Automatic Instantiation and Timing-Aware Placement of Bus Macros for Partially Reconfigurable FPGA Designs

Guruprasad Subbarayan

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Cameron D. Patterson, Chair

Thomas L. Martin

Michael S. Hsiao

November 19, 2010

Blacksburg, Virginia

Keywords: FPGAs, Partial Reconfiguration, Bus macros

Copyright 2010, Guruprasad Subbarayan

Automatic Instantiation and Timing-Aware Placement of Bus Macros for Partially Reconfigurable FPGA Designs

Guruprasad Subbarayan

(ABSTRACT)

FPGA design implementation and debug tools have not kept pace with the advances in FPGA device density. The emphasis on area optimization and circuit speed has resulted in longer runtimes of the implementation tools. We address the implementation problem using a divide-and-conquer approach in which some device area and circuit speed is sacrificed for improved implementation turnaround time. The PATIS floorplanner enables *dynamic* modular design that accelerates implementation for incremental changes to a design. While the existing implementation flows facilitate timing closure late in the design cycle by reusing the layout of unmodified blocks, dynamic modular design accelerates implementation by achieving timing closure for each block independently. A complete re-implementation is still rapid as the design blocks can be processed by independent and concurrent invocations of the standard tools. PATIS creates the floorplan for implementing modules in the design. Bus macros serve as module interfaces and enable independent implementation of the modules. The dynamic modular design flow achieves around 10× speedup over the standard design flow for our benchmark designs.

This work was supported by DARPA and the United States Army under Contract Number W31P4Q-08-C-0314.

To Mom and Dad

Acknowledgments

I would like to thank my advisor Dr. Cameron Patterson for his guidance throughout my graduate life at Virginia Tech.

I would also like to thank Dr. Thomas Martin and Dr. Michael Hsiao for serving in my committee.

A special thanks to my fellow project members Athira Chandrasekharan, Sureshwar Raja Gopalan, Dr. Stephen Craven, Tony Frangieh and Yousef Iskander for their valuable feedback and the encouragement they provided.

Finally, I would like to thank my family and friends for their support and encouragement.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 3

1.3 Organization 4

2 Background **5**

2.1 FPGAs 5

2.2 FPGA Design Flow 8

2.2.1 Design Entry 8

2.2.2 Design Implementation 10

2.2.3 Design Verification 11

2.3 Modular Design Flow 12

2.4 Improvements in Design Flow 14

2.5 Partial Reconfiguration 15

2.6 Need for a New Design Flow 16

3 System Overview 18

3.1 Xilinx EAPR Flow 18

3.2 PATIS - A Modified PR Flow 22

3.2.1 Automatic Floorplanner 25

3.2.2 Incremental Floorplanner 28

3.2.3 Bus Macro Insertion and Placement 29

3.3 Debug Support 30

3.4 Experimental Verification — Proof of Concept 31

4 Bus Macro Insertion and Placement 34

4.1 Bus Macros 34

4.1.1 Best Practices for Bus Macro Placement 37

4.2 Related work 40

4.3 Functional Flow 41

4.4 Bus Macro Insertion Tool 42

4.5 Bus Macro Placement Tool 46

5 Results 51

5.1 Platform Description 51

5.2 Benchmark Suite 52

5.3 Automatic Bus Macro Placement Performance 54

5.4 DMD Flow Performance 64

6 Conclusions and Future Work 67

6.1 Future Work 69

Bibliography 71

List of Figures

2.1 Basic FPGA layout 6

2.2 Xilinx Virtex-4 FPGA layout 7

2.3 Xilinx standard design flow 9

2.4 Xilinx MDF 13

2.5 Xilinx PR flow 16

3.1 Various regions in a PR design 19

3.2 PR implementation flow 20

3.3 PlanAhead screenshot 22

3.4 Overall DMD flow 24

3.5 Automatic floorplanner flow 26

3.6 Slicing tree with alternate horizontal and vertical cut directions 27

3.7 Incremental speculative floorplanner high-level flow 28

3.8 LLD interaction 31

3.9 HLV flow block 32

4.1 Screenshot of fpga_editor with left-to-right bus macro 36

4.2 Screenshot of fpga_editor with top-to-bottom bus macro 37

4.3 PR region with input bus macros 38

4.4 PR region with output bus macros 39

4.5 PATIS bus macro placement and insertion functional flow 42

4.6 Automatic bus macro insertion flow 44

4.7 Module before and after insertion of bus macros 45

4.8 Automatic bus macro placement flow 47

4.9 FPGA Editor screenshot with bus macros and interconnections 48

4.10 Two CLB-wide region on module boundaries 49

5.1 Design – `CFFT 6` 55

5.2 Design – `MB 5` 56

5.3 Design – `Viterbi 7` 57

5.4 Design – `FloPoCo 8` 58

5.5 Design – `FloPoCo 10` 59

5.6 PATIS runtime improvements 66

List of Tables

3.1 Xilinx tool flow execution times 32

5.1 Delay relative to different bus macro types 60

5.2 Comparison of manual and PATIS bus macro placement 60

5.3 Maximum delay on nets 61

5.4 Tool runtimes 63

5.5 Place-and-route times after floorplanning 64

Chapter 1

Introduction

Originally, Field-Programmable Gate Arrays (FPGAs) were developed to be used as glue logic for complex systems. Aided by the advances in semiconductor manufacturing technology, FPGAs grew in size and complexity, thereby widening the scope of their application. Also, the low non-recurring engineering costs and the reprogrammability feature of FPGAs has helped their proliferation. FPGAs are nowadays used for implementing complex systems in application areas such as networking, cryptography, high-performance computing, embedded systems, and digital signal processing.

1.1 Motivation

FPGA implementation tools have not kept pace with the advances in FPGA device density. Design changes to large FPGA designs are tedious as the implementation tools run for a long time to achieve timing closure. The longer runtimes are particularly undesirable during the design development phase. Existing incremental design flows re-use the design implementation data from previous runs to reduce the implementation tool runtimes. Al-

though, such re-use is useful during the end of the design cycle, the existing incremental design flows require complete re-implementation of the design when large changes are made. Sometimes, even small changes may require re-implementation of all the modules when one of the modules is not able to meet timing. Using a software analogy, this is similar to re-building all the libraries required for an application when a module changes. It is desirable to create a methodology that supports incremental design changes and also provides avenues to accelerate a complete re-implementation.

Place-and-route (PAR) of the design is the time-consuming step in the implementation process. In the past, fast implementation was achieved by use of faster processors, re-use of implementation data, and more efficient FPGA routing architectures. The need for reproducible results and possible trade-offs in the quality of results has limited the parallelization of PAR tools. Hence, due to lack of parallelism, the implementation tools do not gain significantly from recent multi-processor architectures. The Dynamic Modular Design (DMD) flow instead uses concurrent invocations of the standard tools, thereby leveraging multi-processor architectures to speedup the design implementation phase. Since the modules in the design are implemented in independent regions, incremental changes require re-implementation of only the changed modules.

The DMD flow leverages the features offered by the Xilinx Partial Reconfiguration (PR) tool flow to accelerate design implementation. A *Partial module-producing, Automatic, Timing-aware, Incremental, Speculative* (PATIS) floorplanner is created to automate the design floorplanning, bus macro insertion and placement steps in the PR flow. Bus macros are pre-placed and pre-routed hard macros used for connecting signals to a PR module. The use of PATIS frees the designer from the design and debug complications that exist in a traditional PR flow. Even though the DMD flow uses the PR tools, the static design being implemented does not inherit the complexities involved in testing the design with hot-swapped modules.

Bus macros also help to provide a passive, readback-based observability of module ports.

1.2 Contributions

The work presented in this thesis is part of the PATIS project that developed a novel FPGA implementation flow aimed at improving FPGA design productivity. This thesis focuses on the automatic instantiation and timing-aware placement of bus macros for a PR design. The bus macro insertion and placement tool is an important component of the DMD flow since the PR flow requires manual instantiation and placement of bus macros in the design. In order to maintain consistency with the standard flow, it is desirable to reduce the interaction required from the designer with respect to bus macro instantiation and placement. A large design typically contains more than fifty bus macro instantiations, and any change to module ports might require changes to a large number of bus macro instantiations. The automatic bus macro insertion tool creates the bus macro instantiations required for the design and also updates the instantiations when changes are made to module ports. Thus, the tedious, time-consuming, and error-prone process of manually creating and updating the bus macro instantiations is avoided.

Bus macro placement is an instance of the general place-and-route problem [1]. Since the objective is to arrive at a good placement in the least possible time, the PATIS automatic bus macro placement tool performs placement-and-routing of the design with just the bus macros and its interconnections. The automatic bus macro placement tool aids designs that are constrained by delays on the intermodule connection nets. Also, as no approximate routing models are used, the results are reproducible when the actual design is implemented. When a design is not able to meet its timing requirements, PATIS generates a new floorplan. Minor changes were made to the PR flow in order to accelerate bitstream generation.

In summary, this thesis makes the following contributions to the project:

1. An automatic bus macro insertion tool creates the bus macro instantiations necessary to convert a static design to a PR design. The tool uses the information supplied by the automatic placement tool to instantiate the correct type of bus macro.

2. The automatic bus macro placement tool optimizes designs that are constrained by delays on intermodule nets.

3. A video filter design used as a proof-of-concept to illustrate the feasibility of the DMD flow. The static design was implemented as a PR design using the DMD flow, and it is shown that incremental changes to the design were implemented faster using the DMD flow compared to the standard flow.

1.3 Organization

This thesis is organized into six chapters. Chapter 1 presents the motivation behind the work and contributions to the DMD flow. Chapter 2 discusses background and related information used in the thesis. Chapter 3 provides an overall description of the DMD design flow. The chapter also presents the proof-of-concept design used to illustrate the effectiveness of the DMD flow. The remainder of the thesis focuses on the author's contribution to DMD and presents a detailed description and analysis of the automatic bus macro insertion and placement tool. Chapter 4 gives an in-depth description of the automatic bus macro insertion and placement tool. An analysis of the bus macro placement with practical designs is provided in Chapter 5. Chapter 6 summarizes the accomplishments and discusses future work.

Chapter 2

Background

This chapter discusses the background material pertaining to the Standard, Modular and PR design flows for Xilinx FPGA devices. Special emphasis is on the need for a new design flow.

2.1 FPGAs

FPGAs are programmable integrated circuits, containing a matrix of configurable logic, routing channels, I/O blocks and memory. The logic elements in the FPGA are connected through programmable interconnects and can implement any logic function an ASIC can implement, at the expense of die area and clock speed. In comparison to ASICs, FPGAs have lesser Non-Recurring Engineering costs and faster time to market. Additionally, as the designs are implemented on a tested and characterized device, there will be no need for re-spins as in the case of ASICs. However, FPGAs tend to have a higher per-unit cost, lower energy efficiency and are slower compared to ASICs, which limits their use to low volume applications.

Complex Programmable Logic Device (CPLD) is another class of programmable device that uses sea-of-gates and macro cells to implement a circuit. Compared to FPGAs, CPLDs are cheaper and have higher pin to pin speed. CPLDs are relatively simpler and contains fewer flip flops compared to FPGAs. The increased logic density in FPGAs compared to CPLDs comes from the use of configurable switch matrix to implement the circuits. FPGAs are based on SRAM and require a configuration to be downloaded at powerup whereas CPLDs are EEPROM-based and are active at powerup. Figure 2.1 shows the layout of a basic FPGA device. In contrast to CPLDs, most modern FPGAs contain heterogeneous components such as DSP slices, processor cores, memory and high speed transceiver capabilities to satisfy the high performance and bandwidth demands. Figure 2.2 is the layout of a Xilinx Virtex-4 device highlighting the basic resources.

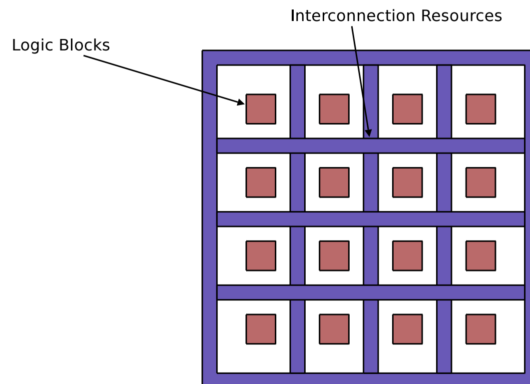

Figure 2.1: Basic FPGA layout

Configurable Logic Blocks (CLB)/Logic Array Blocks (LAB): The basic logic unit of FPGAs is a slice in case of Xilinx devices and an Adaptive Logic Module (ALM) in case of Altera FPGAs. A Virtex-6 slice contains four 6-input LUTs, eight registers, multiplexers and arithmetic carry logic. Two such slices form a Virtex-6 CLB. About 25 to 50 percent of these slices can be used as distributed RAMs or shift registers. In case of Altera Stratix-V device, an ALM is capable of implementing any six input function and ten such ALMs form a LAB.

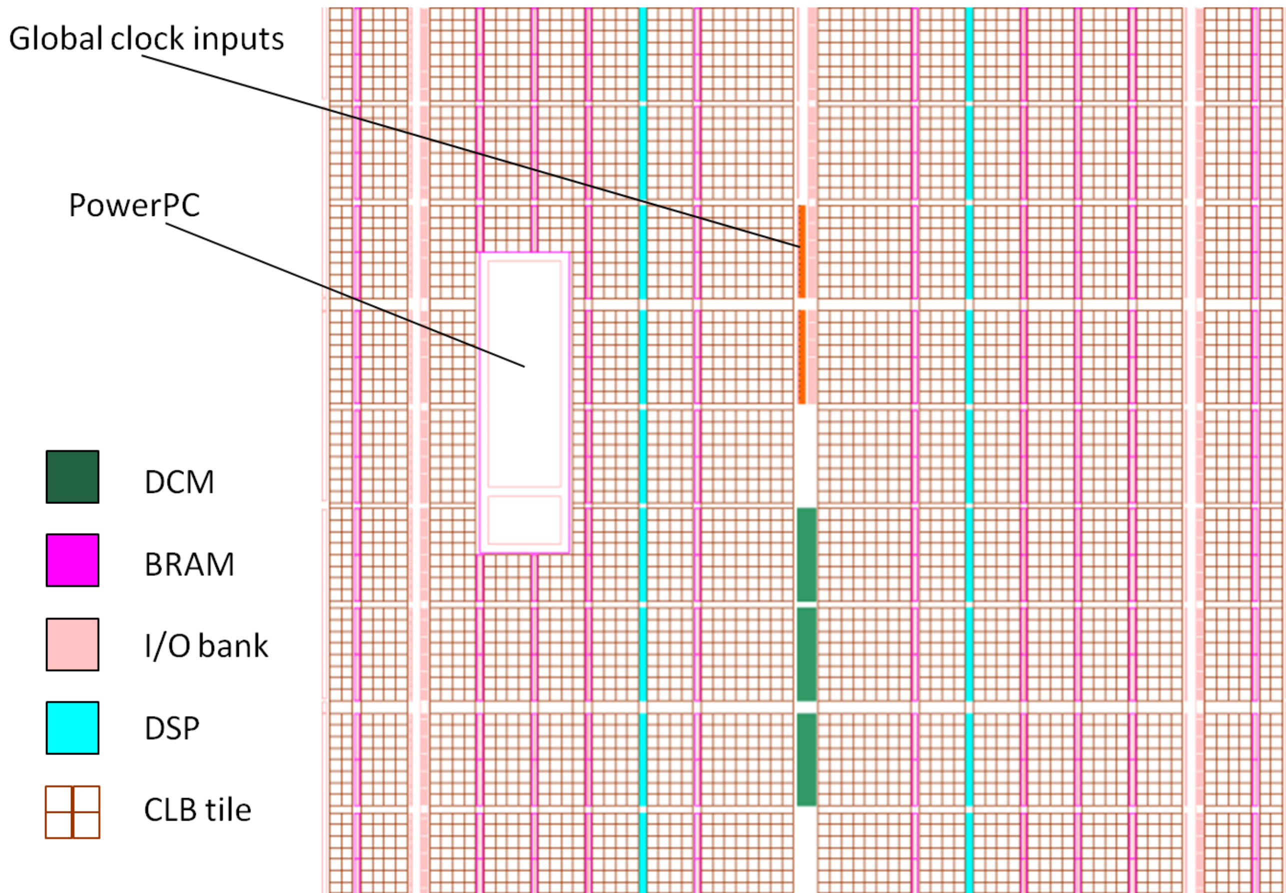

Figure 2.2: Xilinx Virtex-4 FPGA layout

Clock Managers: There are upto 9 Clock Management Tiles (CMTs) in a Virtex 6 device, each consisting of two Mixed Mode Clock Managers (MMCMs) that are PLL based. The Xilinx Virtex 6 devices provides six types of clock lines to address the high fanout, short propagation delay and low skew requirements. The MMCMs are feature enhanced Digital Clock Managers (DCMs). The clock lines are separated into global and regional clock lines. Global clocks are generally driven directly by the CMTs and the regional clock buffers can be driven by either of four clock capable input pins with optional frequency division. Virtex 6 devices also have direct connection from the CLMs to IOs to provide clocks to fast serializer/deserializer circuits.

Block RAMs (BRAM): BRAMs are on-chip memories present in modern FPGA devices and are generally dual-ported to support simultaneous read and write operations. Xilinx devices can provide upto 38304 Kb of memory in Virtex 6 devices as BRAMs apart from the distributed RAMs that can be created from CLBs. Each block RAM in Virtex 6 is 36kb in size and could be used as two independent 18kb blocks.

DSPs: DSP slices are multiplier/accumulator units present in FPGAs to efficiently implement signal processing functions. They can support pipelining and provide pre-adders, bitwise operations and dedicated cascade connections to improve the performance of signal processing related functions.

Other Resources: Modern FPGAs also contain resources such as integrated interface blocks for PCI Express designs and high speed trancerivers.

2.2 FPGA Design Flow

FPGAs, as explained in Section 2.1, contain heterogeneous resources that can be programmed to suit the needs of the designer. Vendor supplied tools for creating FPGA designs are often proprietary with some support for using third party tools in the design flow. For instance, most vendors support the common Electronic Design Interchange Format (EDIF) [2] for synthesized netlists and also have proprietary alternatives. Figure 2.3 shows the Xilinx standard design flow [3].

2.2.1 Design Entry

Designers use HDLs or schematics to create hardware designs. HDLs are preferred as they are easy to update and the designs can be ported between different vendor tools. The most

commonly used HDLs, VHDL and Verilog, are supported by most FPGA vendor tools. Third party vendor tools like Synplify and DC-FPGA provide support for synthesis of SystemVerilog designs. Altera's Quartus II tool chain for FPGAs can also synthesize some SystemVerilog constructs. HDL designs are synthesized to netlist files. Most proprietary and third party vendor tools can work with the common EDIF netlists. Netlists contain the instances, nets and attributes required to describe the design. Synthesis switches can help the third party vendor tools optimize the netlists for a particular device architecture. The Xilinx Native Generic Circuit (NGC) netlist uses the UNISIM library primitives meant for behavioral simulation of the design. Functional simulation of the design is done at this stage to verify the intended behavior.

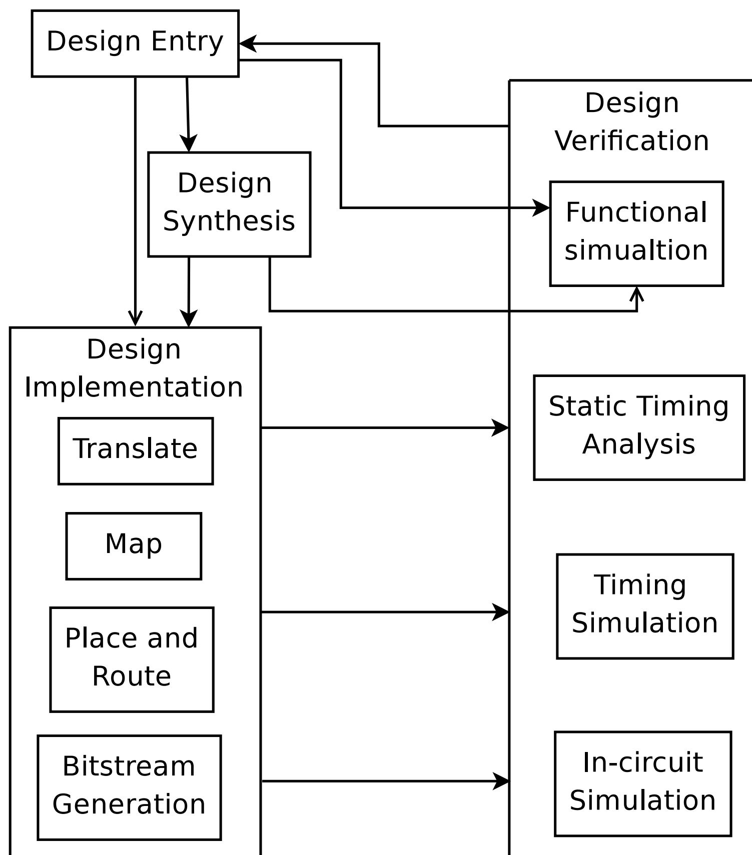

Figure 2.3: Xilinx standard design flow

2.2.2 Design Implementation

After design entry is completed, the netlist generated is implemented to produce a bitstream file which is used to configure the FPGA device. The netlist can also include other third party proprietary Intellectual Property (IP) cores. In the Translate phase of the Xilinx design flow, the netlist file (NGC/EDIF) is converted into an intermediate Native Generic Database (NGD) file. The NGD file contains a logical description of the original netlist reduced to Xilinx primitives and a description of the netlist hierarchy. This process is accompanied by a logical DRC check to verify if the symbols in the NGD file are valid Xilinx primitives. The NGD file is based on the SIMPRIM library used for timing simulation. It contains approximate details about the switching delays in the circuit.

In the MAP phase of the Xilinx flow, the SIMPRIM primitives in the NGD netlists are mapped to the basic Xilinx device primitives [4]. Xilinx Hard Macro Files (NMC) files are used by MAP when the NGD netlist contains macro definitions. The output of the MAP phase is a Native Circuit Design (NCD) file. It contains precise information on the switching delays in the circuit. MAP also produces the Physical Constraint File (PCF) containing any constraints specified during the design entry phase. When a single driver drives multiple loads, the delay on the nets increase. To avoid such scenarios, MAP may perform logic replication to allow a driver to drive only a single load.

The Xilinx device primitives in the NCD file are placed and routed using the Xilinx PAR program. Placement is the process of assigning specific sites to the device primitives in the NCD file from MAP and routing is the process of connecting the primitives using the lines available in the routing channel. Routing resources in FPGAs are limited and hence to achieve optimal timing, placement has to consider various parameters like available routing resources and the length of connections. PAR can be run either in the timing-driven mode

or the automatic timespecing mode. In the timing-driven mode, PAR uses Xilinx timing analysis tool to ensure timing. PAR tries to maximize the clock frequency in the automatic timespecing mode. This is similar to running PAR multiple times in the timing-driven mode with progressively increasing frequencies. PAR also provides options to iteratively place-and-route until the design meets all the timing constraints. In each iteration, placement uses a different random seed called the cost table. It is similar to having placement perform a multi-pass search thorough a range of Quality of Results (QoR). The output NCD file from PAR is used by the Xilinx BitGen program to create a bitstream that can be used to program the FPGAs [5].

2.2.3 Design Verification

Design Verification is the process of ensuring that the created design conforms to the specifications. The Xilinx design flow supports simulation based verification at various stages of the flow and through in-circuit verification where the operation of the device is verified by downloading the bitstream to the FPGA. Simulation based verification can be performed at five different stages with each stage having different capabilities. The RTL simulation facilitates verification of the design at the system level. Verification at the RTL level is not architecture-specific unless the design contains instantiated UNISIM or CORE Generator components.

In post-synthesis verification, the netlist is simulated and the results are compared with the RTL model. The netlist contains the UNISIM primitives and the Core Generator components if any. If the third party vendor tools use their proprietary simulation modules, the netlist based verification is not possible with the Xilinx flow. Hence, an optional Post-NGDBuild simulation can be performed on the netlist with the SIMPRIM primitives. Post-MAP simula-

tion includes block delays and can be used to test if the design meets the timing requirements before PAR. Simulation upto this stage can verify only the functionality of the design and timing to some extent. The placed and routed design can be simulated to provide accurate timing information for the entire design. The simulation model accounts for both the block and routing delays and is used to verify the behavior of the design with all the timing constraints.

2.3 Modular Design Flow

The standard design flow has certain shortcomings when used for large designs. The tools run longer to achieve timing closure for large designs if strict timing constraints are specified. Also, in the standard flow, the whole design has to be synthesized and implemented for a small change in the HDL. In the software domain, this is analogous to building all the independent libraries of an application for a change in one of its libraries. In order to overcome these shortcomings, Xilinx introduced the *Modular Design Flow* (MDF) shown in Figure 2.4 in which a team of designers could work on independent parts of a project called *modules*. MDF helps in achieving independent timing closure of each module and improves turnaround time [6]. It is more suited for large designs that can be partitioned into independent modules where the critical path is contained within the module.

Using the MDF, design entry and synthesis of the modules can be completed by the designers in parallel. Implementation of these modules require the initial budgeting to be completed in which constraints are added to the top-level design. Constraints are added to restrict the implementation of modules to rectangular regions on the target FPGA. The initial budgeting phase also positions the input and output ports for the modules. The module ports are connected to pseudo-logic during implementation. When the design has interconnections

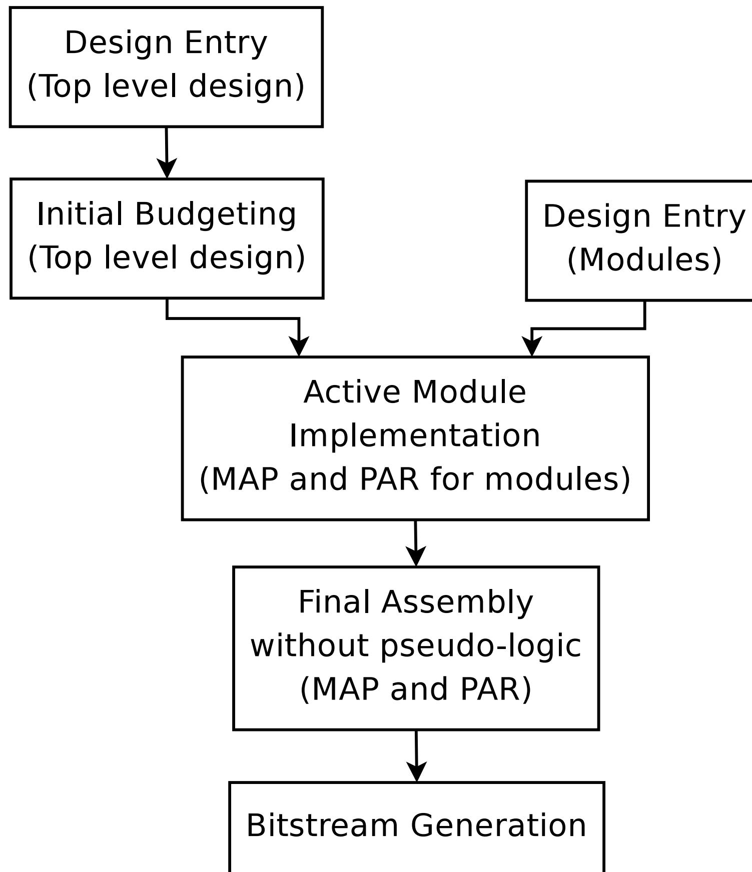

Figure 2.4: Xilinx MDF

between modules, pseudo-logic helps in place-and-route of a module independent of the placement of the other module to which it is connected. The pseudo-logic is removed during the final assembly of the design. Proper positioning of the modules, module ports and the top-level logic is important to achieve timing closure for the whole design. Once the Initial budgeting phase is completed, the designers can independently perform the active module implementation phase in which each module is mapped, placed and routed. During final assembly, the tools copy the place-and-route information of individual modules to preserve the timing performance of each module.

2.4 Improvements in Design Flow

The MAP and PAR phases of the design flow can run longer for large designs with timing constraints that are difficult to meet. In order to reduce the time required by the tools to achieve timing closure, Xilinx introduced the *SmartGuide* [7] technology. SmartGuide is useful towards the end of the design cycle [8] when all the timing constraints are met and small changes are made to the design. SmartGuide enables MAP and PAR to use the most recently created NCD file as a guide to implement the design [9]. As a result, the time required for re-implementation of the design is reduced. Synthesis and implementation of the design is performed for the whole design, and changes to all the modules require complete re-implementation of the design.

Xilinx introduced *Partitions* [10] aiming higher design re-use and reduced tool runtimes. A partition is a source instance that is marked for re-use. Partitions could be a HDL source, schematic or an EDIF source instance. Partitions are implemented without affecting one another and hence un-affected partitions can be copy-pasted into the new implementation. The preserve property of the partition determines which implementation data will be preserved. When the preserve property is set to synthesis, only the instance netlist is preserved and the module need not be synthesized again for changes in other modules. If the preserve property is set to placement, the netlist and placement of the instance is preserved. A default preserve property set to routing preserves the netlist, placement and routing information of the instance is preserved.

2.5 Partial Reconfiguration

FPGAs can be reconfigured while in operation to change the functionality of one or more modules. This may be used to time-share the FPGA resources between different designs. Partial Reconfiguration [11] is the ability to update a portion of the FPGA with a different configuration when the device is in operation. Certain designs that require larger FPGAs can be implemented on smaller ones by time-sharing the resources between its modules. PR also helps in reducing the configuration time for FPGAs when a new design that is implemented differs only slightly from the existing design. Xilinx supports two different styles of PR flow. The module-based PR flow is used for designs that require time-sharing of FPGA resources between its modules. Difference-based PR flow reduces the size of the configuration file when the design changes by a small amount. The small size of the configuration file helps in reducing the time required to configure the FPGA with the new design.

The Module-based PR design flow shown in Figure 2.5 is based on the Xilinx Modular Design methodology. The design entry and synthesis of the PR modules can occur in parallel. Implementation of the modules require the initial budgeting phase to be completed in which rectangular areas are specified for implementing the PR modules along with other global constraints. These rectangular areas called PR regions are time-shared between the PR modules. The initial bitstream loaded into the FPGA will be a complete design. FPGA can then be updated with a partial bitstream that replaces a module in the PR region with a new module.

The module-based PR flow has certain shortcomings related to size and implementation of the PR region. The Early Access Partial Reconfiguration (EAPR) flow overcomes those disadvantages [11]. The Xilinx PR design flow uses physical ports called Bus Macros (BMs) for communications with a PR region. BMs provide slice based anchor points for the modules

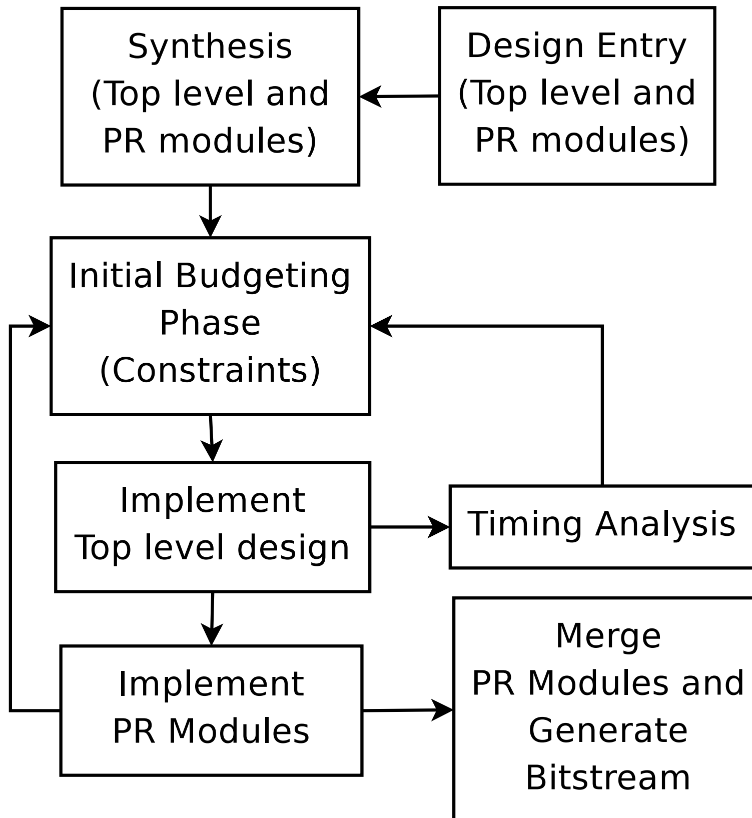

Figure 2.5: Xilinx PR flow

to lock the routing resources to its ports so that a PR module can be replaced by a different PR module while the device is in operation. Unlike pseudo-logic used in modular design flow, bus macros are not removed during final assembly of design to create the bitstream.

2.6 Need for a New Design Flow

The Xilinx standard design flow explained in Section 2.2 treats the implementation of a design on the FPGA as a global optimization problem. Any design change, even if confined to a small portion requires complete re-implementation of the design. Instead, by using a modular design flow explained in Section 2.3, achieving timing closure for a module can be performed independently of other modules when the design changes. The pseudo-logic

inserted in the design should be removed to generate the bitstream. Also, proper placement of modules, module ports and global logic is required to achieve timing closure. The placement process is manual and becomes tedious when multiple iterations are required to achieve timing closure.

Partitions explained in Section 2.4 can handle design changes efficiently by using the implementation files from the previous tool runs. As a copy-paste approach is used, use of the placed-and-routed partitions from previous runs reduce the time required for generating a new bitstream. In case of a timing failure in a partition, the preserve property of the neighboring modules have to be relaxed. This causes re-implementation of the neighboring modules also. MAP and PAR is done for the complete design using the implementation data from the previous runs.

The proposed DMD flow incorporates the principles of both the modular design flow and partitions. Design entry and the initial budgeting phase happens as in the case of modular design flow. In the implementation phase, timing closure is achieved independently for the modules. The final bitstream is generated directly from the implemented NCD files of the modules as there is no need for removal of pseudo-logic. In case of a design change in one of the modules, the changed module alone is synthesized and implemented. As the synthesis and implementation of modules can happen independently and concurrently, it is faster compared to partitions.

Chapter 3

System Overview

The DMD flow is an improvement over the Xilinx MDF and partitions. The design entry and synthesis phase of the DMD flow is similar to that of MDF. The DMD flow leverages certain features offered by the Xilinx EAPR flow to accelerate the design implementation phase at the expense of some circuit area and speed. Incremental changes are implemented faster compared to the MDF as the bitstream is generated directly from the module implementation files. The initial budgeting phase is automated and only the optional timing constraints are added manually by the designer. In comparison to partitions, the complete re-implementation of the design is faster as the modules are implemented by independent and concurrent invocations of the standard Xilinx tools.

3.1 Xilinx EAPR Flow

Run-Time Reconfiguration (RTR) is the capability to change the functional configuration of a part of the system when it is in operation. Modern Xilinx FPGAs support active PR, a feature that allows modules to be swapped on-the-fly when the device is in operation. Using

active PR, the FPGA can be updated with a partial bitstream that can replace an existing PR module with a new module. The active PR is a hardware-based approach to support RTR.

The PR modules are implemented in rectangular regions called Partially Reconfigurable Regions (PRRs). The modular-design-based PR flow requires the PRR to occupy the entire height of the device, whereas the EAPR flow allows PR regions of any rectangular size. Multiple Partially Reconfigurable Modules (PRMs) that are mutually exclusive are usually implemented in the same PRR. The static region in the FPGA does not belong to any PRR and remains operational during PR. The PRMs can be swapped on-the-fly in and out of the PRR without affecting the operation of the rest of the device. Figure 3.1 shows a sample FPGA layout with the static region, a single PRR `A` and three PRMs `A1, A2` and `A3` implemented in the PRR.

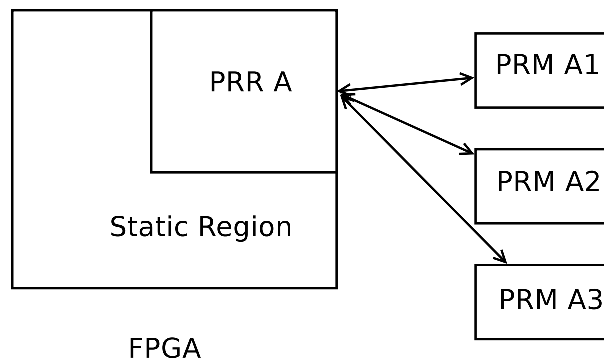

Figure 3.1: Various regions in a PR design

Figure 3.2 illustrates the EAPR flow for a design with two PRMs. The EAPR flow recommends the use of top-level logic with the black-box instantiations of the lower-level modules. The top-level HDL file contains the instantiations for PR modules, static logic, clock primitives, and the bus macros. Each PRR in the design will have a corresponding black-box module instantiation in the top-level logic. During module implementation, the black-box instantiation can be replaced with any of the PRMs, as they carry the same entity name and

are pin-compatible. The top-level logic, the static logic, and PR modules are synthesized separately. The 'Keep Hierarchy' option instructs the Xilinx Synthesis tool (XST) not to perform any global optimization on the design.

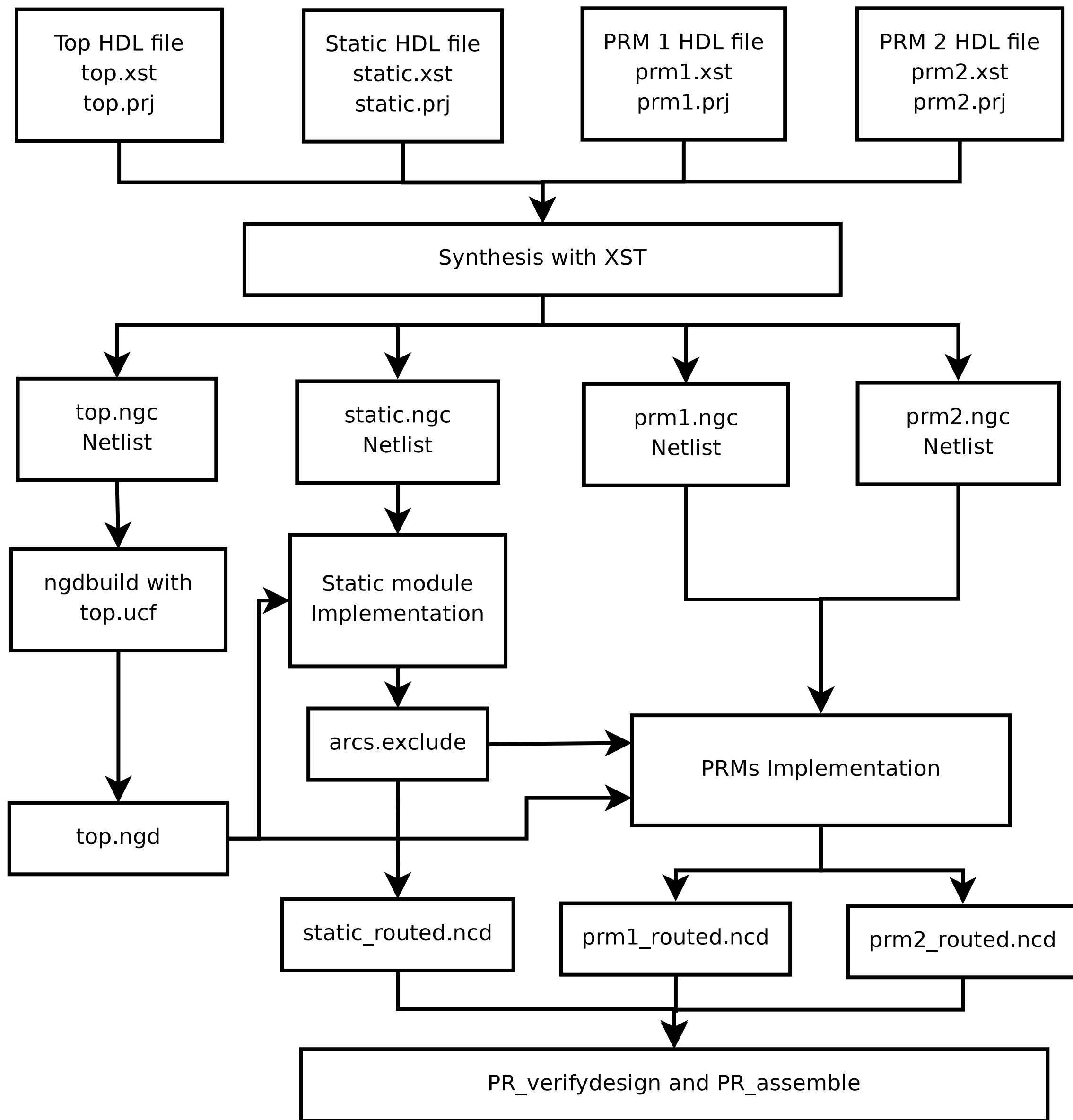

Figure 3.2: PR implementation flow

In the initial budgeting phase, the FPGA is floorplanned to allocate rectangular regions for the PRRs. The PRR placement should take into account the resource requirements of

PRMs such as slices, BRAMs, DSP slices, and on-chip processor cores. Once the placement of PRRs are determined, bus macros are placed on its boundaries by specifying the placement constraints. The User Constraints File (UCF) is created to specify the placement constraints for the PRRs and bus macros. The bus macros are hard macros that are pre-placed and pre-routed specific to an architecture. The Location (LOC) constraint is used to fix the bus macro to a slice such that it straddles the boundary between the PRR and the base design. The `Area Group` (AG) is specified in the UCF file for each PRR to constrain its location to a range of slices and other resources encompassed by those slices. Since the base design can use any of the FPGA resources outside the PRRs, no range values are specified for the base design AG constraint. Proper placement of the PRRs and bus macros is necessary to satisfy the timing constraints of the design. The UCF file is used in the optional non-PR implementation step where the PRMs are implemented as static modules in the AGs specified. As the non-PR implementation uses the same constraints as the final PR design, it can be used to verify if the PRMs can be implemented in the PRR.

The Xilinx PlanAhead tool can also be used for creating the UCF with the AG constraints for the PRRs and LOC constraints for the bus macros [12]. The PlanAhead tool is a complete flow management tool for Xilinx FPGA designs using the standard Xilinx tools. It provides a Graphical User Interface (GUI) as shown in Figure 3.3 to help in design exploration and experimentation. For example, the AG constraints for the PRRs can be specified by creating a Physical block (Pblock) in the floorplanner layout. The tool shows the resources required for the PRM and the resources available with the Pblock so that the designer can change the PRR as required.

The top NGD file generated from the top module netlist, is used to implement the base design. The PR modules appear as unexpanded blocks in the base design NGD file. In order to achieve good timing performance, the base design can use the routing resources

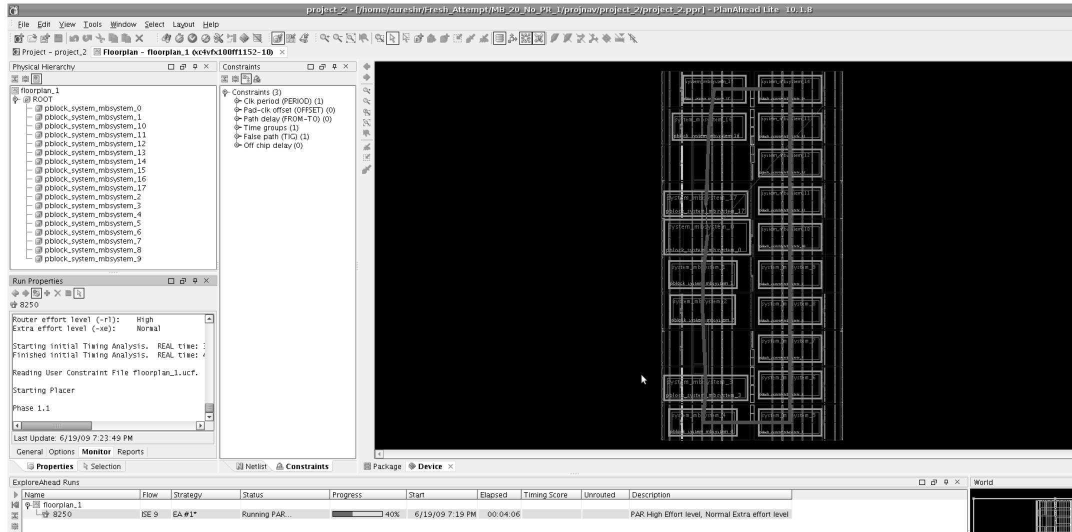

Figure 3.3: PlanAhead screenshot

inside the PRMs [13]. Hence, the implementation of the PRMs can happen only after the base design is implemented. The `arcs.exclude` file contains the list of routing resources used by the base design inside the PRRs and the same `arcs.exclude` file can be used for future PRM implementations as long as there is no change to static logic or the PRM ports. Also, as the PRMs are implemented inside the AG specified for the PRR in the UCF file, the implementation NCD files of the PRMs can be directly used for generating the bitstream. The PR_verifydesign and PR_assemble tools from the EAPR flow generates the partial bitstreams for each PRM and the complete bitstream to program the FPGA.

3.2 PATIS - A Modified PR Flow

The EAPR flow adds additional complexity to the design compared to the modular flow. Further, PR designs cannot be simulated and testing has to be done directly on the hardware where visibility is limited. The DMD flow, instead recasts RTR as a design-time methodology

for implementing static designs to improve design productivity. The EAPR flow contains the necessary tools for creating a methodology that supports incremental build for designs. The DMD flow uses a divide-and-conquer strategy to implement the design in order to generate the bitstream faster at the expense of some chip area and speed. Modules in the design are implemented in separate PRRs to allow independent and parallel implementation of the modules with the standard Xilinx tools. Timing closure is achieved independently for the modules in the design and a change in one of the modules requires re-implementation of only that module. Using the EAPR flow, the final bitstream is generated by directly assembling the module implementation files.

Design productivity is addressed in the DMD flow by creating a partial module-producing, automatic, timing-aware, incremental and speculative floorplanner. The PATIS floorplanner creates the UCF file with regions assigned for PRRs and location constraints for bus macros [14]. The DMD flow can be used during the design development phase to accelerate design implementation. Modules are implemented in parallel using the UCF file. Timing failure or lack of resources for implementing a module necessitates changes to the floorplan. Unlike change to a particular module, changes to floorplan requires a complete re-implementation of all the modules with the new floorplan.

The *viscosity* and *fickleness* attributes [14] for a module can help in floorplanning the design so that future changes can be accommodated into the design efficiently. The viscosity attribute is a measure of the likelihood of change to a module. PATIS uses strategies such as allocating extra resources and grouping the fast changing, low-viscous modules to avoid floorplan revisions. PATIS uses the fickleness attribute to identify modules that will be constrained to particular regions on the FPGA. The fickle modules require more effort to meet timing and sometimes require special macros on the FPGA. The high-level DMD flow, is shown in Figure 3.4.

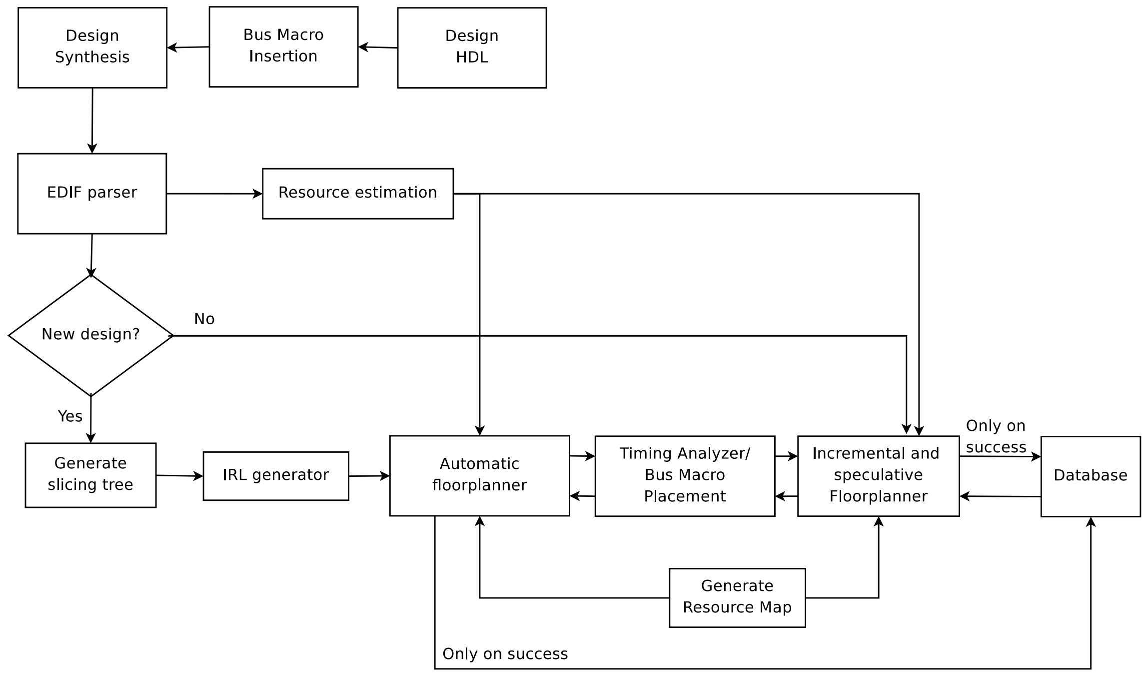

Figure 3.4: Overall DMD flow

In order to keep the design entry process consistent with the Xilinx Standard flow, the floorplanning phase, bus macro insertion and placement steps are automated in the DMD flow. Top-level design HDL is parsed to find the module ports and the associated signals. Module port information is used to update the top-level HDL file with the bus macro instantiations. The Xilinx PlanAhead tool helps in resource estimation for the modules in the design. PATIS operates in one of the two following modes:

1. *New floorplan creation*: An automatic floorplanner is used to create a new floorplan for a design. Bus macros are placed in a manner such that the floorplan meets timing. Created floorplans can accommodate small changes to the modules. The floorplan generated by the automatic floorplanner is stored in the database for use by the incremental floorplanner.

2. *Floorplan reuse*: The incremental floorplanner creates a new floorplan by making incremental changes to an existing floorplan in the database. The speculative floorplanner

anticipates changes to modules using the viscosity and fickleness attributes, and speculative generates floorplans for different scenarios. When the incremental floorplanner is not able to accommodate the resource requirements of the design, a new floorplan is created by the automatic floorplanner.

3.2.1 Automatic Floorplanner

The PATIS automatic floorplanner generates new floorplans for a design. The design HDL is synthesized to an EDIF netlist. When the XST tool is used for synthesis, the `ngc2edif` tool can be used for creating the EDIF netlist from the NGC file. As the synthesis is hierarchical, a recursive descent parser created for the EDIF grammar can extract the module names and their interconnections. The automatic floorplanner uses this information to create a graph with modules as vertices and their connections as the edges between the vertices. In order to represent the multiple fanouts for a signal that might exist in the design, the graph created is a hypergraph, where the edges can connect any number of vertices. The resource estimate in terms of the number of logic slices, BRAMs, and DSP slices for each module is obtained by use of the PlanAhead tool.

The automatic floorplanner tries to place the connected modules in PRRs close to one another to minimize routing delays. PATIS uses the slicing tree approach to generate floorplans from the hypergraph for the design. The floorplan is built bottom-up from the slicing tree. Leaf nodes of the slicing tree are the resource requirement rectangles which are the PRRs that will be placed in the FPGA layout. Intermediate nodes in the slicing tree correspond to meta-rectangles in which its leaves will be placed. The slicing tree is obtained by partitioning the hypergraph recursively using the hMetis tool [15].

In each step, the hypergraph has to be two-way partitioned such that minimum edges are

cut to create the partitions with approximately same number of vertices. In order to improve the partitioning quality, the hMetis tool uses a multi-level approach to partitioning the hypergraph. Edges and vertices in the original hypergraph are collapsed during the *coarsening* stage to create a smaller hypergraph. The reduced hypergraph is partioned by hMetis such that, a good partitioning of the smaller hypergraph is not significantly worse than the partitioning of the original hypergraph. The hMetis tool also has a refining phase in which the vertices along the partioning boundary are given focus to improve the quality of results. Once the partitioning is completed, hMetis *uncoarsens* the smaller hypergraph to create partitions of the original hypergraph. The number of vertices in the partitions created by hMetis might differ and hence a hMetis wrapper balances the partitions by moving the least-connected vertices across the boundary.

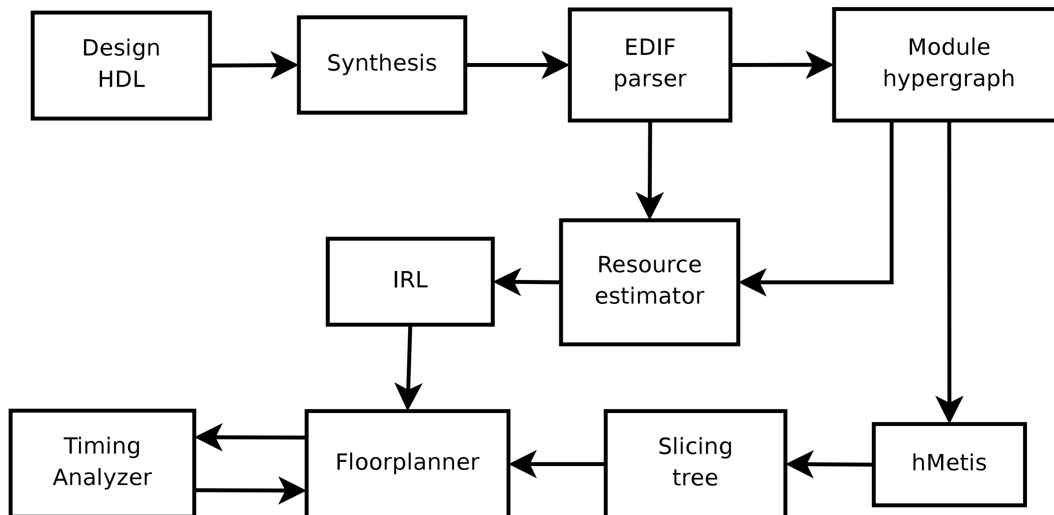

Figure 3.5: Automatic floorplanner flow

The PATIS automatic floorplanner creates Irreducible Realization Lists (IRLs), a list of all possible implementations of a module on the FPGA such that its resource requirements are satisfied. Each implementation is defined by a 4-element vector $\langle x_0, y_0, x_1, y_1 \rangle$, where (x_0, y_0) denotes the bottom-left coordinate and (x_1, y_1) the top-right coordinate. Implementations in the list cannot be reduced to smaller sized implementations having the same bottom-left

coordinate (x_0, y_0). The floorplan for the design is created from the slicing tree and the IRL in a bottom-up manner. Adjacent leaf nodes in the slicing tree are the modules to be placed adjacent in the floorplan.

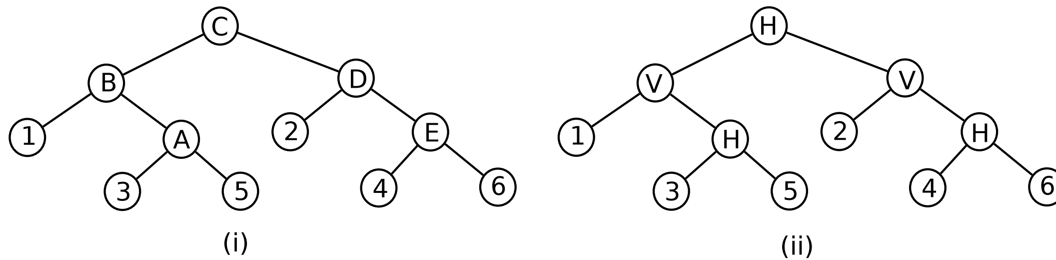

Figure 3.6: Slicing tree with alternate horizontal and vertical cut directions

The PATIS automatic floorplanner creates the AG constraints for modules in the depth-first order in the slicing tree. Slicing tree helps in determining the adjacent modules, but it does not give the relative position of the modules in the layout. Cut directions in the floorplan are used to determine the relative placement of adjacent modules. The automatic floorplanner assigns vertical and horizontal cut directions alternately as shown in Figure 3.6 to obtain a uniform floorplan. Cut directions are changed if required.

When the left child node is placed, a module implementation is chosen such that its aspect ratio is close to unity. The child node could be a leaf node or a clustered node which is a collection of its leaf nodes. In case of a horizontal cut, the width of the right child is chosen close to the width of the left child so that extra white spaces are reduced in the layout. Similarly, in case of a vertical cut, the height of the right child is chosen close to the height of the left child node. The cut direction is modified when the module has no possible implementation for the given (x_0, y_0) co-ordinate. The floorplan generated is static timing verified and added to the database. In case PATIS is not able to place the bus macros to meet timing in the created floorplan, a new floorplan is created.

3.2.2 Incremental Floorplanner

PATIS re-uses the layout created by the automatic floorplanner, so that a new floorplan need not be generated for every design change. The speculative floorplanner generates a set of floorplans anticipating the changes to modules using the viscosity and fickleness attributes. The set of feasible floorplans created by PATIS is stored in a database. When a new floorplan is required, the items in the database are checked to find a floorplan that suits the new resource requirements.

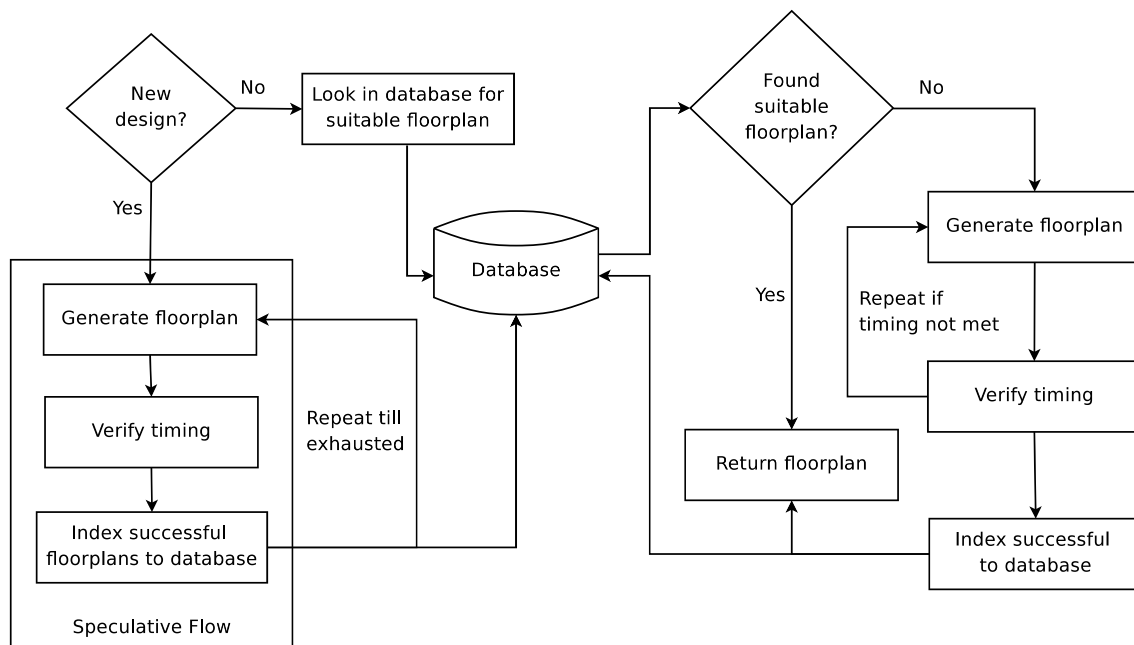

Figure 3.7: Incremental speculative floorplanner high-level flow

The speculative floorplanner creates multiple variants of the floorplan from the floorplan created by the automatic floorplanner. Modules are floorplanned in the order of their increasing viscosities as the low-viscosity modules are susceptible to change frequently. Fickle modules in the design are frozen in place and are not speculatively floorplanned by PATIS. The speculative floorplanner explores the design space with reasonable aspect ratios for the modules. Similar to the automatic floorplanner, the incremental floorplanner uses the EDIF

parser and the resource estimator to create the floorplans.

When the resource requirements for a module changes, PATIS searches the database for a suitable floorplan that meets the resource requirements. The incremental floorplanner is used when none of the floorplans in the database can meet the resource requirements, and attempts to modify the existing floorplan to generate a new floorplan that fits the new resource requirements. In case the incremental floorplanner is not able to generate a feasible floorplan, PATIS uses the automatic floorplanner to create a new floorplan.

The incremental and speculative floorplanners use two algorithms: *White Space Occupation* and *Neighbor Displacement*. The white space occupation algorithm looks for the unused resources around a module requiring resources. The algorithm alternately expands the horizontal and vertical boundaries of a module until the resource requirements are met. The neighbor displacement algorithm is used when the free resources in the vicinity of the module requiring resources are exhausted. The algorithm performs translation and shrink operations on the neighbor modules such that the modules respect their resource requirements. When the algorithm is no longer able to perform translate or shrink operations in any of the directions, the current floorplan can no longer be used and a new floorplan is generated using the PATIS automatic floorplanner.

3.2.3 Bus Macro Insertion and Placement

The EAPR flow requires the use of bus macros for communication between the static region and the PR modules. Instantiation of bus macros in the HDL source can be complicated and tedious when the module ports change in the design. In order to maintain consistency with the design entry phase of the standard flow, the DMD flow automates the instantiation of the bus macros in the top-level HDL. Any change in module ports are automatically identified

and appropriate changes are made to the bus macro instantiations. PATIS also automatically assigns LOC constraints to the bus macros. The bus macro placement tool aims to improve design timing by reducing the delays on intermodule nets. Bus macro insertion is discussed in detail in Chapter 4.

3.3 Debug Support

Debug activities in DMD are handled by two mechanisms: Low-Level Debugging (LLD) and High-Level Validation (HLV). Much like simulators and the embedded logic analyzer cores provided by FPGA vendors, LLD handles data at the bit-level. However, LLD is not based on capture methodologies which rely heavily on embedded memory to record signal activity, but instead is based on conditional breakpoints such as those used in software development to halt the design and enable it to be stepped and analyzed using a microprocessor. Breakpoint logic is implemented in a designated reconfigurable region where it can be modified and rapidly re-integrated without altering the rest of the design. Register state is retrieved through an Internal Configuration Access Port (ICAP), a Xilinx proprietary interface allowing direct access to register state. LLD is illustrated in Figure 3.8.

HLV, as shown in Figure 3.9, abstracts away low-level implementation details by creating a framework for validating individual modules against a functional model written in a high-level language implementation. HLV can compare hardware implementation against software behavioral models in the same manner that software developers apply unit tests. The DMD debug system consists of a microprocessor and programmable debug controller (PDC) on the FPGA to provide a platform for user interaction. By reserving one of the reconfigurable regions for debug logic, the PDC allows breakpoints, assertions and signal constraints to be interactively specified and modified with out need to re-implement the entire design. Design

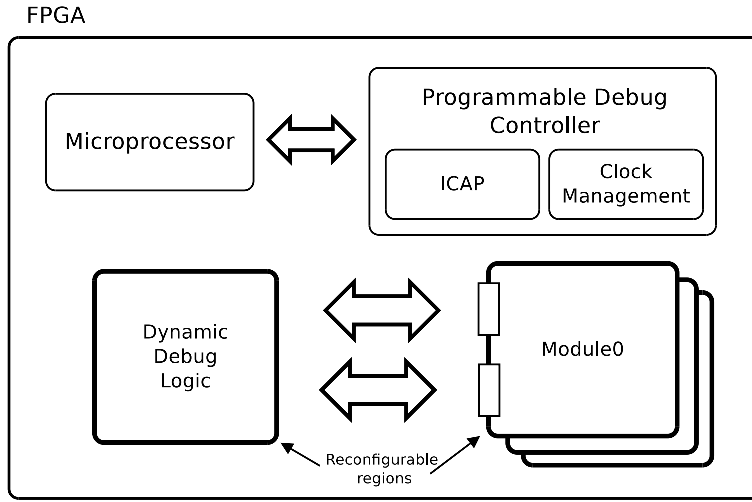

Figure 3.8: LLD interaction

validation can be automated in the same manner as used in software unit-testing, where individual components are tested against known conditions.

3.4 Experimental Verification — Proof of Concept

A proof-of-concept design was developed to verify the feasibility of the DMD flow. The design contains a DMA engine that handles data from the PCI interface and a grayscale filter that takes a color pixel as input and produces the grayscale equivalent of the pixel. The static design is implemented using the standard Xilinx tool flow to create the bitstream. An equivalent PR design is created with one PRR that is used to implement the grayscale filter. For implementing the PR design, the floorplan and the LOC constraints for the bus macros are created manually. The design was implemented using Xilinx ISE 9.2.04i with the PR-12 patch targeting a Virtex-4 FX100-ff1517 FPGA [16] device. The design was implemented on a Intel Core i7-920 2.66 GHz Desktop with 12 GB of memory running Ubuntu 8.10 on the 2.6.27 generic Linux kernel. The execution times for implementing the static design, top module and the grayscale filter is shown in Table 3.1.

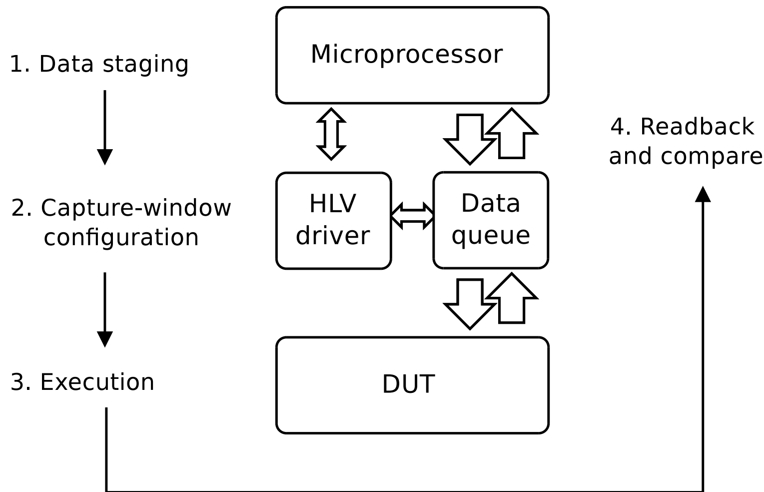

Figure 3.9: HLV flow block

Table 3.1: Xilinx tool flow execution times

	Execution time (seconds)		
		PR design	
Xilinx process	**Static design**	**Top module**	**Grayscale filter**
Synthesis	-	50	12
Ngdbuild	-	6.74	6.75
MAP	-	23	24.6
PAR	-	59	45
Bitstream generation	-	-	121.32
Total time	289	138.74	209.67

The following observations are made from the results of the proof-of-cost experiment.

1. Using the DMD flow, the total time required for generating the bitstream is the sum of the time taken for implementing the top-level design and the longest time among the PRM's implementation times. Hence, the bitstream for a large modular design can be generated rapidly using the DMD flow compared to the standard flow, MDF and partitions.

2. When the design is implemented the first time, the standard flow takes 289s to gen-

erate the bitstream. Using the DMD flow, 139s is required for implementation of the top-level HDL and 209s for implementing the grayscale filter. As the design size and the number of modules increase, the standard flow will take longer to generate the bitstream compared to the DMD flow where the PRMs can be implemented concurrently.

3. The time taken for generating the bitstream using the DMD flow for incremental changes in the grayscale filter is 209s compared to 289s using the standard design flow. Thus, the DMD flow can generate the bitstream for incremental changes faster compared to the standard flow.

4. With a good modular design, the DMD flow can generate bitstreams faster for incremental changes and complete re-implementation compared to the standard flow, the MDF, and partitions.

Usually PAR requires more time for completion compared to other steps in the Xilinx flow. As the PAR for the modules in the design occur concurrently with DMD, timing closure for the entire design is achieved faster. In case of timing failures in a module, the tools concentrate on optimizing an individual module and hence the overhead associated in dealing with the entire design is avoided. With a good floorplan and bus macro placement, bitstreams can be generated faster using the DMD flow.

Chapter 4

Bus Macro Insertion and Placement

Bus macros are physical ports used for communication between the static region and the PRRs. In the EAPR flow, instantiation and placement of bus macros is a manual process. The DMD flow automates the insertion and placement of bus macros. This chapter begins with an overview of bus macros and discusses the functional flow of the automatic bus macro placement and insertion process. The implementation flow for placement and insertion of bus macros is also described along with the strategies used for automatic placement.

4.1 Bus Macros

Bus macros are pre-placed and pre-routed hard macros used by the EAPR flow to provide route-locking between the PRRs and the base design. The bus macros provide slice-based anchor points for the module ports so that the modules can be implemented independently and concurrently. With the exception of the clock signal, all communication between base design and the PRR should be through bus macros. The bus macros straddle the boundary between the PRR and the base design. Hence, when the PRMs are pin-compatible, the

existing bus macros are reused for the new PRM module ports.

In the Xilinx MDF, pseudo-logic is used only for communication between two modules and is not used for communication between a module and the base design. As a result, re-implementation with all the modules together is required to build the whole bitstream. In the EAPR flow, the PRRs cannot use any resources outside of its module boundary because bus macros are used for all signals and the static region that might use the routing resources inside a PRR is implemented before the PRMs implementation stage. As a result, the implementation of the PRMs can be directly copied to create the bitstream.

The bus macros are architecture specific and the Xilinx EAPR flow provides the bus macros for Virtex-II, Virtex-4 and Virtex-5 architectures. The main features related to bus macros are:

1. *Signal direction* — The Virtex-4 bus macros are uni-directional. They can be left-to-right, right-to-left, top-to-bottom or bottom-to-top bus macros. The directionality of a bus macro to be used is dependent on whether its used for input or output and the boundary of the PRR on which the macro is placed.

2. *Physical width* — The narrow bus macros are two CLBs wide and the wide bus macros are four CLBs wide. The width refers to the physical width of bus macros and both the narrow and wide bus macros provide eight bits of data. The wide bus macros are similar in structure to the narrow bus macros. In order to support straddling of three wide bus macros, each of the wide bus macros have unused CLBs in their structure. The wide bus macros can be straddled along a single CLB row or column to support 24-bit bus macro signals.

3. *Synchronous and asynchronous macros* — The synchronous bus macros register the data and provide superior timing performance. The data through the asynchronous

bus macros are not registered.

Figure 4.1 shows the screenshot of the `fpga_editor` tool with a left-to-right narrow bus macro highlighted in red. The macro is pre-placed and pre-routed occupying two CLBs. The four slices on the left side of the macro belongs to a single CLB and the four slices on the right side of the macro belongs to another CLB. When the macro is placed straddling the boundary of the PR region, it consumes one CLB from the PR region and another CLB from the static region. Similarly Figure 4.2 shows the screenshot of the *fpga_editor* tool with a top-to-bottom narrow bus macro.

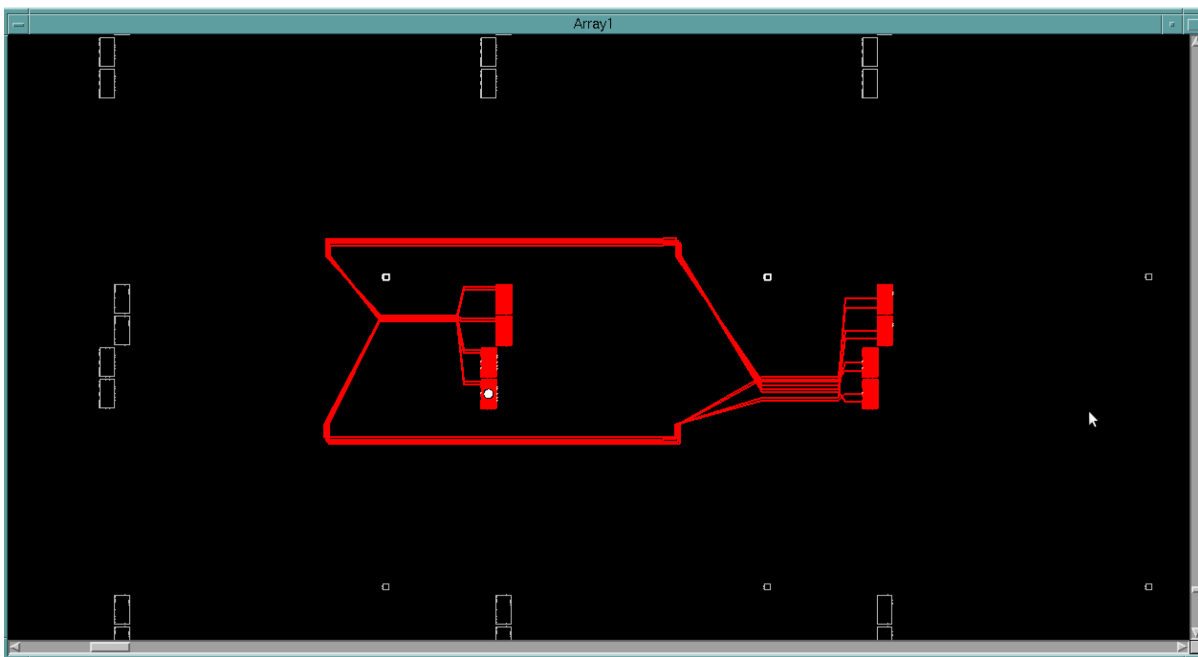

Figure 4.1: Screenshot of fpga_editor with left-to-right bus macro

The bus macros are instantiated in the top-level design file. Each bus macro provides eight input ports and eight output ports. The inputs to a PRM from the static region are connected to the input port of the bus macro and the output port of the bus macro is connected to the PRM input port. Similarly, the outputs from a PRM to the static region are connected to the input port of the bus macro and the output port of the bus macro is connected to

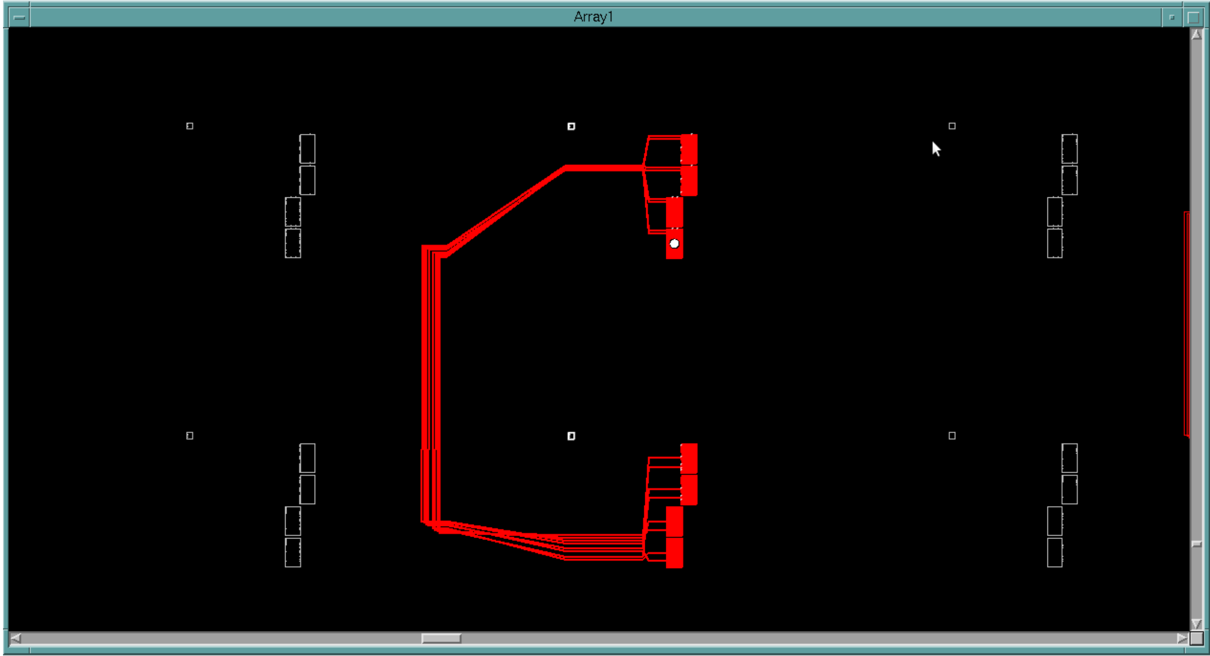

Figure 4.2: Screenshot of fpga_editor with top-to-bottom bus macro

the signals in the static region. The inputs to a PR module through bus macros on its left boundary requires left-to-right bus macros and the inputs through the right boundary requires the use of right-to left bus macros as shown in Figure 4.3. Similarly, the outputs from a PR module through its left boundary requires the use of right-to-left bus macros and the outputs through the right boundary requires the use of left-to-right bus macros as shown in Figure 4.4

4.1.1 Best Practices for Bus Macro Placement

The type of bus macro to be instantiated in the top-level HDL is based on the placement of the bus macro and the direction of data. The effectiveness of a placement can be identified only after a place-and-route of the complete design. Hence, it becomes necessary to identify certain best principles that help in the placement of bus macros. The following is recommended by the FPGA vendor for a PR design:

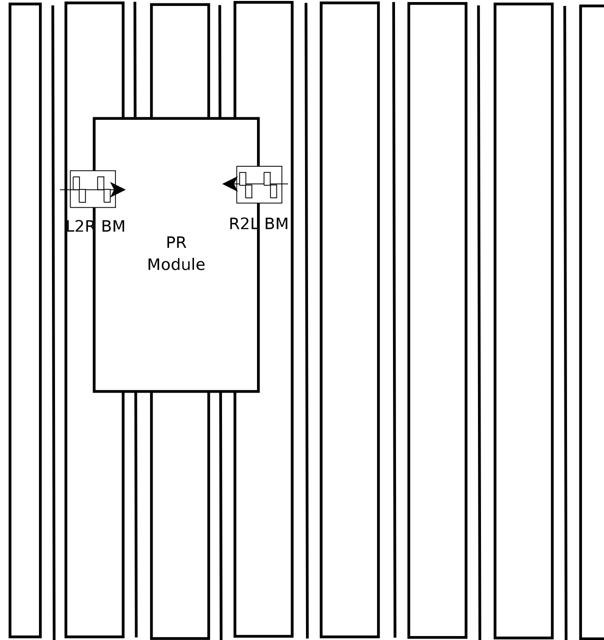

Figure 4.3: PR region with input bus macros

1. The carry-chain logic prefers to grow vertically in the Xilinx tool flow. Hence, it is preferable to grow the PR regions vertically as the chain can use additional logic in the adjacent vertical chain. This principle has no direct implication on the placement of bus macros for a PR module.

2. The input and output bus macros are to be placed on opposite boundaries of a PRR. Placing the input and output bus close to one another increases the chance of zigzag routing paths in the PRM.

3. Grouping signals by bus or by interface helps PAR to optimize the placement of time-critical logic as a group.

The best practices suggested are more suitable for PR in Virtex-II devices where the PRRs span the entire height of the device. In the case of Virtex-4 devices, the PRRs can be of any rectangular size. Also, as the number of PRRs and interconnections between the PRRs

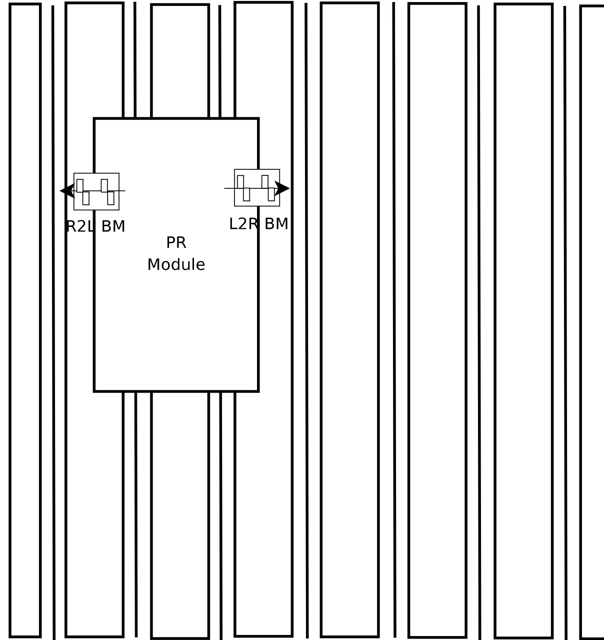

Figure 4.4: PR region with output bus macros

increase, the placement of bus macros even using the best practices yields sub-optimal results. Additionally, as the bus macros support only eight-bit wide data, a change in module ports can cause changes to multiple bus macro instantiations. Manually changing the bus macro instantiations is time-consuming. Automating the instantiation of bus macros in the top-level HDL file helps to avoid mistakes that may happen when large changes are made to the HDL file.

The placement of bus macros is critical to achieve timing closure for the entire design. The Xilinx 9.2 PR flow does not support automatic placement of bus macros, as the effectiveness of placement can be determined only after complete implementation of the design for all the PRM combinations. The design space also grows exponentially with the increase in number of bus macros, the number of PRRs and the size of the PRRs. Hence a trial-and-error approach is not a feasible strategy to find a suitable placement for the bus macros.

4.2 Related work

The work by Carver et al. [17] shows that manual placement of bus macros using even the accepted best practices might produce sub-optimal timing. A simulated annealing (SA) algorithm is used to place the bus macros. The placement algorithm is based on the following steps:

1. Copy the current solution.
2. Randomly swap the positions of bus macros and create a new placement.
3. Implement the design and find the effectiveness of the new placement.
4. Accept or reject the solution based on the acceptance probability.
5. Modify the temperature used for annealing and repeat until the cut-off temperature is reached.

The initial placement is obtained by random placement of bus macros along the PRR boundaries. The *timing score* metric reported by PAR is the total time (in picoseconds) of all the paths that fail the timing constraints specified for the design. All PRM combinations are implemented using the same placement and the sum of timing scores reported is the fitness score metric used to study the effectiveness of a placement. In each SA iteration, all combinations of PRMs are implemented and the algorithm usually converges in 250 iterations. Also, the top-level HDL file has to be manually updated with the correct type of bus macros after the tool identifies a good placement of bus macros. Placement of bus macros was found to affect timing inside a module.

Shaon et al. [18] attempt to automate the Xilinx EAPR flow. The focus of the work is on floorplanning the design and automating the placement of bus macros, and the designer

must manually instantiate the bus macros in the top-level HDL file. A SA algorithm places the bus macros and the tool takes around 25 hours for 100 iterations on a desktop PC with Intel Core 2 Duo E6750 2.66 GHz CPU with 3.24 GB of RAM. The tool was able to reduce the search space to find the optimal solution by 23% compared to a random search.

4.3 Functional Flow

The work by Carver *et al.* and Shaon *et al.* considers the placement of bus macros to be a global optimization problem. The tool runtimes in both the cases are in hours and not ideal for use in the DMD flow. Also, the top-level HDL has to be updated manually with the correct bus macro types once placement is completed. The DMD flow shown in Figure 4.5 completely automates the bus macro insertion and placement process.

When a new design is floorplanned by PATIS, the bus macro insertion tool parses the top-level HDL and extracts the design topology. The tool also determines the number of input and output bus macros required for each module in the design. The automatic placement tool creates a dummy design file using the design topology, the bus macro list, and the floorplan to place the bus macros. The bus macro placement information is used to identify the bus macro type required. For instance, inputs on the left boundary of a PRR requires left-to-right bus macros whereas the inputs on the right boundary of a PRR requires right-to-left bus macros. The placement tool updates the floorplanner UCF with the LOC constraints for placing the bus macros in the design.

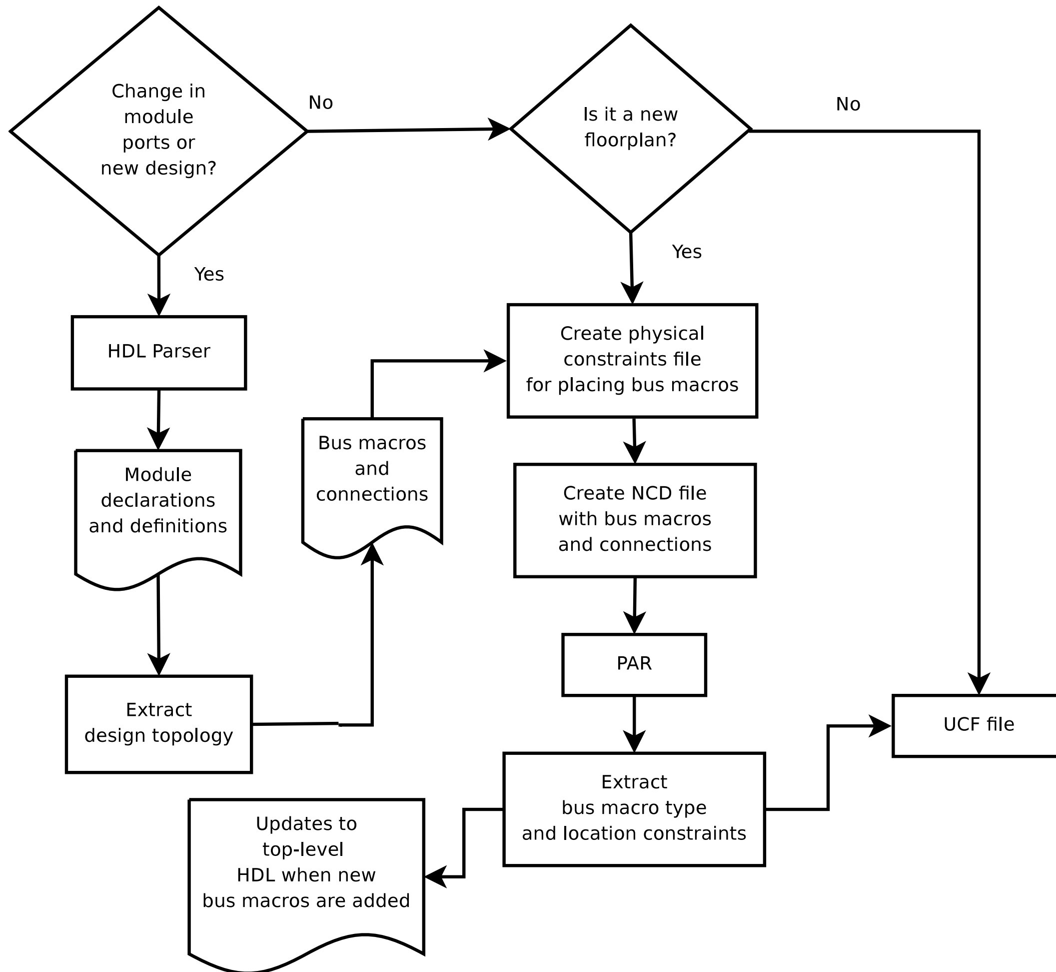

Figure 4.5: PATIS bus macro placement and insertion functional flow

4.4 Bus Macro Insertion Tool

The Xilinx PR flow uses bus macros to simplify the design implementation phase. The bus macros provide route-locking for module ports so that the PRMs can be built independently for the same base design implementation. Reconfiguration time is reduced since the partial bitstream updates only the PRRs and the static region essentially remains the same. The bus macros have to be manually instantiated and placed by the designer. As the bus macros support only eight-bit wide data, a large number of bus macro instantiations might be required in a design.

The DMD flow uses an automatic bus macro insertion tool with the following features:

1. Bus macro instantiations are automatically created and inserted in the top-level HDL.
2. Changes to the module ports are identified and appropriate changes are automatically made to the bus macro instantiations.
3. When bus macro placement changes, the type of bus macro to be instantiated is automatically identified and updated to the top-level HDL file.

The automatic bus macro insertion tool flow is shown in Figure 4.6. A VHDL parser is written to extract the module declaration and definition from the top-level HDL file. The module definitions are required to determine the signals that connect to the module ports and the module declarations are necessary to identify whether a port is used for input or output. The signals associated with the module ports can be of type bit, bit vector or an indexed part of a bit vector. Also, the tool requires the use of named association for the module definition to simplify the process of mapping the module port to the signals attached to them. The Xilinx design flow cannot implement the EDIF netlist created by the `ngc2edif` tool. Hence, insertion of bus macros has to be performed directly in the design HDL.

The tool aims to create a hypergraph of the design topology with bus macros as vertices and the connections as edges. In order to create the hypergraph, signals connecting two or more modules must be identified. The flexibility of HDLs allow the designer to create a bit slice of the signal and connect it to a port. In such a case, a simple comparison of signal names might result in connections being missed from the topology. Hence, the tool expands any vectors that connect to the module ports and uses comparison of signal names from this expanded list. The following combinations of module ports and signals are handled by the tool:

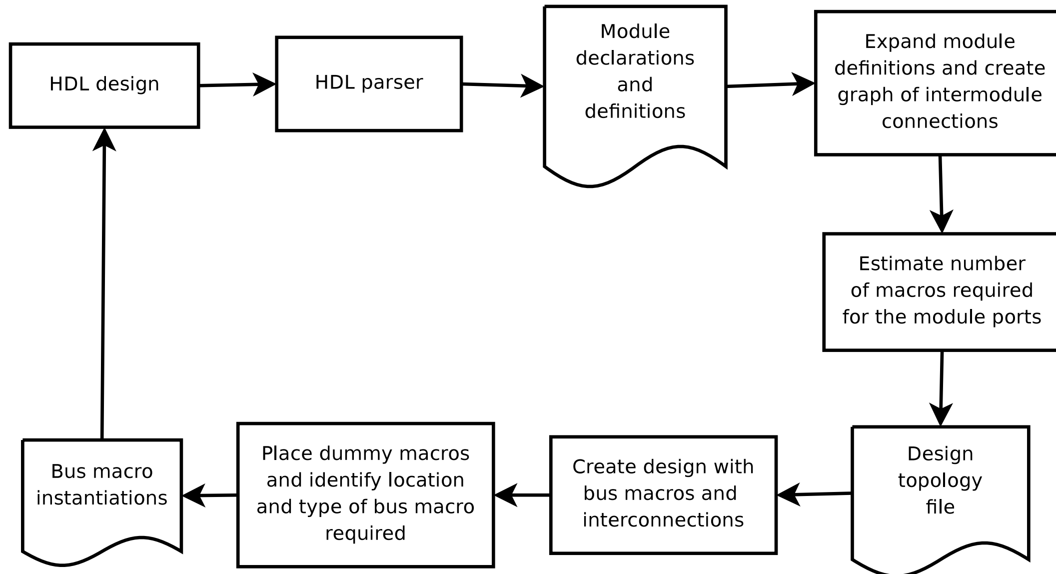

Figure 4.6: Automatic bus macro insertion flow

1. *port => signal*: The port and signal can be either both of type bit or type bit vector. The port name is searched in the module declaration to identify the correct port type. When the port is of type bit, no expansion is necessary. When the port is of type bit vector, the size of the vector is obtained from the module declaration and is expanded into individual signals by indexing.

2. *port(i to j) => signal*: In this case, both the module port and the signal is of type vector. The index for the module port is obtained directly from the positional association and the index for the signal is determined by searching for the signal in the top module. The bit vectors are expanded into signals of type bit by indexing.

3. *port => signal (i to j)*: The port and the signal attached to it is of type bit vector. The index for the port is obtained from the module declaration and the signal index is determined from the positional association.

4. *port (i to j) => signal (m to n)*: In this case, both the module port and the signal are of type bit vector and the index is determined from the positional association. The

vectors are expanded into individual signals of type bit.

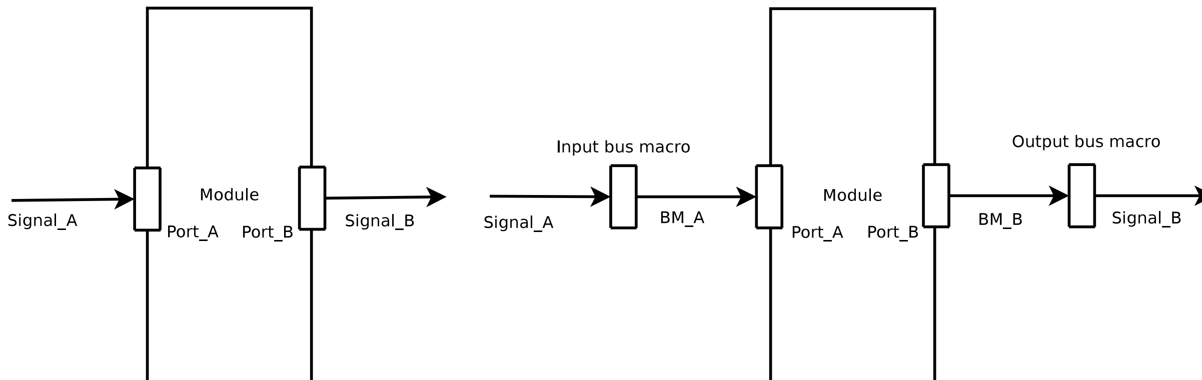

Figure 4.7: Module before and after insertion of bus macros

As the EAPR flow handles clock distribution to the PR region directly, bus macros are not needed to connect the clock to the PRM. The expanded signals in the module definition are separated into module inputs and outputs. The signals are then grouped to contain at most eight signals that connect to a bus macro. Consider the module shown in Figure 4.7 with input port `Port_A` and output port `Port_B`. `Signal_A` is an input to the module that connects to the module port `Port_A`, and `Signal_B` is the module output from the module port `Port_B`. Bus macro insertion happens such that the module input `Signal_A` is connected to the input port of a bus macro. A new signal `BM_A` is created to connect the output port of the bus macro to the module input port. Similarly for the output bus macro, a new signal `BM_B` connects the module output port to the input port of the bus macro and the output port of the bus macro provides the `Signal_B` to the base region.

Bus macros cannot be instantiated until the placement of bus macros is completed as the type of bus macro is based on the boundary on which it is placed. Once the signals that connect two or more modules are identified, the insertion tool extracts the bus macro name that will be instantiated for the signal. The bus macro insertion tool creates a *design topology file* that contains a list specifying the source bus macro name, source bus macro output pin,

destination bus macro name, and the destination bus macro input pin from the expanded module definition. The automatic bus macro placement tool uses the design topology file to construct a hypergraph with bus macros as vertices and the connections as edges.

4.5 Bus Macro Placement Tool

PATIS uses a two-step strategy to achieve timing closure for a design. In the first step, the bus macros are placed on the boundary of the PRR such that the delay associated with the inter-module nets is minimized. The second step requires manual intervention and is used during implementation when any of the PRMs do not meet timing. The two-step strategy helps to reject unusable floorplans immediately when the inter-module nets do not meet timing. The automatic bus macro placement implementation flow is shown in Figure 4.8.

The automatic bus macro placement tool uses the design topology file created by the bus macro insertion tool. During the first phase of the placement process, the base design and the PRM implementations are not considered for placement. The objective of this phase is to reduce the delay on the inter-module nets without considering the implementations of the modules. The automatic bus macro placement tool creates a new NCD design with just the bus macros and their interconnections. The Xilinx FPGA Editor is a graphical tool that displays a NCD file implemented on the FPGA, and can be used for creating an entirely new NCD design by hand [19]. The fpga_edline tool, a command line version of FPGA Editor, is used to create a new NCD design with the bus macro instantiations and their interconnections. As the type of bus macro to be used is not yet determined, the tool uses the left-to-right type macros in the design by default. Bus macro interconnections are created by adding nets between the source bus macro output pin and the destination bus macro input pin. Figure 4.9 is a screenshot of an NCD with four output pins of two different

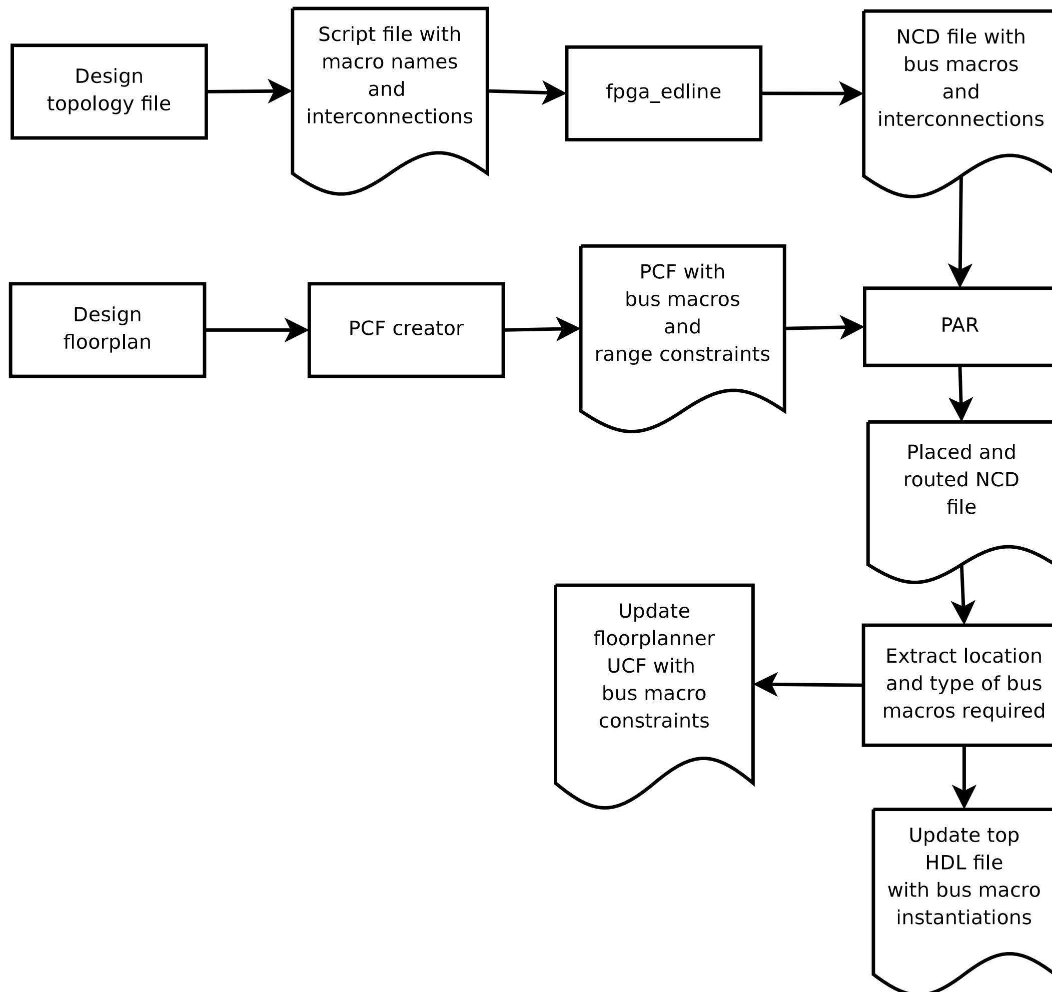

Figure 4.8: Automatic bus macro placement flow

macros on one module boundary connected to the eight input pins of a single macro on a different module boundary.

Bus macros are to be placed on their respective module boundaries such that the maximum delay on the interconnecting nets is reduced. The UCF created by the PATIS floorplanner contains the AG constraints for the PRRs. The automatic bus macro placement tool parses the floorplanner UCF to extract the module boundaries for each PRR in the design and uses a feature offered by PAR in which a hard macro can be implemented in a specified range of slices [20]. A Physical Constraints File (PCF) is created specifying the *LOCATE* constraints

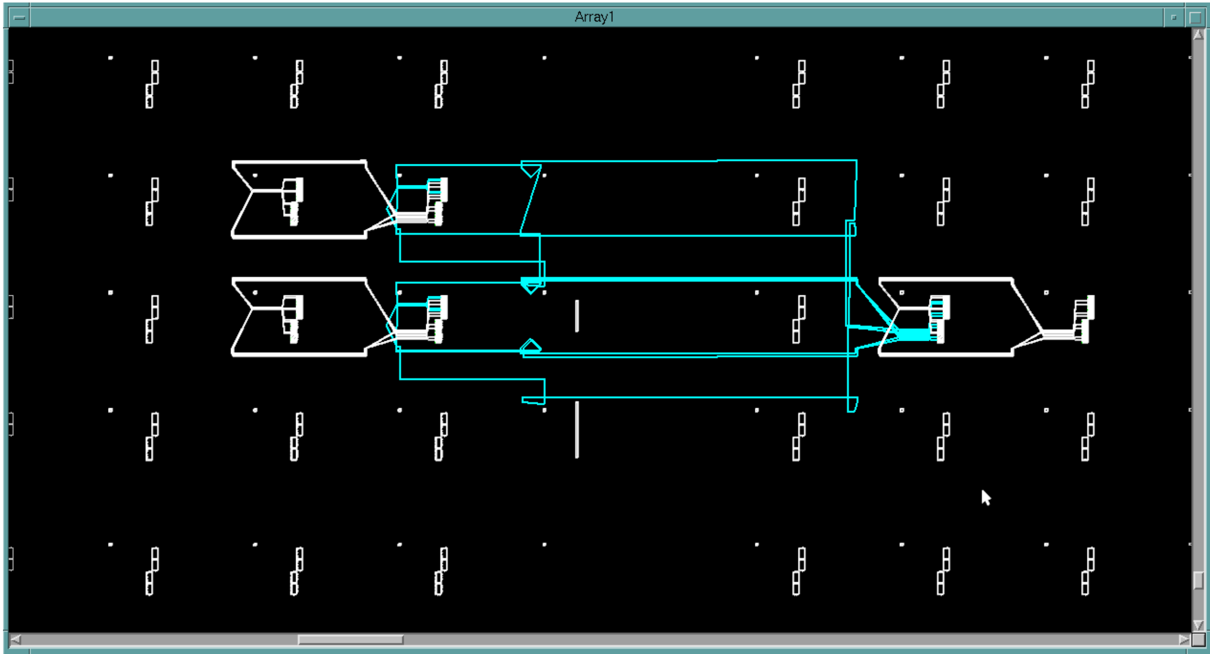

Figure 4.9: FPGA Editor screenshot with bus macros and interconnections

for the bus macros [21]. Since every bus macro requires two CLBs for its implementation, the range constraints are specified such that the bus macros can be implemented along a two CLB-wide area straddling the boundary of the PRR as shown in Figure 4.10.

Placement and routing of the NCD design with the bus macros and it interconnections is faster compared to PAR of the whole design. Because the NCD design contains only a few components, PAR can efficiently place the bus macros. Since the timing obtained is not based on any approximate routing models, the results are reproducible. Once the placement and routing of the design is complete, the NCD file from PAR contains the bus macros placed along the module boundaries.

A fpga_edline script file extracts the slice locations of the bus macros in the design. Using the slice locations and the floorplanner UCF files, the module boundary on which the bus macro is located can be identified. The boundary information is used by the bus macro insertion tool to identify the type of bus macro to be instantiated in the top-level HDL. When PAR

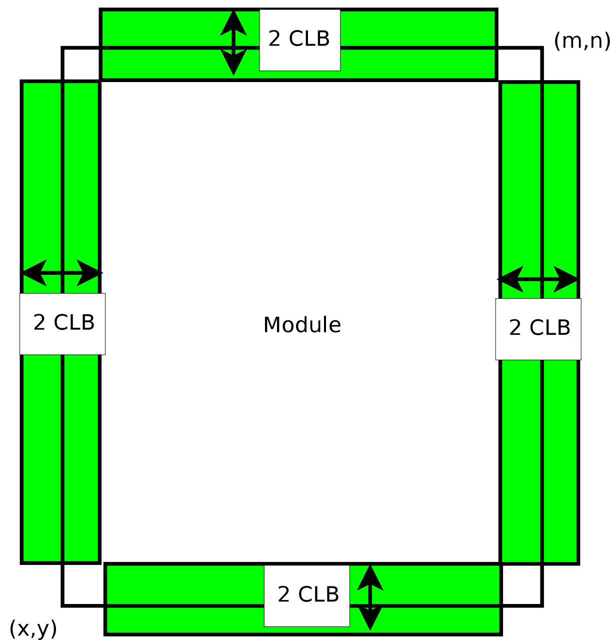

Figure 4.10: Two CLB-wide region on module boundaries

encounters timing error at this stage, a new floorplan is required as the implementation tools will not be able to meet timing at a later stage.

Any failure during implementation requires the designer to manually identify and solve the issue. The PRMs might fail timing during implementation because of the following reasons:

1. As the base design implementation uses the routing resources inside a PRR, the PRM module implementation can fail because of routing resource contention. This can be confirmed if the PRM can be implemented successfully with increased PRR size. In such a case a new floorplan is required with a size increase.

2. The study [1] used a design with two PRRs on the top corners of the FPGA to cause timing failures inside a module. The Xilinx EAPR Guide also uses a module with a combinational path spanning 80 rows to create timing failure inside a module. In most cases, the distance between the connected PRRs are longer than the size of the PRR

themselves, and optimizing the placement to reduce the inter-module routing delays is sufficient. Timing failures inside modules can be avoided by using good design principles such as registering the output of the module.

3. Although a good placement of bus macros is not based on the amount of logic inside a PRM, the placement might cause timing failures for some designs. When a high percentage of delay in the failing path is attributed to routing, it is a strong indicator that the placement of bus macros is contributing to the timing failure. As the basic resources in the FPGA are uniformly distributed, even a poor placement using synchronous macros should give a good timing performance. Also, as a good design principle, it is advised to register the outputs of a module. Hence, only rarely can timing fail inside a module because of the long combinational paths and in such cases PATIS requires the user to manually test placements for bus macros for that module.

The slice locations extracted using the fpga_edline tool is used to update the floorplanner UCF with the LOC constraints for the bus macros. Slice locations need to be adjusted suitably for each type of bus macro so that they are placed straddling the boundary of the PRR during implementation.

Chapter 5

Results

A suite of designs with varying resource requirements and sizes were developed and implemented using the DMD flow. The performance of the DMD flow is evaluated by comparing the tool runtimes and the timing performance achieved for the designs with the Xilinx MDF. This chapter begins with an introduction to the designs used in the benchmark suite and their characteristics. The tool runtimes of the DMD flow and the Xilinx MDF are presented along with the speedup achieved using the DMD flow. Performance of the automatic bus macro placement tool is also analyzed.

5.1 Platform Description

The designs were implemented on a Intel Core i7-920 2.66 GHz Desktop with 12 GB of memory running Ubuntu 8.10 on the 2.6.27 generic Linux kernel. All the benchmark designs target a Virtex-4 FX100-ff1517 FPGA device [16]. The Xilinx ISE 9.2.04i toolchain was used with the PR-12 patch.

5.2 Benchmark Suite

The PATIS benchmark suite contains a variety of designs enumerated below, with different resource requirement vectors and chip utilizations. In these designs, the clock frequency constraints are chosen to make timing challenging but attainable.

1. **Cascaded complex FFTs**: Fast Fourier Transform(FFT) is an efficient algorithm to compute the Discrete Fourier Transform (DFT) [22]. DFT is a type of Fourier transform used to convert time domain data to the frequency domain. DFTs have extensive applications in spectral analysis and data compression. Two versions of the design were created with either three or six modules connected in a ring topology using two 16-bit buses between the modules. The modules contain two or three 1024-point pipelined complex FFT cores. The three-module design `CFFT 3`, is the smallest in the benchmark suite with a chip area utilization of almost one-sixth, while the six-module design `CFFT 6` used about one-third of the FPGA, as shown in Figure 5.1.

2. **Multiple MicroBlaze processors**: The MicroBlaze is a 32-bit RISC Harvard architecture soft processor core targeting Xilinx FPGAs [23]. A MicroBlaze processor was generated using the Xilinx EDK with bidirectional Fast Simplex Link interfaces. Several instances of the MicroBlaze were connected in a ring topology. The MicroBlaze processors are mainly used in embedded applications and each MicroBlaze core requires three DSP48 slices and four RAMB16 blocks. Different versions of this design are generated by instantiating different numbers of MicroBlaze processors. Figure 5.2 shows the five MicroBlaze system `MB 5` created with a chip utilization of 40%.

3. **A scalable Viterbi decoder**: A Viterbi decoder uses the Viterbi algorithm to decode a bitstream that has been encoded using forward error correction based on a convolutional code. The Viterbi decoder has a high resource requirement to perform

maximum likelihood decoding. Each Viterbi decoder requires twenty BRAMs for its implementation. The memory-intensive Viterbi decoder is scalable according to the constraint length for the convolution code [24]. Several modules of differing sizes were implemented by varying the constraint length, and then cascaded to form a ring. The `Viterbi` design was implemented with seven modules that occupy approximately 70% of the chip, as shown in Figure 5.3.

4. **FloPoCo implementations of arithmetic expressions**: Floating Point Cores (FloPoCo) [25] is an arithmetic core generator that can implement fixed-point, floating-point and integer expressions. The tool generates HDL for cores such as square root, logarithm and collider that can be instantiated in the design [25]. These operations mainly use the DSP48 slices for their implementation and the intermediate results were stored in the BRAMs. Several data-intensive modules implementing different arithmetic operations were implemented in two variants of this application. The first variant, `FloPoCo 8`, has eight modules with around 85% chip area utilization. The other variant, `FloPoCo 10`, contains 10 modules with approximately 90% chip area utilization. The layouts of the `FloPoCo 8` and `FloPoCo 10` designs are shown in Figures 5.4 and 5.5, respectively.

5. **Video filters**: The video filter design contains a single PRR that is used to implement a grayscale filter or a negative filter. The grayscale filter produces the grayscale equivalent of a RGB32 pixel and the negative filter performs one's complement of the RGB32 pixel to produce the negative of a image. Mplayer, a video player for Linux, is modified to send the video frame data to the target board and display back the processed data [26]. Data from the workstation is transferred to the target ADM-XRC-4FX [16] board through the PCI interface. The design implemented on the FPGA receives the data and sends the processed data back to the workstation. The design is used only

for the proof-of-concept experiment and is not used for the DMD flow performance evaluation experiments since the design has only 2% chip area utilization.

The PATIS floorplanner generates the AG constraints for the PR modules in the design. Using the floorplanner UCF file, the automatic bus macro placement tool determines LOC constraints for the bus macros in the design. The bus macro insertion tool uses the LOC constraints to identify the type of bus macro to be instantiated in the design HDL.

5.3 Automatic Bus Macro Placement Performance

The automatic bus macro placement tool uses a design with the bus macro instantiations and its interconnections to find a good placement of bus macros. As the type of bus macro to be instantiated cannot be determined before its placement, the tool uses a temporary bus macro type initially. Table 5.1 shows the maximum net delay for the benchmark designs when different bus macro types are used. The maximum delays obtained by using the left-to-right bus macro as the temporary bus macro type is found to have the lowest root-mean-square error. Hence, the left-to-right bus macro is used by the automatic placement tool to create the design NCD.

Table 5.2 compares the maximum net delays in benchmark designs using automatic placement compared to manual placement by a PR expert. The automatic bus macro placement tool lowers the delay on intermodule nets for all the benchmark designs. For instance, the automatic placement tool reduces the maximum delay by 29% for the `Viterbi 7` design. The tool produces a 5.98% improvement over the manual placement for the `Flopoco 8` design. The following observations are made from the implementation experiments performed using the automatic bus maro placement tool.

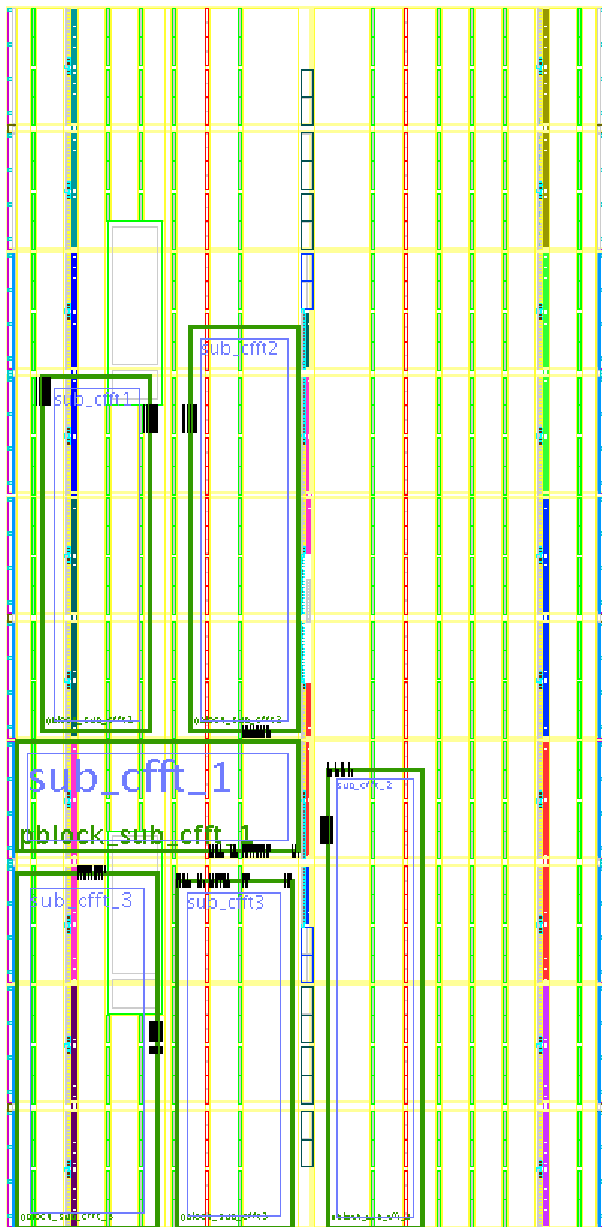

Figure 5.1: Design – `CFFT 6`

(Screenshot captured from PlanAhead tool showing bus macros on the module boundaries)

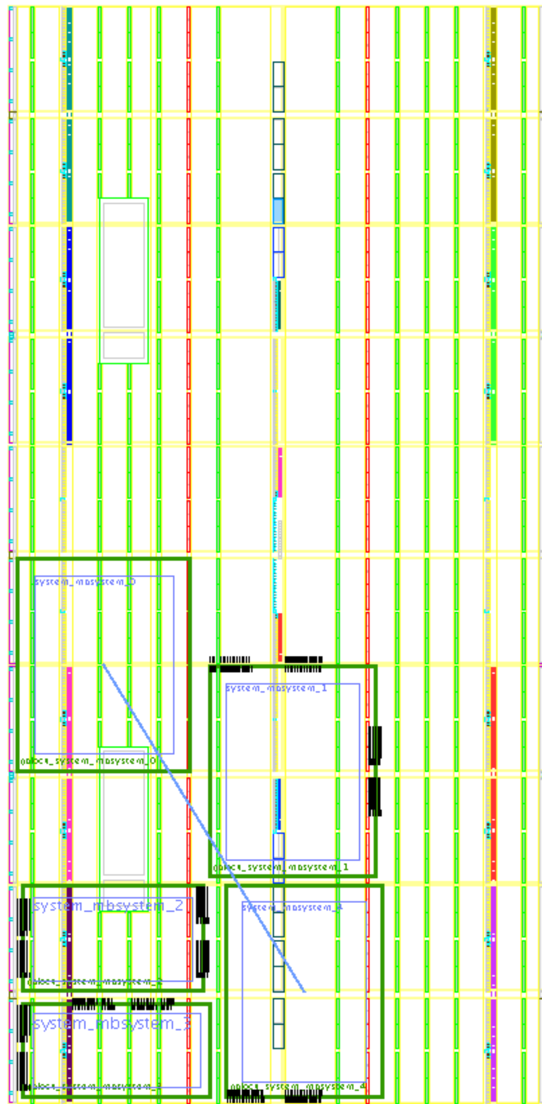

Figure 5.2: Design – `MB 5`

(*Screenshot captured from PlanAhead tool showing bus macros on the module boundaries*)

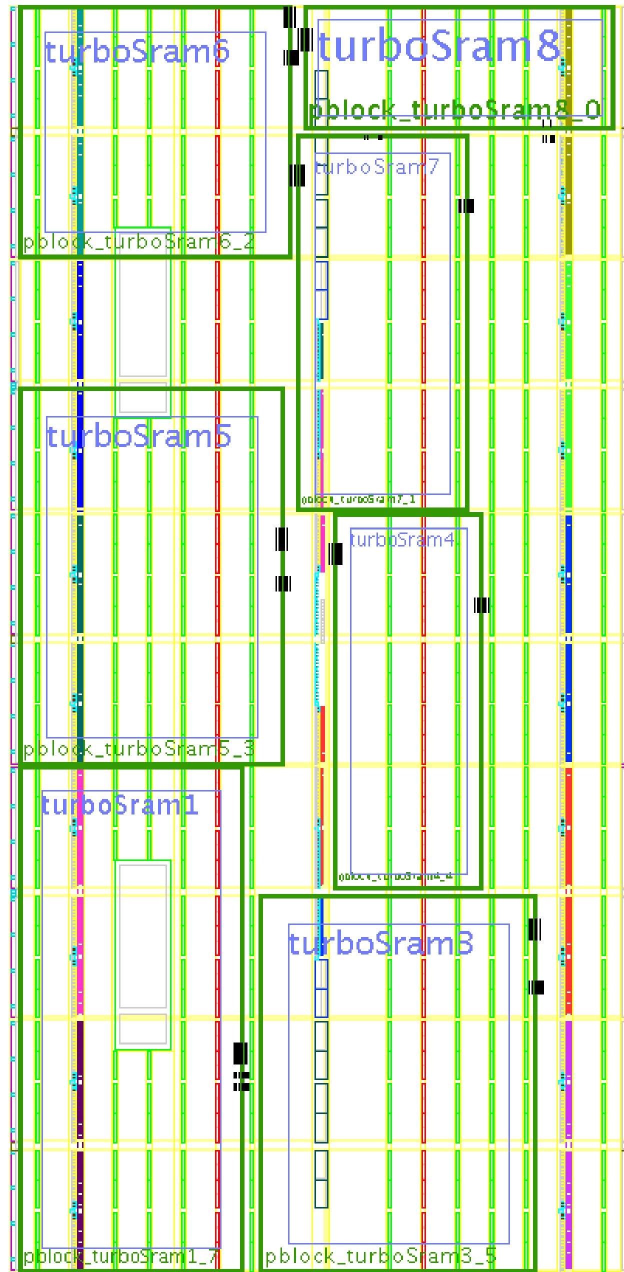

Figure 5.3: Design – `Viterbi 7`

(Screenshot captured from PlanAhead tool showing bus macros on the module boundaries)

Figure 5.4: Design – `FloPoCo 8`

(Screenshot captured from PlanAhead tool showing bus macros on the module boundaries)

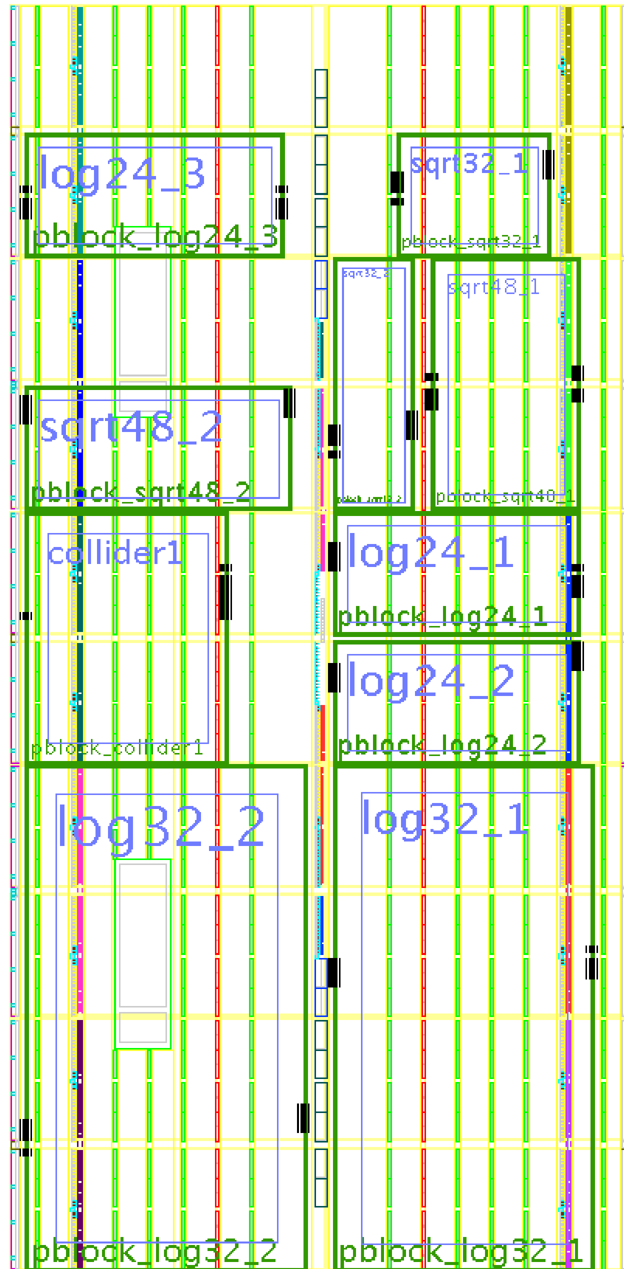

Figure 5.5: Design – `FloPoCo 10`

(Screenshot captured from PlanAhead tool showing bus macros on the module boundaries)

Table 5.1: Delay relative to different bus macro types

Design name	Maximum delay (ns)			
	T2B	**B2T**	**L2R**	**R2L**
CFFT 3	2.81	2.39	2.65	2.74
MB 5	2.17	1.90	2.35	2.15
CFFT 6	2.81	2.51	2.55	2.38
Viterbi 7	4.38	4.44	4.46	4.56
FloPoCo 8	3.67	4.07	3.99	4.88
FloPoCo 10	2.55	2.75	2.67	2.54

Legend: T2B – Top-to-bottom bus macros; B2T – Bottom-to-top bus macros; L2R – Left-to-right bus macros; R2L – Right-to-left bus macros.

Table 5.2: Comparison of manual and PATIS bus macro placement

Design / # modules	# bus macros	Avg delay (ns)		Max delay (ns)	
		Manual	**PATIS**	**Manual**	**PATIS**
CFFT 3	24	3.02	2.33	3.30	2.82
MB 5	84	3.66	3.44	4.85	4.01
CFFT 6	48	2.50	2.30	2.80	2.40
Viterbi 7	35	6.27	4.18	6.51	4.62
FloPoCo 8	65	5.00	4.40	5.20	4.90
FloPoCo 10	80	3.01	2.19	3.27	2.81

Legend: Avg delay – average over the 10 longest nets, as reported by PAR; Max delay – maximum delay, as reported by PAR.

1. The delay on a net is dependent upon the net length, and the number of Programmable Interconnection Points (PIPs) used to connect the source to the destination. The maximum delay for the `Flopoco 8` design is between a net connecting two bus macros that are placed 62 slices apart. Counterintuitively, the `Viterbi 7` design has a lower value for its maximum delay net as shown in Table 5.2 even though the source and destination for the net is separated by 100 slices. In such cases, the automatic placement yields better results compared to manual placement.

2. The least improvement in maximum delay is observed for the `Flopoco 8` design. The

floorplan generated for the design is such that the connected modules are parallel to one another in the layout. In such a case, even an intuitive manual placement of bus macros produces results that are comparable to the automatic bus macro placement tool.

3. Timing errors are produced when the `FloPoCo 10` base design is implemented to operate at 197.33 MHz, as the manual bus macro placement is not sufficient to meet the strict timing constraints. The same design can be operate at 197.33 MHz using the constraints created by the automatic bus macro placement tool.

The LOC constraints created by the automatic bus macro placement tool are used to place the bus macros during implementation by the DMD flow. Since the automatic bus macro placement tool aims to reduce only the intermodule net delays, a study is performed on the impact of bus macro placement to delays inside modules. Table 5.3 shows the maximum delays for the benchmark designs in the base region and in the PR module. Since the frequency at which a design can operate is constrained by either the delay in the base design or with any of the PR modules, the measurements shows the region where the longest delay net is present. The delays listed in Table 5.3 is given by the PAR program when the designs are implemented. The following observations are made from the experiments conducted with

Table 5.3: Maximum delay on nets

Design name	**Maximum delay (ns)**	
	Base region	**PR modules**
MB 5	5.82	5.76
CFFT 6	8.4	9.62
Viterbi 7	6.64	5.97
FloPoCo 8	7.21	7.21
FloPoCo 10	6.18	6.27

the base design and PR module implementations:

1. The maximum delay-causing net in the `FloPoCo 10` design is present in one of its modules that has a twenty two CLB long carry chain. When the design frequency is increased, the static region gets re-implemented according to the timing constraints specified but the PR module implementation fails with timing errors. The failing module has around 90% resource occupancy and hence, PAR is not able to implement the design with strict timing constraints. Although a large routing delay on the failing net can be attributed to the bus macro placement, the presence of long carry chains can also contribute significantly to the delay. In the case of the `FloPoCo 10` design, the carry chain in the PR module contributes 3.34 ns to the delay.

2. When the design frequency is increased for the other designs, the PAR program takes longer to implement the base design and the PR modules. As PAR implements one module at a time, the module implementation can be tried with different cost tables for its initial placement. When the design is implemented as a whole, considering multiple implementations for each module increases the runtime of the tools by a large extent.

3. The objective of the automatic bus macro placement tool is to produce a placement that can meet the timing on the intermodule nets. Since the module ports are generally registered, the tools can generate implementations for most designs. In the case of modules that fail timing because of high resource occupancy, PATIS can be used to generate a new floorplan for the design.

The runtime of the automatic bus macro placement flow is shown in Table 5.4. Runtimes are separated into design creation time, PAR time, and the time required for constraint extraction as they are used independently under different scenarios:

Table 5.4: Tool runtimes

Design name	**Tool runtime (CPU seconds)**		
	Design creation	**PAR of design**	**Constraints extraction**
CFFT 3	139	59	16
MB 5	395	114	18.6
CFFT 6	310	71	18.7
Viterbi 7	241	156	17.15
FloPoCo 8	395	124	18.3
FloPoCo 10	479	64	19.7

1. When a new design is implemented using the DMD flow, the automatic bus macro placement tool creates the design NCD with the bus macros and its interconnections. The design is placed-and-routed and the LOC constraints for the bus macros in the design are extracted from the NCD created by PAR.

2. The automatic placement tool creates a new design when the module ports change. The tool runtime for creating the design is directly proportional to the number of bus macros in the design.

3. Placement and routing of the design is performed everytime the floorplan changes.

4. The LOC constraint extraction requires roughly the same time irrespective of the number of bus macros in the design.

The LOC constraints created by the automatic bus macro placement tool are used for implementing the design using the DMD flow.

5.4 DMD Flow Performance

The benchmark designs are implemented using the DMD flow and a manually floorplanned Xilinx MDF. The PAR runtimes of the flows are analyzed to evaluate the implementation improvements arising from the DMD flow. The PAR runtimes is shown in Table 5.5. The designs are implemented at a frequency that is attainable using the MDF.

Table 5.5: Place-and-route times after floorplanning

Design	f_{clk} **(MHz)**	**Elapsed PAR time (minutes)**		
		Manual floorplan	**PATIS floorplan**	**PATIS incremental (modules changed)**
CFFT 3	256.40	35	10	5 (2)
MB 5	127.40	80	17	7 (2)
CFFT 6	170.20	175	16	5 (3)
Viterbi 7	44.40	380	18	4 (1)
FloPoCo 8	74.10	120	11	5 (2)
FloPoCo 10	80.32	124	12	4 (1)

The modules in the benchmark designs are manually assigned AG constraints and place-and-route is performed on the design as a whole in the Xilinx MDF. In the DMD flow, implementation acceleration comes from the parallel reduction of a large global optimization problem to a set of smaller independent problems. The PAR time given in Table 5.5 for the PATIS flow is the total time required for place-and-route of the base design and the maximum of the PAR time for all the modules. The runtimes of the rest of the modules in the design are not considered as their implementation can be completed concurrently along with the module that requires the maximum PAR time. As the modules are implemented only within the range specified by their AG constraints in the EAPR flow, concurrent implementation becomes possible. The DMD flow accepts some degradation in area and speed in order to produce a faster design implementation. Because synchronous bus macros are used in the

designs, module ports are automatically registered and hence the modules have better timing performance. For instance, the `Viterbi 7` design can achieve a frequency of 71.4 MHz using the DMD flow and 44.4 MHz in the Xilinx MDF. Similarly, the `FloPoCo 8` design can be implemented at 121.4 MHz using the DMD flow compared to 74.1 MHz using the Xilinx MDF.

Incremental floorplanning re-implements only the changed modules, leading to shorter place-and-route times shown under PATIS incremental in Table 5.5 for the number of modified modules given in parentheses. In the manual non-PR flow, any change in the design requires re-implementation of the entire design. In DMD, the incremental floorplanner tries to modify the existing floorplan which may reduce the runtime by re-implementing only modified modules. The incremental floorplanner tries to reduce the number of modified modules by reducing the ripple effects.

The following observations are made from the implementation of benchmark designs using the DMD flow and the Xilinx MDF:

1. In the DMD, the total implementation time is the sum of the time required for implementing the static design and the maximum module implementation time. Hence, the time required for implementing a design is not dependent upon the size of the design but on the module that requires the longest implementation time. As seen in Table 5.5, all the benchmark designs can be implemented on the order of minutes eventhough they vary widely in their sizes.

2. Synchronous bus macros are used for registering module ports so that modules have good timing performance. As the module ports are registered, no combinational path crosses the module boundaries and hence the operating frequency of a design is constrained only by the ability of PAR to implement a design within the region specified.

Delays in the static region is also reduced by the automatic bus macro placement tool.

3. Design implementation using the DMD flow does not have the additional routing overhead that exists in a PR design. Usually, additional signals are routed to the enable ports of bus macros to bring the port outputs to zero when RTR is performed on the design. In the DMD flow, bus macro enable signals are always tied high.

Figure 5.6 shows the speedup achieved when the design is implemented using the DMD flow compared to the manually floorplanned Xilinx MDF. The runtimes shown for the manual floorplan and PATIS floorplan are for a complete re-implementation of the design. Incremental changes to modules require placement and routing of that module alone and hence achieves good speedup when compared to complete re-implementation. The speedup using the DMD flow for complete re-implementation comes from the concurrent implementation of modules and the use of synchronous bus macros.

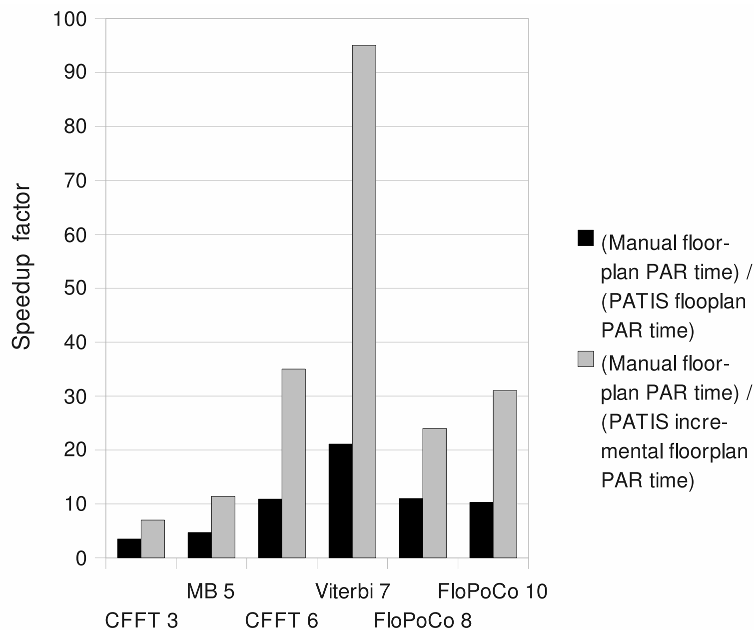

Figure 5.6: PATIS runtime improvements

Chapter 6

Conclusions and Future Work

This thesis presents a novel approach developed to accelerate the implementation of static FPGA designs. The Xilinx standard flow treats the implementation of FPGA designs as a global optimization problem and hence may require re-implementation of the complete design even for a small change in one of its modules. Although the Xilinx modular design flow supports independent implementation of modules by the designers, the bitstream for the final design cannot be generated directly from the implementation files. Xilinx partitions support incremental updates to the design but requires complete re-implementation of all modules when a module does not achieve timing closure after re-implementation. Using the DMD flow, incremental changes can be incorporated into the design by re-implementation of the changed modules only. Also, a complete re-implementation is still rapid as the modules can be implemented independently by concurrent invocations of the standard place-and-route tools.

The DMD flow leverages the features offered by the Xilinx EAPR flow to accelerate FPGA design implementation. As the PRMs can be reconfigured during run-time, the modules are restricted to use only the resources contained inside the PRR defined for its implementation.

Hence, the PR flow indirectly facilitates parallel implementation of modules. Although implementing the hierarchical netlist in separate PRRs reduces the area and optimizations that can be produced by the implementation tools, it is still preferred so that designs can be implemented faster. The EAPR flow requires additional effort from the designer as the design has to be manually floorplanned, and the bus macros required for the PRMs have to be manually instantiated and placed. Since the tools created for the DMD flow automate floorplanning, bus macro instantiation and placement in the PR flow, the design entry process is similar to that of the standard flow.

The important features of the DMD flow are listed below:

1. The automatic floorplanner assigns AG constraints to modules such that the resource requirements of the modules are respected. The floorplanner aims to generate a layout in which connected modules are placed close to one another.

2. Speculative floorplanning to generate a suite of floorplans by anticipating changes to modules using their viscosity and fickleness attributes.

3. Incremental floorplanning speeds up generation of a new floorplan by modifying an existing floorplan.

4. Automatic instantiation of bus macros required for the PR modules. The bus macros are automatically placed such that the delays on the intermodule nets are reduced.

5. A debug module passively monitors the module interfaces through configuration read back.

6. Concurrent implementation of modules so that a complete re-implementation is faster compared to implementation using the existing flows. Incremental changes requires re-implementation of only the changed modules.

The contribution of this work to the DMD flow is the development of the automatic bus macro insertion and placement tool. Implementing a PR design requires additional effort from the designer. To ensure that the productivity improvements resulting from the DMD flow are not lost because of the additional effort required, automation of design floorplanning, bus macro insertion and placement are critical. Using the DMD flow produces a 10× speedup on average for a complete implementation of the benchmark designs. The use of synchronous bus macros and concurrent implementation of the modules is responsible for the speedup produced by the DMD flow. Incremental changes using the DMD flow are 35× faster on average compared to a complete design implementation.

6.1 Future Work

The tools created were aimed at reducing the additional effort required from the designer for implementation using the DMD flow. Since the DMD flow uses the PR implementation tools, the modules in the design also have the same restrictions that are enforced on PR modules in the EAPR flow. For instance, modules of the implemented static design cannot contain any clock or reset-related primitives. Removing such restrictions on modules require changes to the Xilinx PR implementation tools and is beyond the scope of this work.

The usability and performance of the DMD flow can be improved further in the following ways:

1. **Portability:** The DMD flow currently supports Xilinx Virtex-4 devices. The methodology is scalable to other architectures that support active PR. Since bus macros are pre-placed and pre-routed for a specific architecture, support for new architectures require instantiation and placement of the appropriate type of bus macro. The configuration frame size is architecture specific and hence the floorplanner has to be modified

appropriately.

2. **Flow Automation:** The PR flow requires the top-level HDL to contain only the black-box instantiations of its PR modules. Hence, the DMD flow requires the designer to synthesize and implement the modules and the top-level HDL in separate directories. The standard flow does not enforce such restrictions and it is desirable to automate the flow such that the synthesis and implementation of top-level HDL and the modules occur in separate directories.

3. **Verilog top module support:** The bus macro insertion tool can create instantiations for a VHDL top-level file as it contains component declarations and instantiations for all the modules. In case of a Verilog top-level file, only the component instantiations are present and the module declarations are present in separate directories where the modules are synthesized. Hence in order to support automatic bus macro insertion for Verilog top-level HDL files, HDL for multiple modules should be parsed. It is desirable to include bus macro insertion for Verilog top-level HDL as Verilog is supported by the Xilinx EAPR flow.

Bibliography

[1] J. M. Carver, R. N. Pittman, and A. Forin, "Automatic bus macro placement for partially reconfigurable FPGA designs," in *FPGA'09: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays.* ACM, 2009, pp. 269–272.

[2] H. J. Kahn and R. F. Goldman, "The electronic design interchange format EDIF: present and future," in *DAC '92: Proceedings of the 29th ACM/IEEE Design Automation Conference.* Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 666–671.

[3] FPGA design flow overview. [Online]. Available: http://www.xilinx.com/itp/xilinx4/pdf/docs/dev/dev.pdf

[4] Xilinx MAP Program. [Online]. Available: http://www.xilinx.com/itp/xilinx7/books/data/docs/dev/dev0065_12.html

[5] Xilinx BitGen Program. [Online]. Available: http://www.xilinx.com/itp/xilinx4/data/docs/dev/bitgen.html

[6] Xilinx Modular Design Flow. [Online]. Available: http://www.xilinx.com/itp/xilinx7/books/data/docs/dev/dev0025_7.html

[7] Using SmartGuide. [Online]. Available: http://www.xilinx.com/itp/xilinx10/isehelp/ise_p_using_smartguide.htm

[8] Xilinx 5.1i incremental design flow. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp418.pdf

[9] Xilinx Guided PAR. [Online]. Available: http://www.xilinx.com/itp/xilinx7/books/data/docs/dev/dev0088_14.html

[10] Incremental design reuse with partitions. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp918.pdf

[11] Early access partial reconfiguration user guide. [Online]. Available: http://www.xilinx.com/support/prealounge/protected/docs/ug208_92.pdf

[12] PlanAhead user guide. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/PlanAhead_UserGuide.pdf

[13] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs," in *FPL 2006, International Conference on Field Programmable Logic and Applications.* IEEE, 2006, pp. 1–6.

[14] T. Frangieh, A. Chandrasekharan, S. Rajagopalan, Y. Iskander, S. Craven, and C. Patterson, "PATIS: using partial configuration to improve static FPGA design productivity," in *17th Reconfigurable Architectures Workshop (RAW 2010), Atlanta, GA*, 2010.

[15] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: Application in VLSI domain," in *DAC '97: Proceedings of the 34th Annual Design Automation Conference.* ACM, 1997, pp. 526–529.

[16] Alpha Data ADM-XRC-4FX. [Online]. Available: http://www.alpha-data.com/products.php?product=ADM-XRC-4FX

[17] Relocation and Automatic Floor-planning of FPGA Partial Configuration Bit-Streams. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=70619

[18] S. Yousuf and A. Gordon-Ross, "DAPR: Design Automation for Partially Reconfigurable FPGAs," in *The International Conference on Engineering of Reconfigurable Systems and Algorithms*. ERSA, July 2010.

[19] Xilinx FPGA Editor. [Online]. Available: http://www.xilinx.com/itp/xilinx6/help/fpga_editor/fpga_editor.htm

[20] Xilinx Place and Route Program (PAR). [Online]. Available: http://www.xilinx.com/itp/data/alliance/dsu/dsu3_3.htm

[21] Xilinx Constraints Guide. [Online]. Available: http://www.xilinx.com/itp/xilinx8/books/docs/cgd/cgd.pdf

[22] Fast Fourier transform. [Online]. Available: http://mathworld.wolfram.com/FastFourierTransform.html

[23] MicroBlaze processor reference guide. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf

[24] Adaptive soft output Viterbi algorithm (ASOVA) turbo decoder. [Online]. Available: http://www.ecs.umass.edu/ece/tessier/rcg/benchmarks/asova.html

[25] FloPoCo. [Online]. Available: http://www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/

[26] Mplayer. [Online]. Available: http://www.mplayer.org/

Nomenclature

AG Area Group, page 21

ALM Adaptive Logic Module, page 6

BRAM Block RAM, page 8

CMT Clock Management Tile, page 7

CPLD Complex Programmable Logic Device, page 6

DCM Digital Clock Manager, page 7

DFT Discrete Fourier Transform, page 52

DMD Dynamic Modular Design, page 2

EDIF Electronic Design Interchange Format, page 8

FFT Fast Fourier Transform, page 52

FloPoCo Floating Point Cores, page 53

FPGA Field-Programmable Gate Array, page 1

GUI Graphical User Interface, page 21

HLV High-Level Validation, page 30

ICAP Internal Configuration Access Port, page 30

IP Intellectual Property, page 10

IRL Irreducible Realization List, page 26

LLD Low-Level Debugging, page 30

LOC Location, page 21

MDF Modular Design Flow, page 12

MMCM Mixed Mode Clock Manager, page 7

NCD Native Circuit Design, page 10

NGC Native Generic Circuit, page 9

NGD Native Generic Database, page 10

PAR Place and Route, page 2

PATIS Partial module-producing, Automatic, Timing-aware, Incremental, Speculative floorplanner, page 2

Pblock Physical block, page 21

PCF Physical Constraints File, page 47

PDC Programmable Debug Controller, page 30

PIP Programmable Interconnection Point, page 60

PR Partial Reconfiguration, page 2

PRM Partially Reconfigurable Module, page 19

PRRs Partially Reconfigurable Regions, page 19

QoR Quality of Results, page 11

RTR Run-time Reconfiguration, page 18

SA Simulated Annealing, page 40

UCF User Constraints File, page 21

XST Xilinx Synthesis Tool, page 20