n)$. Therefore, reducing the granularity of the modules will add an order of magnitude more complexity to the placement problem. As the overriding concern of TFlow is time, this is an unacceptable trade-off. However, should the placement complexity drop significantly or additional time budgeting be given to placement, this issue can be revisited.

Table 4.17: Zynq BPSK Design Area Penalty With Sub-frame Modules

	Resource Usage			Percent Overhead		
	CLBs	BRAMs	DSPs	CLBs	BRAMs	DSPs
ISE Reference	415	3	11	-	-	-
1 Combined Sub-frame Module	420	6	12	1%	200%	9%
3 Separated Sub-frame Modules	545	4	14	31%	33%	27%

4.3.4 Area Overhead Versus Placement Time

Section 4.3.3 discussed the overhead penalty for the 7 Series devices. This overhead penalty can be generalized onto other Xilinx devices. The actual overhead in the form of resource wastage cannot be determined for a generic module because TFlow has a limited number of possible module sizes. For example, the smallest CLB-only TFlow 7 Series module will be allocated 100 CLBs. Any unused CLBs are wasted. Depending on the module's resource usage, wastage can vary wildly. The only way to categorize possible module wastage for a generic module is to determine the device's minimum granularity.

TFlow module granularity can extend from the size of the device's clock region granularity down to size of the largest single resource, normally the BRAM. Changing the size of a TFlow module will only be useful if module placement is no longer aligned with the device's clock region. Removing this restriction increases the number of possible placements significantly. This is a fixed increase based on the properties of the architecture, as shown in Table 4.18. Placement will require additional computational effort to deal with these new placements. This new placement strategy will allow for smaller modules. Since the penalty is fixed, the best option for the new module size is the minimum possible granularity.

Table 4.18: Minimum Granularity Vs Placement Time

Architecture	Minimum Granularity			Sub-frame Granularity			Sub-frame
	CLBs	BRAMs	DSPs	CLBs	BRAMs	DSPs	Placement Penalty
Spartan 6	16	4	4	4	1	1	x4
Virtex 4	16	4	8	4	1	2	x4
Virtex 5	20	4	8	5	1	2	x4
Virtex 6	40	8	16	5	1	2	x8
Series 7	100	10*	20*	10	1†	2†	x10

*Also requires 50 CLBs. †Also requires 5 CLBs.

Table 4.18 shows the minimum module granularity for both clock region and sub-frame modules for multiple Xilinx architectures. Newer architectures tend to have larger clock regions, so sub-frame modules can have a larger impact in reducing area overhead. However, the larger clock region also means that there is a larger placement penalty when no longer aligned placement to the clock region boundaries. In addition, the 7 Series devices must always have an even number of columns due to the structure of their clock tree. This increases the minimum module size and can increase area overhead.

Depending on the time budget for placement, this penalty may be worth paying to gain access to smaller modules and less area overhead. TFlow currently prioritizes speed over area, and thus does not use sub-frame modules. However, this would change were the amount of time given for placement increased.

4.3.5 Zynq 7 Static Design

The amount of logic and routing that can be pushed into the static design directly influences both the eventual speed of design assembly as well as the capabilities of the design. As the

static grows in size, the resources available for the modules shrinks. Moving logic into the static increases the speed of design assembly at the cost of flexibility.

For the static portion of the design, the timing information that is relevant to TFlow is the rate of the clocks going into the blacktop region. Clocks that are wholly internal to the static are not relevant to TFlow, although if they do not meet timing, the static will not function correctly. This, however, would be solved during static creation using the standard timing tools.

Static creation for this complex Zynq design takes 1407.5 seconds. This static is built for the Zynq implementation of GReasy [89]. This is much longer than the Virtex 5 static, as it is a much larger device and a more complex design. From TFlow's perspective, this long static creation time will not impact the speed of assembly, but it does mean that S_T from Equation 3.10 will be large and so the compilation time saved should be considerable.

The relevant clocks for this design can be seen in Table 4.19. Unlike the Virtex 5 example, this static design has two clocks available for the modules. These are the maximum possible clock rates; the clocks are run at a lower frequency. BT_CLK runs at 61.44 MHz while BT_CLK_DAC runs at exactly twice that - 122.88 MHz. Either or both clocks can be routed to a module in the design using TFlow's clock router.

This static design is complex, and includes interfaces with both the on-die ARM and an external DAC. Therefore, it is expected that TFlow will show a significant improvement over the standard flow, since all of these calculations are cached in the library. The following

Table 4.19: Zynq-7000 Static Timing Analysis

Clock	**Frequency (MHz)**
BT_CLK	148.1
BT_CLK_DAC	161.4

section will show that these improvements enable TFlow to meet the attention span deadline.

4.3.6 Design Assembly on the Zynq 7 Architecture

Multiple different designs were built using TFlow's rapid design assembly technique. As anticipated, TFlow's design assembly process took orders of magnitude less time than the standard Xilinx flow, fast enough to meet the hard deadline.

TFlow was compared against the Xilinx reference flow for this set of test cases. Neither QFlow nor HMFlow support the 7 Series, and thus direct comparison is not possible. The amount of time it takes to run back-end compilation and generate a bitstream is used to measure the performance difference between TFlow and the Xilinx flow.

These designs were built for the Zynq XC7Z045 architecture on a desktop machine using an Intel i7-2600 with 8 GB of DRAM. Four different designs were assembled. TFlow used GReasy as the front-end, while the Xilinx flow used ISE.

The BPSK design includes three modules. The first is a decimator, which reduces the signal by 16. This is followed by the BPSK demodulator to convert the waveform into binary. This is followed by the data recovery module, which takes this stream and converts it into ASCII to recover the initial text.

The tuner example takes a signal as input and will tune the frequency to the desired carrier wavelength. It has additional functionality to allow this frequency to be selected at run-time, but this is not done using TFlow. Instead it has an integrated parameterization port that connects into the static for frequency adjustment. Connecting these parameterization ports into the static is done using TFlow's router.

The DES Encryption design implements the Data Encryption Standard (DES) symmetric-key algorithm. This algorithm is highly parallelizable and thus is a reasonable design to implement on an FPGA.

The ZigBee Receiver is a radio receiver that takes in a modulated ZigBee waveform and returns the binary signal. This data can then pass through a data recovery stage to be converted into ASCII or fed into other modules or outputs for additional processing.

The time requirements for assembly of these designs can be seen in Table 4.20. Three different options were explored. The first was using the standard Xilinx toolflow to generate the full design. The next two options both use TFlow. However, one uses the XML metadata format and the second uses the binary metadata format. As seen, TFlow runs significantly faster than the standard flow. While compiling the full design for ISE took approximately twenty minutes, TFlow completed in seconds. The difference between the time required for XML and binary metadata formats can also be seen due to the large read and write times required for XML. A production version would use the binary format, which can complete a design in less than ten seconds. This is more than two orders of magnitude faster than ISE and handily completes within the attention span deadline. This validates the model's

Table 4.20: Zynq Design Build Time

Design	Time (s)			BIN vs ISE
	ISE	TFlow		
		XML	BIN	
BPSK Receiver	1184	22	8.5	139x
Frequency Tuner	1015	17	6.5	156x
DES Encryption	1122	18	8.7	129x
ZigBee Receiver	1106	21	7.4	149x

modular design approach. In addition, this puts TFlow assembly time squarely in the realm of software compilation.

Comparing the ISE assembly time against the static creation time in Table 4.19 reveals an interesting data point: static creation takes longer than assembling the full design in ISE. This is due to the additional processing that must be done to constrain the static to keep out of the sandbox region. Logic placement for the static is limited, and thus the optimal placements for this design may not be available. Additionally, route blockers are added to the design to force routing to stay outside the sandbox. When the Xilinx tools then route the static, they are again limited in how they can optimize. Many routes may end up congesting the region just outside the sandbox, as these are the closest they can get to their optimal cross-sandbox path. Adding and removing the route blocker also adds to the time requirement for static generation. These static constraints slow the compilation time for static generation. In return, TFlow can assemble the design in seconds. While the overhead for populating and assembling the TFlow libraries may be longer than building the design once, every subsequent run of TFlow yields significant gains and can be done without losing the user's attention.

Table 4.21: Total Design Time including Precompilation (s)

Design	Static	Module(s)	Assembly	TFlow	ISE
BPSK Receiver	1407.5	(480.3 + 490.5 + 502.1)	8.5	2888.9	1184
Frequency Tuner	1407.5	461.7	6.5	1875.7	1015
DES Encryption	1407.5	523.2	8.7	1939.4	1122
ZigBee Receiver	1407.5	455.9	7.4	1870.8	1106

To put TFlow's assembly time into perspective, running just Xilinx Bitgen for the Zynq takes 96.7 seconds, over eleven times longer than the time TFlow takes to assemble a full design.

Model Analysis

TFlow's approach does not perfectly mirror the idealized version of the model from Equation 3.2. Instead, global optimizations and overhead must be added as a negative term, G, as shown in Equation 3.3. Since TFlow's static and modules are built independently, no global optimization can take place. In addition, any overhead will occur for each block. For example, there is initialization overhead when running the Xilinx router. This overhead will occur when compiling every library component. When running as one monolithic flow, initialization only needs to occur once. G could therefore be added to equations 3.6 and 3.7 in Section 3.3, where combining each of the placement or routing steps into a monolithic process can reduce the total complexity. However, this is only an issue if these global optimizations can overcome the speed advantages from precomputation.

Equation 4.1 can be used to determine the amount of time that can be saved through global optimization. From this equation, only when global optimization, G_I, is large enough to make

monolithic design time D_I less than TFlow assembly time T_A does modular precomputation become contraindicated.

$$T_S + T_M + T_A - G_I = D_I \tag{4.1}$$

For example, when creating the DES Encryption design, $T_S = 1407.5$, $T_M = 523.2$, $T_A = 8.7$, and $D_I = 1122$. These are static creation time, module creation time, assembly time, and monolithic design time, respectively. The following equation determines the time savings from a monolithic flow.

$$1407.5 + 523.2 + 8.7 - G_I = 1122$$

$$G_I = 817.4$$

Global optimization can yield significant time savings, were the process to be considered as a whole. The overall price in time for separating the stages can be seen in Table 4.21. Fortunately, the strict time deadline for TFlow only involves the amount of time the assembly stage must take, not the setup and precompilation time. Since $T_A \ll D_I, \forall(T_A, D_I)$, these optimizations do not overcome the significant gains that come from precomputation, as seen in Equation 3.12. Global optimizations are therefore insufficient when attempting to perform deadline assembly. Pushing these computations out of the critical path is what enables TFlow to meet its goal, despite the drawbacks.

Table 4.22: Zed Design Build Time

Design	TFlow (s)
BPSK Receiver	60.9
Frequency Tuner	50.7
DES Encryption	52.5
ZigBee Receiver	50.4

Embedded TFlow

Another capability of TFlow is that it can run on embedded platforms such as ARM. The assembly speed for TFlow carries over to make embedded assembly occur rapidly. Unlike the standard tool flow, which does not run on these processors, TFlow is fully compatible. However, TFlow may be unable to meet the deadline with these less capable processors, and would therefore target different applications.

The Zynq architecture includes an embedded ARM processor. The TFlow design examples were run in this embedded environment, using a ZedBoard with a Zynq-7000 ZC702 FPGA. The included ARM processor is a dual-core Cortex-A9, running at 866 MHz with 1 GB of DRAM.

As seen in Table 4.22 the embedded version of TFlow takes about a minute to complete the same designs that finished in just seconds on the desktop. However, this is still more than an order of magnitude faster than running these test cases using ISE on a desktop, as reported in Section 4.3.6. As ISE does not run on the ARM, even this is not a fair comparison. With this capability, a ZedBoard can both implement a design and reconfigure its own FPGA. Dobson [90] has a more in-depth look at this capability.

Zynq Flow Analysis

Most importantly, the full flow, from the end of the design process where the user presses the button to begin implementation to having a functioning design programmed onto the FPGA, must complete within the allotted time to meet the strict time requirements of human attention span. As such, the implementation overhead for beginning TFlow and putting the design on the board, combined with the TFlow assembly time, must be less than the thirty second window.

To test whether this is possible, the best case design time performance must be analyzed. Assuming that TFlow can take zero seconds to assemble a design, the implementation overhead must still be taken into account. For this test case, the GReasy flow overhead will be analyzed. This overhead eats into TFlow's time allotment, and is the reason that HMFlow is unable to meet the time constraints, as seen in Section 4.2.2. HMFlow's overhead exceeded the time requirement significantly, removing it from contention.

A naive approach would program the Zynq FPGA via the JTAG interface with a full bitstream. This takes 68 seconds, and so is unacceptable. In addition, some of the ARM interfaces on the Zynq use the FPGA fabric. For the GReasy static design, the HDMI video port and the external Analog-to-Digital Converter (ADC) both have their connectivity routed through the FPGA. Fully reprogramming the FPGA resets these interfaces and the ARM does not properly recover. To address these issues, the GReasy flow for the Zynq board takes advantage of the embedded ARM's capability to program the FPGA. Since the

TFlow sandbox can be treated as a partial bitstream design, a method of partial bitstream generation using the differences between the base static design and the new bitstream is used. The ARM can then load this partial bistream; this maintains its critical interfaces and can complete within the critical time window.

Table 4.23 shows the overhead for the GReasy flow on the Zynq architecture. The desktop used in these tests is a Intel i7-2600 with 8 GB of DRAM and the Zynq ARM is an 866 MHz dual-core Cortex-A9 with 1 GB of DRAM. Design setup generates the input EDIF file for TFlow from the GNU Radio Flow Graph. This is fed into TFlow. After TFlow completes and gives GReasy a bitstream, the partial bitstream file is generated. This generation and programming stage also includes the ssh and scp communication between the ARM and the CPU. The scp communication is the transport mechanism for the bitstream. The ssh communication includes ARM configuration. After the FPGA is programmed, there is still additional FPGA configuration for systems like the ADC. Overall, this overhead adds up to 12.8 seconds. This leaves TFlow with 17.2 seconds to assemble a design.

4.4 Summary

As seen in Table 4.20, TFlow completes under 17 seconds, the adjusted time window including all implementation overhead. This is within the narrow window of human attention span. FPGA design implementation using TFlow can therefore complete within the human attention span window, validating the modular design model and opening up new productivity

Table 4.23: GReasy Zynq Overhead

GReasy Stages	Time (s)
Design Setup	1.3
TFlow	*See Table 4.20*
Partial Bitstream Generation / Programming	8.5
FPGA Configuration	3.0
Total Overhead	12.8

opportunities.

Chapter 5

Conclusion

This work is a significant improvement on the state of the art. Working designs can be built without losing the attention of the user, enabling the design flow to proceed smoothly. This capability is realized through the use of reusable and clonable modules that can be rapidly fetched and assembled in seconds. Throughout the process, assembly completion within the deadline has been the overriding concern that has informed each design decision. With large portions of the process performed prior to use, when speed is unnecessary, the amount of work required for assembly is drastically reduced.

5.1 Contributions

The presented model suggests that it is possible to achieve human span-scale results through aggressive modular design precomputation. The implementation and subsequent tests of

TFlow validate that assertion. Thus, FPGA design assembly can be made a seamless part of the tight design-build-test cycle, improving user focus and productivity. In addition to maintaining user attention, quicker build times also permit more runs to occur within the same time span. This leads to faster design space exploration as well as increased productivity.

1. **FPGA Design Model for Sub-Attention Span Implementation**

   This work reformulates the design process to perform assembly within the constraints of the human attention span. It implements this reformulation using modular design and module precompilation. The presented model describes a way to move much of the computational effort of implementation out of the critical path. This is done by separating the design into modules, and running each of them through most of the implementation process beforehand. These modules are then used as building blocks when assembling the desired design, significantly speeding the process.

2. **Model Proof - TFlow**

   The proof that the presented design process model is effective is shown through the use of the rapid assembly tool, TFlow, which can assemble bitstreams within the thirty second window required by the limits of human attention span. Designs built by TFlow on the Zynq have assembly times that are less than nine seconds. No other existing flow can assemble designs within the thirty second requirement, as prior attempts did not fully precompile the modules. For example, HMFlow does not precompile the

bitstream, as shown in Equation 3.20, and thus is unable to complete within the time constraint, as shown in Section 4.2.2. ISE, QFlow, and TFlow ZigBee Radio designs were assembled for the Virtex 5 XC5VLX110T, using a 2.83 GHz Intel Core 2 Quad with 3 GB of DRAM. Whereas ISE completes in 236 seconds and QFlow in 158 seconds, by following the design model, TFlow meets the attention span deadline with a time of 15 seconds.

(a) **Placement**

To implement deadline assembly, a modular preplacement constraints-based solver was implemented and used during precompilation. A module placer was implemented to simplify and deterministically solve the placement problem on a range of devices within the specified time constraint.

The module preplacer is enables moving computations out of the critical path, in accordance with the model. The module preplacer generates a set of all valid placements for a module prior to design assembly [81]. This tool is competitive with other placement tools [81], but unlike the alternatives, it is portable to all TORC-supported devices. All of the work done by this preplacer is work that the module placer will not need to perform at design time, enabling the module placer to meet the deadline.

The implemented deadline placer is both fast and efficient; placement can complete in fractions of a second. Placement time for the sequential placement algorithm is less than two seconds for the most complicated test case with the number

of modules, $m = 147$ and the number of placements, $n = 62$, running on a 2.83 GHz Intel Core 2 Quad with 3 GB of DRAM. As was shown in Table 4.4, the brute force method fails to yield a result and other, faster placement strategies yield results 4 times worse. Even hand-optimized placement for this design yields a result that has 1.4 times worse quality. These results are consistent, with the sequential placer rapidly producing high quality results for complex placement problems. While placement is a well-explored field, most placers have focused on high granularity placement strategies. Modular placement is much more restrictive, and TFlow requires deadline placement as well. Thus, TFlow's placement strategy focuses on achieving the best solution within the strict time budget. This placer is built on top of the TORC tools, and thus can be easily moved to other TORC-supported architectures. This high granularity placer contributes to the field by being flexible and efficient while still meeting the deadline imposed by the user attention span.

(b) **Router** The router from [84] was adapted for use with TFlow. This router performed 7.8 times faster than the standard ISE tools on an Intel i7-2600 with 8 GB of DRAM. The router performs inter-module connectivity; intra-module connectivity occurs during module creation using the standard router.

(c) **Bitstream Assembly** TFlow assembly occured at the bitstream level. This was possible through the created bitstream relocation tools. Module bitstreams [66] and routing micro-bitstreams [69] were stitched together at run-time to generate

the desired design.

(d) **Meta-data Description** A data structure that could contain the overarching logical and physical description of each module, static, and design was built. This data structure is automatically populated and processed to control TFlow assembly.

3. **Tool Independence Corollary**

   Analysis of the standard tools revealed that they could not meet the time deadline. Consequently, the created assembly flow is not tied into the Xilinx toolchain, and thus can be used in otherwise unsupported applications.

   Other approaches to modular design assembly, such as HMFlow [28] and QFlow [29] require reintegration with the Xilinx tool flow prior to bitstream generation. In contrast TFlow's design assembly approach is independent from the Xilinx tools. Removing the reliance on on the Xilinx tools is a significant contribution that enables standalone applications such as embedded and autonomous applications [90]. Running TFlow on a 866 GHz dual-core Cortex-A9 with 1 GB of DRAM can generate bitstreams in less than 60 seconds, while this is impossible using other contemporary tools.

This modular design methodology rearranges the computational effort to enable second-long assembly times to meet the constraints of human attention.

5.2 Ongoing and Future Work

TFlow has been integrated as the back-end for the GNU Radio environment. GNU Radio normally uses software blocks to perform software radio design. The TFlow extension to GNU Radio, GReasy, adds in the capability to have hardware FPGA blocks for software/hardware radio design [91] [92] [93]. This library of hardware FPGA blocks has been successfully integrated into the GReasy environment [76].

As always, there is room for additional work to expand TFlow's core competencies into a wider range of applications.

While the current tools support sub-frame modules, the process is only semi-automated. Each sub-frame version of a module is treated independently by the tools. This is only an issue during placement, when choosing different sub-frame versions of a module is done manually. Sub-frame modules are fully supported by TFlow's current routing and bitstream generation tools. However, library management of these independent sub-frame modules is a manual process. Proper library management would allow for the correct sub-frame version of the bitstream to be fetched. At the moment, each module has a single associated bitstream. Adding in a one-to-many mapping for library management is necessary for full automation. This issue is also applicable when discussing multiple shapes for a module. The current independent module flow requires manual intervention to determine which shapes should be used. Expanding the library management procedure to handle these types of one-to-many mappings are necessary if the gains for multiple shapes and smaller module granularity are

to be realized; thus, a fully automatic library management system is a promising avenue for future work, as it would enable sub-frame modules and multiple shapes to be used easily.

TFlow could be extended to perform hierarchical design. Smaller TFlow modules could be built, placed, and assembled together into a larger TFlow module. This larger module could then be either combined with other modules or used as part of a design. Complicated modules could be built from a library of smaller modules. Using these larger modules gives the benefit of hierarchical design, including abstracting away implementation details for ease of use. However, area overhead incurred by each submodule is cumulative, making the combined module larger than if it were built as a single module.

The current TFlow device support consists of those architectures supported by TORC. This limits the flow to Xilinx devices. Altera devices are completely unsupported, due to their tool's closed nature. A future area for research is to investigate Altera's willingness to allow for more in-depth information regarding low-level device architecture, module implementation, and bitstream organization so that TFlow support is possible.

TFlow currently restricts the design process from exceeding the attention span limit. As the time necessary for downstream stages is not known, each step seeks to complete as rapidly as possible. One avenue for future work is to give each stage its own hard deadline. Each stage could use an iterative algorithm to fill the time searching for better solutions. Once its deadline is reached, the current best solution can be passed to the next stage. This would yield better results without violating the attention span deadline. Placement could run both a sub-frame and clock region placer. If sub-frame placement completed within the allotted

time, the advantages for sub-frame modules could be gained without penalizing the ability to meet the attention span deadline. Sub-frame modules yield better packing and less resource wastage.

Another opportunity for growth is to release an open-source or service-based version of TFlow for use by outside parties. This would enable real-world rapid assembly and testing. With additional users, areas for improvement could be better identified, and the advantages of TFlow could be realized by a wider audience.

In addition, releasing TFlow to outside parties would enable users from a wider range of application domains to use this tool. Currently, TFlow has been integrated with GReasy to explore the software-defined radio domain. Other application domains, such as digital signal processing, communication interface translation, or machine learning, can use TFlow to good advantage. For example, a machine learning application could run TFlow to update connectivity or internal blocks based on current performance. Cognitive radio, with access to TFlow modules, could choose new radio modules based on spectral analysis and rapidly reconfigure for better performance. Applying TFlow to these domains is a promising avenue for future work.

Section 3.3.2 covered the trade-offs inherent to preprocessing portions of the design. TFlow does not precompile every term in the model, and so it would be possible to reduce the computational effort still further at the cost of design flexibility. Future work investigating design strategies that preprocess these additional terms would be valid avenues for meeting tighter deadlines. Future work could also reduce the amount of precompilation should more

time be available. Determining which terms to precompile and which ones to leave for design time is a promising avenue for future work.

The model terms from Equation 3.9 could also be used as a framework for categorizing precompilation flows. This spectrum would extend from flows without precompilation, such as ISE, down to a flow that has a library of prebuilt design bitstreams. Future work could categorize these flows based on which of the nine terms are precompiled. Any new precompilation flow could be slotted into this system to determine similar design tools and yield more precise comparisons.

Bibliography

[1] Lenz and T. Moeller, *.NET - A Complete Development Cycle*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[2] N. Hamilton and D. H. Society, *From Spitfire to microchip: studies in the history of design from 1945*, ser. History of design. Design Council, 1985.

[3] Ruttan, *Is War Necessary for Economic Growth?* Oxford University Press, 2006.

[4] M. Csikszentmihalyi, *Flow: The psychology of optimal experience*. HarperPerennial New York, 1991, vol. 41.

[5] Y. Itō, *Modular Design for Machine Tools*, ser. McGraw Hill professional. McGraw-Hill, 2008.

[6] “ISO 2562, 2727, 2769, 2891, 2912, and 2934 (Modular Units for Machine Tool Construction),” International Organization for Standardization, Standard, 1973.

[7] Y. Doi, “On Application of BBS,” Toyoda Iron Works, Toyoda Technical Report 4(3): 22-32, 1963.

[8] Y.-P. Luh, C.-C. Pan, and J.-W. Su, "A study on modular design representation," in *Industrial Engineering and Engineering Management, 2007 IEEE International Conference on*, Dec 2007, pp. 1327–1331.

[9] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, 2nd ed. Boston, MA: Addison-Wesley, 2005.

[10] B. E. Bürdek, *Design: History, theory and practice of product design.* Birkhäuser, 2005.

[11] E. S. Almeida, A. Alvaro, V. C. Garcia, J. C. C. P. Mascena, V. A. A. Burgio, L. M. Nascimento, D. Lucrdio, and S. R. L. Meira, *C.R.U.I.S.E: Component Reuse in Software Engineering.* Brazil: C.E.S.A.R e-book, 2007.

[12] J. Sametinger, *Software Engineering with Reusable Components.* New York, NY, USA: Springer-Verlag New York, Inc., 1997.

[13] J. D. Gaeddert, "Liquid dsp library," 2013. [Online]. Available: https://github.com/jgaeddert/liquid-dsp

[14] H. Zhu, *Software Design Methodology: From Principles to Architectural Styles.* Elsevier Science, 2005.

[15] K. E. Wiegers, *Software Requirements*, 2nd ed. Redmond, WA, USA: Microsoft Press, 2003.

[16] . Altmann, E. M and J. G. Trafton, "Task interruption: Resumption lag and the role of cues," in *Proceedings of the 26th annual conference of the Cognitive Science Society*, 2004, pp. 42–47.

[17] M. Czerwinski, E. Horvitz, and S. Wilhite, "A diary study of task switching and interruptions," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '04. New York, NY, USA: ACM, 2004, pp. 175–182.

[18] R. B. Miller, "Response time in man-computer conversational transactions," in *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, ser. AFIPS '68 (Fall, part I). New York, NY, USA: ACM, 1968, pp. 267–277.

[19] S. K. Card, G. G. Robertson, and J. D. Mackinlay, "The information visualizer, an information workspace," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '91. New York, NY, USA: ACM, 1991, pp. 181–186.

[20] J. Nielsen, *Usability Engineering*, ser. Interactive technologies. Morgan Kaufmann, 1993.

[21] B. A. Myers, "The importance of percent-done progress indicators for computer-human interfaces," *SIGCHI Bull.*, vol. 16, no. 4, pp. 11–17, Apr. 1985.

[22] B. O'Conaill and D. Frohlich, "Timespace in the workplace: Dealing with interruptions," in *Conference Companion on Human Factors in Computing Systems*, ser. CHI '95. New York, NY, USA: ACM, 1995, pp. 262–263.

[23] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: towards an open-source tool flow," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 41–44.

[24] V. Betz and J. Rose, "VPR: a new packing, placement and routing tool for FPGA research," in *Field-Programmable Logic and Applications*, ser. Lecture Notes in Computer Science, W. Luk, P. Cheung, and M. Glesner, Eds. Springer Berlin Heidelberg, 1997, vol. 1304, pp. 213–222.

[25] J. Teich, "Hardware/software codesign: The past, the present, and predicting the future," *Proceedings of the IEEE*, vol. 100, no. Special Centennial Issue, pp. 1411–1430, 2012.

[26] R. Hoetzlein, "Graphics performance in rich internet applications," *Computer Graphics and Applications, IEEE*, vol. 32, no. 5, pp. 98–104, 2012.

[27] R. Tessier, *Fast Place and Route Approaches for FPGAs.* Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1999.

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

[29] T. Frangieh and P. Athanas, "A design assembly framework for fpga back-end acceleration," in *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, 2012, pp. 1–6.

[30] C. Zeh, "Incremental design reuse with partitions," June 2007, xilinx XAPP918 (v1.0). [Online]. Available: www.xilinx.com/support/documentation/application_notes/xapp918.pdf

[31] G. Herrmann and G. Dost, "Entwurf und technologie von mikroprozessoren," in *Taschenbuch Mikroprozessortechnik, 3. neu bearbeitete Auflage*, T. Beierlein and O. Hagenbruch, Eds. Fachbuchverlag Leipzig im Carl Hanser Verlag Mnchen Wien, May 2004.

[32] T. Winograd, "Breaking the complexity barrier again," *SIGIR Forum*, vol. 9, no. 3, pp. 13–30, Nov. 1973.

[33] Xilinx, "Xilinx CORE Generator System." [Online]. Available: http://www.xilinx.com/tools/coregen.htm

[34] R. Smith and J. Timberlake, *Prefab Architecture: A Guide to Modular Design and Construction*. Wiley, 2011.

[35] IBM, "IBM Archives: 1928." [Online]. Available: http://www-03.ibm.com/ibm/history/history/year_1928.html

[36] D. Fisk, "Programming with Punched Cards," 2005. [Online]. Available: http://www.columbia.edu/cu/computinghistory/fisk.pdf

[37] M. Smith, *Application-Specific Integrated Circuits*, ser. Addison-Wesley VLSI Systems. Addison Wesley Professional, 1997.

[38] P. Maidee, C. Ababei, and K. Bazargan, "Timing-driven partitioning-based placement for island style FPGAs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, no. 3, pp. 395–406, March 2005.

[39] M. Xu, G. Grewal, and S. Areibi, "StarPlace: A New Analytic Method for FPGA Placement," *Integr. VLSI J.*, vol. 44, no. 3, pp. 192–204, Jun. 2011.

[40] A. Love, W. Zha, and P. Athanas, "In pursuit of instant gratification for fpga design," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, 2013, pp. 1–8.

[41] A. Wold, A. Agne, and J. Torresen, "Module placement using constraint programming in run-time reconfigurable systems," in *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, May 2014, pp. 287–292.

[42] T.-H. Lin, P. Banerjee, and Y.-W. Chang, "An efficient and effective analytical placer for fpgas," in *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, May 2013, pp. 1–6.

[43] T. Frangieh, "A design assembly technique for fpga back-end acceleration," Ph.D. dissertation, Virginia Tech, 2012.

[44] A. Montone, M. D. Santambrogio, D. Sciuto, and S. O. Memik, "Placement and floorplanning in dynamically reconfigurable fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 4, pp. 24:1–24:34, Nov. 2010.

[45] Xilinx, "UG640: System Generator for DSP." [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/sysgen_user.pdf

[46] "Impulse Accelerated Technologies." [Online]. Available: http://www.impulseaccelerated.com/

[47] M. Lehky and S. Bilik, "Reducing fpga design modification time," in *VHDL International Users' Forum, 1997. Proceedings*, 1997, pp. 143–149.

[48] J. Cong and H. Huang, "Depth optimal incremental mapping for field programmable gate arrays," in *Design Automation Conference, 2000. Proceedings 2000*, 2000, pp. 290–293.

[49] M. Teslenko and E. Dubrova, "Hermes: Lut fpga technology mapping algorithm for area minimization with optimum depth," in *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, 2004, pp. 748–751.

[50] D. Singh and S. Brown, "Incremental placement for layout-driven optimizations on fpgas," in *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*, 2002, pp. 752–759.

[51] D. Leong and G. Lemieux, "Replace: An incremental placement algorithm for field programmable gate arrays," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, 2009, pp. 154–161.

[52] J. Emmert and D. Bhatia, "Incremental routing in fpgas," in *ASIC Conference 1998. Proceedings. Eleventh Annual IEEE International*, 1998, pp. 217–221.

[53] E. Keller, "JRoute: A Run-Time Routing API for FPGA Hardware," in *Parallel and Distributed Processing*, ser. Lecture Notes in Computer Science, J. Rolim, Ed. Springer Berlin Heidelberg, 2000, vol. 1800, pp. 874–881.

[54] E. L. Horta and J. W. Lockwood, "Automated method to generate bitstream intellectual property cores for virtex fpgas," in *Proc. Field Programmable Logic.2004*, 2004.

[55] Xilinx, "UG702: Partial Reconfiguration User Guide," 2010. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/ug702.pdf

[56] A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "Openpr: An open-source partial-reconfiguration toolkit for xilinx fpgas," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, 2011, pp. 228–235.

[57] P. Athanas, J. Bowen, T. Dunham, C. Patterson, J. Rice, M. Shelburne, J. Suris, M. Bucciero, and J. Graf, "Wires on demand: Run-time communication synthesis for reconfigurable computing," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, 2007, pp. 513–516.

[58] C. Beckhoff, D. Koch, and J. Torresen, "Go Ahead: A Partial Reconfiguration Framework," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, Apr. 2012, pp. 37–44.

[59] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder — A novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, Sept 2008, pp. 119–124.

[60] A. Otero, E. de la Torre, T. Riesgo, T. Cervero, S. Lopez, G. Callico, and R. Sarmiento, "Run-time scalable architecture for deblocking filtering in h.264/avc-svc video codecs," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, 2011, pp. 369–375.

[61] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "Rapidsmith: Do-it-yourself cad tools for xilinx fpgas," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, Sept 2011, pp. 349–355.

[62] C. Lavin, B. Nelson, and B. Hutchings, "Improving clock-rate of hard-macro designs," in *Field-Programmable Technology (FPT), 2013 International Conference on*, Dec 2013,

pp. 246–253.

[63] A. Arnesen, K. Ellsworth, D. Gibelyou, T. Haroldsen, J. Havican, M. Padilla, B. Nelson, M. Rice, and M. Wirthlin, "Increasing design productivity through core reuse, meta-data encapsulation, and synthesis," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, Aug 2010, pp. 538–543.

[64] A. Tavaragiri, "A Management Paradigm for FPGA Design Flow Acceleration," Master's thesis, Virginia Tech, 2011.

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

[66] Xilinx, "UG191: Virtex-5 FPGA Configuration User Guide," 2012. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug191.pdf

[67] D. Montminy, R. Baldwin, P. Williams, and B. Mullins, "Using relocatable bitstreams for fault tolerance," in *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, 2007, pp. 701–708.

[68] T. Becker, W. Luk, and P. Y. K. Cheung, "Enhancing relocatability of partial bitstreams for run-time reconfiguration," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, 2007, pp. 35–44.

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

[70] E. Hung and S. J. E. Wilton, “Limitations of incremental signal-tracing for fpga debug,” in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, 2012, pp. 49–56.

[71] D. Grant and G. Lemieux, “A spatial computing architecture for implementing computational circuits,” in *Microsystems and Nanoelectronics Research Conference, 2008. MNRC 2008. 1st*, 2008, pp. 41–44.

[72] H. Mili, F. Mili, and A. Mili, “Reusing software: issues and research directions,” *Software Engineering, IEEE Transactions on*, vol. 21, no. 6, pp. 528–562, Jun 1995.

[73] W. Frakes and K. Kang, “Software reuse research: status and future,” *Software Engineering, IEEE Transactions on*, vol. 31, no. 7, pp. 529–536, July 2005.

[74] T. Frangieh, R. Stroop, P. Athanas, and T. Cervero, “A modular-based assembly framework for autonomous reconfigurable systems,” in *Reconfigurable Computing: Architectures, Tools and Applications*, ser. Lecture Notes in Computer Science, O. Choy, R. Cheung, P. Athanas, and K. Sano, Eds. Springer Berlin Heidelberg, 2012, vol. 7199, pp. 314–319.

[75] “IEC 61690-2-2000 (Electronic design interchange format (EDIF) - Part 2: Version 4.0.0),” International Electrotechnical Commission, Standard, 2000.

[76] R. Stroop, "Enhancing GNU Radio for Run-Time Assembly of FPGA-Based Accelerators," Master's thesis, Virginia Tech, 2012.

[77] Y. Hamadi, E. Monfroy, and F. Saubion, *Autonomous Search*, ser. SpringerLink : Bücher. Springer, 2012.

[78] Xilinx, "UG628: Command Line Tools User Guide," 2013. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/devref.pdf

[79] R. Kamat, S. Shinde, and P. Gaikwad, *Harnessing VLSI System Design with EDA Tools*. Springer Science+Business Media B.V., 2011.

[80] W. Chen, *The VLSI Handbook*, ser. Electrical Engineering Handbook. CRC Press, 2010.

[81] A. Sohanghpurwala, P. Athanas, and A. Love, "A Device-Agnostic Tool For Precomputing Legal Placements in Modular Design Flows," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig 2014)*, 2014.

[82] C. Lavin, B. Nelson, and B. Hutchings, "Impact of hard macro size on fpga clock rate and place/route time," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sept 2013, pp. 1–6.

[83] E. Blossom, "GNU Radio: Tools for Exploring the Radio Frequency Spectrum," *Linux J.*, vol. 2004, no. 122, pp. 4–, Jun. 2004.

[84] W. Zha, "Facilitating FPGA Reconfiguration through Low-level Manipulation," Ph.D. dissertation, Virginia Tech, 2014.

[85] K. Kepa, F. Morgan, and P. Athanas, "ERDB: An Embedded Routing Database for Reconfigurable Systems," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, 2011, pp. 195–200.

[86] Xilinx, "UG029: ChipScope Pro Software and Cores User Guide," 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/chipscope_pro_sw_cores_ug029.pdf

[87] W. Zha and P. Athanas, "An fpga router for alternative reconfiguration flows," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, May 2013, pp. 163–171.

[88] T. M. Brewer, "Instruction set innovations for the convey hc-1 computer," *IEEE Micro*, vol. 30, no. 2, pp. 70–79, Mar. 2010.

[89] R. Marlow, "Making Radios with GReasy: GNU Radio With FPGAs Made Easy," Master's thesis, Virginia Tech, 2014.

[90] C. Dobson, "An Architecture Study on a Xilinx Zynq Cluster with Software Defined Radio Applications," Master's thesis, Virginia Tech, 2014.

[91] M. Carrick, J.-O. Jeong, R. Stroop, S. Deyerle, A. Love, P. Athanas, and C. Dietrich, "Rapid FPGA Deployment with GNU Radio," 2012 Wireless Personal Communications Symposium, May 2012.

[92] R. Marlow, M. Carrick, S. Deyerle, A. Love, P. Athanas, and C. Dietrich, "Rapid FPGA Deployment with GNU Radio," 2013 Wireless Personal Communications Symposium, May 2013.

[93] P. Athanas, K. Kepa, C. Dobson, A. Love, R. Marlow, and K. Rooks, "Making Radios The GReasy Way," 2014 Wireless Personal Communications Symposium, May 2014.