UNIVERSITY OF OSLO
Department of Informatics

Reconfigurable FPGA Accelerator for Databases

Master thesis

Jonas Julian Jensen

August 1, 2012

Abstract

Database management systems have traditionally been implemented entirely in software. However, adding hardware to database cluster servers to gain more speed has its price. Firstly, the cost of the hardware itself, secondly the increased power consumption from the numerous processors that have to be used.

The purpose of this study is to investigate how partial run-time reconfiguration in FPGAs can be used to accelerate databases. It aims to show how an FPGA based query processor can work in collaboration with a regular software database to accelerate certain queries.

This thesis proposes a novel way of using dynamic partial reconfiguration in FPGAs to process arbitrary queries in hardware. We investigate how SQL queries can be decomposed and turned into hardware modules that are 'stitched' together at run-time to form a stream processing datapath. Consequently, a set of customizable hardware modules that each can implement a range of SQL operators are presented. In addition, the thesis gives a method for floorplanning a high capacity FPGA for slot based partial reconfiguration.

In the end, by the help of a case study we can conclude that the main bottleneck is the interface to the host computer. Another interesting finding is that unlike conventional databases, the accelerator is not slowed down by a more complex query. On the contrary, filtering out results actually speeds it up.

Acknowledgements

Foremost, I would like to sincerely thank my supervisor Postdoc. Dirk Koch for very clear guidance and constructive feedback throughout the course of this project.

I wish to express my gratitude to my girlfriend Lene for moral support and for proofreading and correcting my (sloppy) English.

I would also like to thank my brother Daniel who provided valuable comments to the writing of this thesis.

Last but not least, Thanks to Martin, Geir and everyone else at the lab who were exposed to my _ _ _ _ humor. You will be missed!

Jonas Julian Jensen
University of Oslo
August 1, 2012

Contents

List of Figures

Chapter 1

Introduction

Database management systems have traditionally been implemented entirely in software, and are typically run on servers using standard operating systems. High-end databases utilize multiprocessor computers with lots of memory and RAID disk arrays to achieve high speed data retrieval and to ensure data resilience. This approach is a well proven and cost-effective way of dealing with database applications such as inventory systems found in web shops or travel itinerary databases used by travel agencies. In contrast, when it comes to high volume database operations like data mining and analytical processing, conventional databases constitute a bottleneck. This project aims to demonstrate how to gain performance improvements in databases using run-time reconfigurable field-programmable gate arrays (FPGAs).

During the 1990s the use of FPGAs in digital electronics increased dramatically. Due to its ability to adapt a wide variety of circuits it has been used in everything from system on chip (SoC) to digital signal processors, and other kinds of specialized processors and digital circuitries. It has been embraced by the industry due to its ability to be reconfigured, and due to the low startup cost of producing FPGAs compared to application-specific integrated circuits (ASICs). Economic risks involved with launching new FPGA implementations are low compared to the enormous cost of having to do a fabrication process re-spin if it turns out an ASIC implementation has to be discarded.

Also, the reconfigurability means that vendors can offer updates, and sell functionality to customers long after the hardware has deployed, with vendors having to do no more than sending the updates electronically. This is all beneficial, but in a sense FPGAs have been used merely as an ASIC substitute. This does not utilize the full advantages of reconfigurable hardware over ASICs. Unlike ASICs that once produced are like cast in stone, certain FPGAs can be run-time reconfigured or partially reconfigured in a matter of milliseconds. By exploiting this ability, configuring the FPGA with the required hardware configurations on the fly, an FPGA will seemingly be bigger than it physically is, and can potentially provide more on-chip functionality than any ASIC. That is the real technological advantage FPGAs have over ASICs, and what this project is demonstrating for database acceleration.

Databases are commonly accessed via the Structured Query Language (SQL). SQL was developed by IBM in the early 1970s and has since become the most popular query language. For this reason, basing this project on the SQL syntax is the obvious choice. SQL has been standardized by the American National Standards

Institute; this standard is referred to as ANSI SQL.

1.1 Problem Description

Conventional databases offer some degree of parallelism. Speed gain from parallelism in cluster servers comes at a cost. It comes at the cost of the hardware itself and the increased power consumption from the numerous processors that are used.

One possible solution for increasing performance for such high volume database operations is turning to dedicated hardware for executing queries. Now, this is where run-time reconfigurable hardware becomes an interesting option due to some FPGAs ability to logically change its internal circuit at run-time. Because different queries inevitably would require a new circuit to be executed in hardware, no static hardware would be usable for accelerating arbitrary queries.

There are solutions that create dedicated hardware query accelerators for FPGAs (e.g. Glacier compiler[7]). This works only for a limited number of predefined queries, and it requires time consuming synthesis and place & route steps for every possible query. This thesis aims at creating any hardware SQL accelerator on-the-fly by using partial run-time reconfiguration of FPGAs.

The goal of this project is to show that some queries can be accelerated in hardware by stitching together basic building blocks at run-time. The building blocks implement SQL operators that are composed together with respect to the currently executed query.

This means that the system should be able to accelerate SQL queries that are not known at design time. This approach is only limited by the available SQL operators, the capacity of the FPGA, or the architecture of the system implemented on the FPGA.

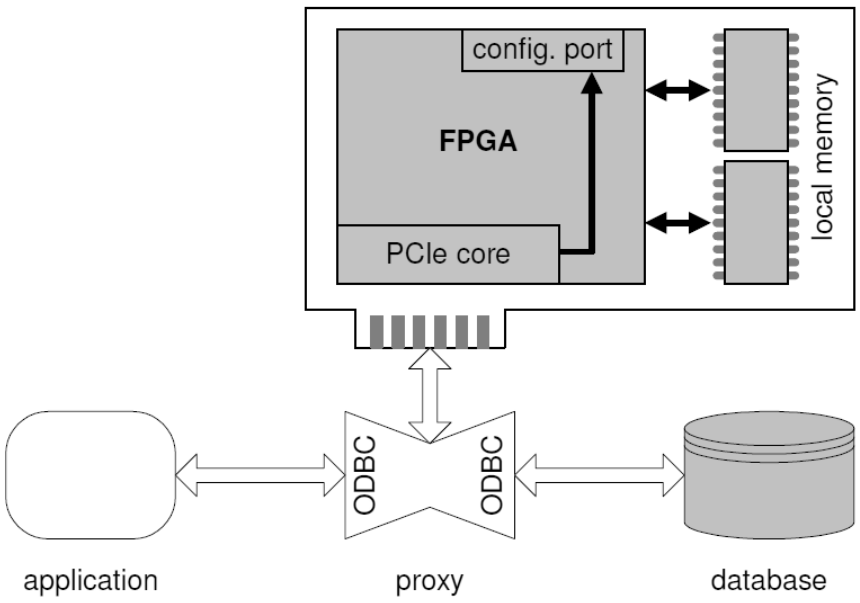

Figure 1.1: An overview of the system.

The overall architecture of the FPGA accelerated database system is shown in Figure 1.1. Without the accelerator an application can interact directly with a software database. Cross-platform database access is commonly implemented using the Open Database Connectivity Protocol (ODBC). By inserting a proxy between the application and the database, it is possible to forward queries that are suitable for FPGA acceleration to the FPGA. All remaining queries that are not supported by the FPGA will be forwarded to the original database (which is MySQL in this case).

The proxy consequently parses queries from the application and manages the FPGA accelerator hardware. This includes the creation of the query specific accelerators from the SQL library as well as the data movement. The proxy can store copies of all or selected tables from the database in local memory attached to the FPGA. More than 23GB of local memory is available for storing tables or temporary data on the dedicated hardware. The communication with the FPGA is done via a fast PCI-express interface. This allows fetching results from the FPGA with up to 2GB/sec.

In addition to the FPGA management and data movement, the proxy ensures data consistency between the tables stored in the local memory of the FPGA board and the database running on the host PC. With this system architecture, it should be possible to support the full functionality of the original database while being able to accelerate specific queries with the help of the reconfigurable FPGA.

This approach comprises some overhead for parsing queries, composing the accelerator, and for configuring the FPGA. However, for queries processing large data sets (GBs of data) this overhead can be neglected.

Such a database accelerator should act as a helper system to a conventional master database, taking load of the master database whenever possible. The accelerator must remain transparent to the application utilizing the database, as if the application was talking directly to the master database itself as shown in Figure 1.1. Queries may be passed on to the master database, or they may be intercepted by the accelerator, processed, and the results given back to the application without involving the master database.

1.2 Contribution

The main contribution of this thesis is to show that selected SQL queries can be accelerated by a run-time reconfigurable hardware in an FPGA. The thesis proposes methods for floorplanning the device to accommodate a reconfigurable datapath that spans over one continuous region or over several disjoined regions on the FPGA.

A method is presented for parsing an SQL query and deriving from it a reconfigurable datapath consisting of query processing hardware modules. Also, a discussion is given on how to manage tables in stored in the FPGA near memory.

A set of query processing hardware modules are provided that each can implement a number of different SQL operators. Through the use of a *module initialization* protocol developed in this project, configuration registers in the modules can be altered after they have been placed on the device.

1.3 Chapter Overview

This report is organized into seven chapters and one appendix. Following is a brief description of each chapter:

Chapter 1, Introduction
In this chapter the background and motivation for developing a database accelerator is presented. An outline of the desired system is described. Also, the main contributions of this thesis are stated.

Chapter 2, Maxeler System Description
This chapter gives a description of the Linux workstation that was used in this project. The hardware was supplied by Maxeler Technologies. A description of the Maxeler hardware and software is given, and at the end of this chapter follows a brief summary of the changes that will have to be made to it.

Chapter 3, Reconfiguration
An introduction to the methods and tools that are available for partial reconfiguration. There is a discussion throughout the chapter on the advantages and disadvantages of using different methods and tools. We will also decide which partial reconfiguration methods and tools to use.

Chapter 4, Concepts and Design Decisions
This chapter presents the concepts and methods used in the design process. The aim of this chapter is to give the reader a deeper understanding of the different parts of the system, and how they relate to each other. It also explains how and why principal design decisions were made.

Chapter 5, Implementation
This chapter gives a presentation of the software and hardware that was implemented. Notable obstacles that were faced during the design process are highlighted and explained. The implemented hardware and how it relates to the concepts described in Chapter 4 is reviewed.

Chapter 6, Results
This chapter attempts to benchmark the accelerator. We look at a case study where the accelerator is running a test case query. From the results we identify two different bottlenecks that govern the performance of the accelerator.

Chapter 7, Conclusion
The principal findings in this thesis are summarized and presented.

Appendix
The appendix contains the most important source code and scripts. The software developed in this project consists of more than 2500 lines of code, and the VHDL code is also thousands of lines long. Thus, only the crucial components needed to reproduce the results are included in the appendix.

Chapter 2

Maxeler System Description

Database acceleration required high-performance I/O to both local memory and to the host computer. For this project, a MAX3 development system from Maxeler Technologies was selected. The system consists of a fast PC and a built-in FPGA board featuring a large Xilinx Virtex-6 FPGA. The FPGA is surrounded by 24GB of local DDR-3 memory (up to 48GB is possible) and the board is integrated in the PC by using a fast PCI-express interface. The large and fast memory combined with the provided fast host PC interface makes the MAX3 system an ideal research platform for this thesis.

Maxeler Technologies provides hardware solutions for accelerating computer intensive software routines. The hardware used in this project was acquired through the Maxeler University Program (MaxUP).

The Maxeler design flow is based on their Java to hardware compiler. It allows the design of complex stream processing hardware implementations using a modest amount of Java code. These Java representations of hardware can be run in a simulation environment eliminating the need for RTL simulators, and reducing the amount of hardware builds needed in the development phase.

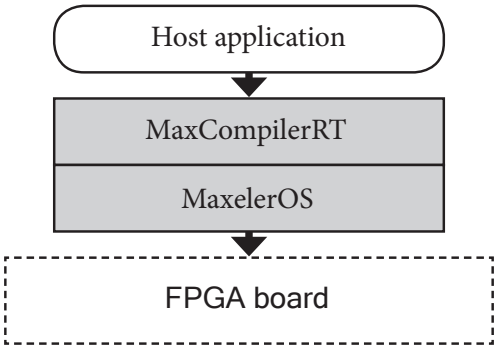

Figure 2.1: The Maxeler software stack.

A program that is accelerated by the Maxeler system is referred to as a *host application*. The run-time environment is called the *MaxelerOS*. This run-time environment is interfaced by the host application through a program library called

the *MaxelerRT* as shown in. The Maxeler software stack is illustrated in Figure 2.1.

2.1 The MaxWorkstation

The hardware provided is basically a desktop computer running the Linux distribution *Centos 5* with a PCI-express card mounted inside that contain their acceleration hardware. This PCI-express card that is called *MAX3* has a Virtex-6 XC6VSX475T FPGA and 24GB of Dynamic RAM (DRAM) memory mounted on it. The FPGA is one of the largest devices available today, and has 476,160 logic cells and 38,304 kb of Block RAM (BRAM).

The DRAM memory is connected via six SODIMM sockets to the FPGA, and the maximum supported memory size is 48GB. Maxeler provides a high performance memory controller that accesses all six 64-bit memory modules as one 384-bit wide channel.

At the maximum DRAM frequency of 400MHz at double data rate, it gives the theoretical maximum read/write bandwidth of $384/8 * 2 * 400\text{MHz} = 38.4\text{GB/s}$. For interfacing the host computer with the FPGA board a PCI-express interface is used. The maximum speed of this connection is 2GB/s in each direction.

2.2 MaxCompiler

The Maxeler Java Compiler is a Java to hardware compiler that is specific to the Maxeler design flow. The algorithms that need to be accelerated must be written in Java code using a special Java library. This Java code is then compiled into VHDL code by the MaxCompiler, and the VHDL code is in turn compiled to hardware by the regular Xilinx design flow. A slightly modified version of the Eclipse IDE, the MaxIDE is supplied to ease the Java coding process.

This Java code uses valid Java syntax, but is in reality a simplified hardware description. It can describe the behavior of stream computation logic accurately, but on a high level of abstraction. It is not a Register Transfer Level (RTL) description like VHDL or Verilog.

The hardware components that accelerate software functions in the FPGA are referred to as *kernels* or *nodes*. Several kernels can be run in the accelerator in parallel, and data streams can go to, from and between each of these.

Figure 2.2 shows the Java code for a simple dataflow kernel and its corresponding data flow graph. The Java data types `HWVar` are in fact hardware register values that are given by the relative offset in the data stream. This particular implementation calculates a three point average value from a moving window on the data stream, and outputs it to the host application.

With the release of MaxCompiler version 2011.3, it is also possible to include custom VHDL or Verilog code and let this take the place of a kernel. This is referred to as a *custom HDL node*. The MaxCompiler also generates streams to and from custom HDL nodes. These streams can have either a push interface with `stall` and `valid` control signals, or a pull interface with `empty` and `read` signals. The control logic for streams is auto generated by the compiler.

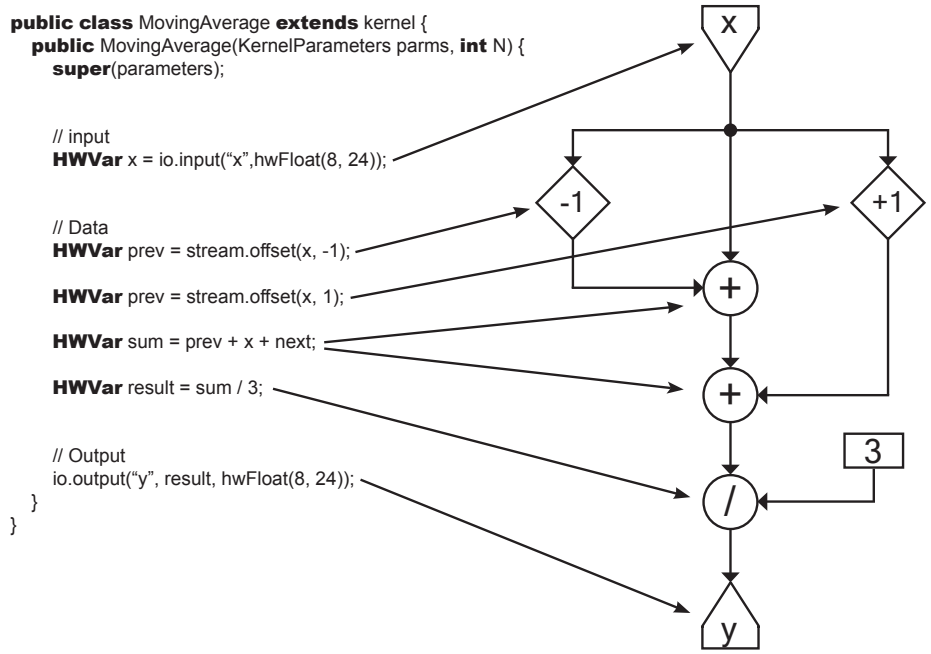

Figure 2.2: Java source code and corresponding data flow graph for a simple kernel. Example from Maxeler R&D brochure.

In this thesis the MaxCompiler was used to generate the main communication infrastructure. It was used to connect the software part of the query accelerator to the FPGA-near memory, and to set up the PCI-express streams to and from the FPGA. The SQL operators themselves are implemented in VHDL.

2.3 MaxelerRT

The MaxelerRT API (Application Programming Interface) is a C library that is used by the host application to interface with the FPGA and the MAX3 card. An application that has functions that are accelerated by hardware kernels must use this library to control the execution of these accelerated functions.

The library has functions for configuring and resetting the FPGA. It has functions for streaming data to and from the FPGA and to and from the memory. The streaming of data can be done in several different ways. There is the synchronized blocking function call, and there are also asynchronized input and output options that allow the host application to have more detailed control over the transfers through polling.

The MaxelerRT library acts as an abstraction level between the hardware and the host application, and any host application must go through it to access the hardware. This library is well suited for our project as it provides a straightforward way of communicating with the FPGA from a C program.

2.4 MaxelerOS

This is the run-time software environment for the MAX3. It comprises the Linux device driver and a daemon process that controls and monitors the MAX3 card and the FPGA. The Maxeler software bundle features several console applications for things like polling the status of the FPGA and configuring bitstreams to it. The MaxelerOS can be utilized by host applications through the MaxelerRT API.

The MaxelerOS hides the memory management from the host application. It provides a simpler command-based interface that the host application can access by using the MaxelerRT API. Command queues and in-between buffering is handled transparent to the user. The MaxelerOS is very beneficial to this project as it takes away the complexity of software and hardware flow control and scheduling, allowing us to focus on the tasks of interest.

2.5 Adding Custom Hardware to the Maxeler System

To build an FPGA system with a reconfigurable region it is necessary to have the VHDL code of both the static part of the system and for the partial modules. The static part of the system is present at any time (e.g. the memory controller) while the partial modules located in the reconfigurable region alter during run-time.

The Maxeler design flow is hidden in a Makefile environment, and the first step was dissecting this Makefile to extract the commands used to generate the system. This enables us to add user constraints and change parameters to the Xilinx tools, which is also necessary when defining a reconfigurable region.

The modules have a high degree of customization and are developed in VHDL. A VHDL wrapper component is also needed to connect the partial region to the rest of the Maxeler system. This means that there needs to be a way to include a custom VHDL component into the Maxeler project.

When work on this thesis started, there was no straightforward way to include VHDL code in the Maxeler design flow. The hardware description is meant to be written in Java and compiled into VHDL code by the MaxCompiler. To try to find the best way to insert a VHDL component into the design flow, and to gain random access to the DRAM memory from it, a lot of research and digging into the VHDL code of compiled projects was done.

With the release of MaxCompiler version 2011.3, New Year 2012, support for including custom HDL nodes in the design was added to MaxCompiler. This was obviously a better way to do this, so all attempts to hack the kernel VHDL description ended here.

The Maxeler system is made for accelerating dataflow computations such as 3D convolution. A stream's access pattern to the DRAM is typically controlled by the host computer, and it follows a pre defined pattern. Random access to the DRAM has to be implemented manually in the VHDL nodes.

For storing and updating multiple large tables in the memory some kind of fragmentation handling must be implemented. This was achieved by having a memory *file system* to add an abstraction level between the data storage and the raw memory. While writing data to the file system requires assistance from the host PC, reading from the file system was achieved directly by the FPGA.

Chapter 3

Reconfiguration

Partial run-time reconfiguration in FPGAs refers to the act of replacing parts of the logic on an FPGA without a complete reset of its operational state.

In this chapter we provide an overview of the different configuration styles that are available. We also introduce a few tools that can be used for implementing run-time reconfigurable systems. In the process, we discuss pros and cons regarding the different tools and techniques.

3.1 Benefits of Partial Reconfiguration

Reconfiguring parts of the FPGA has several benefits[3]. Some of the most important ones are listed here:

- While the partial region is being reconfigured, the rest of the device can continue to function. For example, a memory controller can be active to issue DRAM refresh commands. Consequently, data stored in DRAM can be used before and after a configuration process.
- It allows the usage of more logic on a chip than would have been the case with a static implementation. Through hardware sharing, a system can be implemented on a smaller device. Using a smaller device reduces the cost of the FPGA and allows for smaller circuit boards. A smaller device will also consume less power, which can be an advantage to embedded systems.
- Partial reconfiguration can be used to remotely deploy updates and bug-fixes. A product held by the end-user could download the partial bitstream and have its FPGA update its core logic by means of partial reconfiguration. This might otherwise have been done by a microcontroller or a second FPGA.

3.2 Configuration Style

A partially reconfigurable system consists of a static region and one or more partial regions. The static region remains unchanged during run-time, while logic in the partial regions can be changed.

The smallest possible granularity of reconfiguration in the Xilinx Virtex family of FPGAs is one resource column wide and the height of one clock region. With

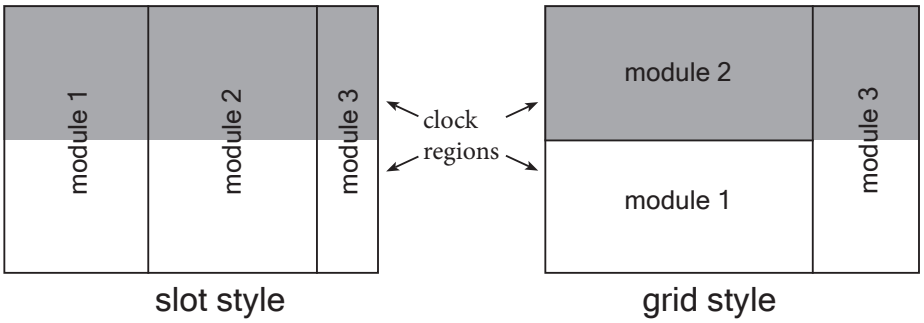

Figure 3.1: Slot and grid style reconfiguration.

this granularity the reconfigurable area (partial region) can be managed in several different ways[6]. Single island style uses one partial region while a multi-island style implementation has several partial regions.

A partial island may be reconfigured as a whole, or it may be split up into smaller adjacent regions that can be reconfigured independent of each other. A vertical division of a partial region is known as *slot style* reconfiguration. Splitting the slots horizontally into modules as small one clock region in height is also possible. A vertical and horizontally split bounding box scheme is called *grid style* reconfiguration. Figure 3.1 shows an example partial island spanning the height of two clock regions marked with grey and white.

The configuration style has impact on how module bounding boxes can be adjusted. In *island style* reconfiguration, only one module can be executed exclusively in a reconfigurable region regardless if there is space to host another module at the same time in the region. However, *slot* and *grid* style reconfiguration permit to host several modules with different resource requirements as illustrated in Figure 3.1. Due to the streaming nature of the SQL processing, slot style reconfiguration was chosen for this thesis.

3.3 Interfacing Static and Partial Regions

For two modules to be interchangeable they must have identical input and output signals. That is, for a module to fit into the interface between the static and partial region these signals must use the same physical wires at the static-partial border. There are at least three ways to accomplish this[6]:

- *Bus macro* approach. By placing a hard macro (routed netlist) at the border between the static and the partial region in such a way that half of it resides in either region, an interface can be created to the partial region. The bus macro consists of two LUTs and wires routed between them.

  Using this macro ensures that the same physical wires are used when crossing the border, but it wastes two LUTs for every signal. It also adds additional latency to the interface.

- *Proxy logic* is used by the latest version of the partial reconfiguration tool flow from the FPGA vendor Xilinx. In this technique, anchor LUTs are placed at desired locations in the partial region before routing the static region. The static system is routed and the wires cross the static-partial border anywhere.

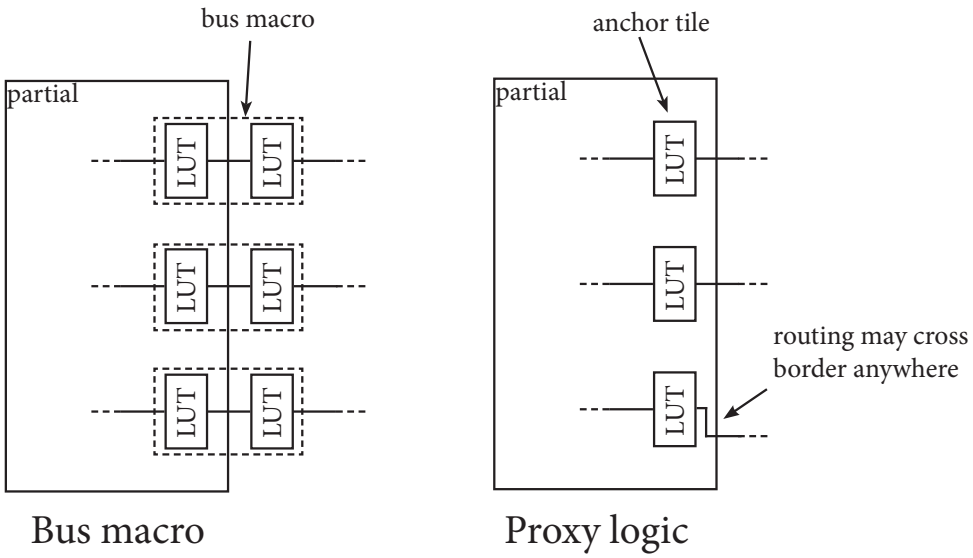

Figure 3.2: Bus macro and proxy logic concept.

When building the modules for the partial regions, the routing in the partial region belonging to the static build is kept. The modules inputs and outputs go through the anchor LUTs. The router is free to route the border crossing signals using any wires and consequently the routing becomes different with every run.

A drawback when using this technique is that the modules may not be relocated to different partial regions due to unconstrained border routing. Another drawback is that all modules have to be rebuilt for every build of the static system. Also, the proxy logic consumes one LUT per border crossing signal.

- *Connection macros* are hard macros like the bus macros, but they serve a different purpose. Instead of having the border crossing wires part of a macro, this technique forces the router to use specific wires by using *blocker macros*. *Blocker macros* are pre-routed hard macros that are designed to force the router to use specific wires for specific nets.

  In the VHDL code for the static region, border crossing signals are bound to connection macros that reside in the partial region. So far this is similar to the use of proxy logic. But before the router is started a blocker macro is placed over the partial region. This blocker macro has dummy routing that occupy all available wires in the partial region except the wires that we want the router to use to reach the connection macros. The concept is illustrated in Figure 3.3 on page 24.

  The connection macros work in conjunction with the blocker macro to create a predictable border crossing interface. With this procedure, the router is left with only one possible solution to reach the connection macros, and is thus forced to take it.

  The partial region is blocked before the static region is routed, and the blocker is later deleted from the fully routed design. When building the partial modules the static region must be blocked, and the modules later cut from the netlist.

This approach has several advantages:

- The LUTs in the connection macros are meant to be overwritten when the partial region is reconfigured, so it uses no additional LUTs for border crossing signals.
- Modules can be relocated to any partial region that shares this interface as long as it fits the resource footprint (see Section 4.1).
- Several modules can be configured onto the same partial region using either slot or grid style reconfiguration.
- Modules can be built independently of the static system.
- A module can be built on a smaller device to save build time as long as the resource footprints match.

3.4 Partial Reconfiguration Tools

For implementing run-time reconfigurable systems, tools are needed for floorplanning the FPGA's resources, and for generating the communication infrastructure. The following list gives an overview of the tools made to assist and automate these processes:

- *PlanAhead* is a graphical design analysis tool from the FPGA vendor Xilinx. It can be used to analyze and floorplan Xilinx projects. It is also Xilinx's tool for integrating partial reconfiguration in their design flow. PlanAhead can be used for implementing the static system as well as partial modules.

  The latest version of PlanAhead (v14.1) uses proxy logic to constrain the routing for the interface with the partial region. Previous versions of PlanAhead used bus macros.

  Due to the proxy logic integration method, PlanAhead is only suitable to implement island style reconfiguration. This would be too limiting considering the SQL application of this thesis.

- *GoAhead*[6] originates from the ReCoBus-Builder project developed at the University of Erlangen-Nuremberg. It is a reimplementation of the graphical application from this project.

  GoAhead supports scripting, and can in addition to the graphical interface also be run from the command line. GoAhead supports single island style as well as multi island style reconfiguration, and it can be used for slot as well as grid style reconfiguration. Bus macros or connection macros can be used for interfacing the static region.

  The tool can generate VHDL templates for connection to the macros and print constraint for the User Constraints File (UCF). It also generates hard macro blockers that contain dummy routing. This is used to block routing in the partial region when implementing static system as shown in Figure 3.3 on page 24. The connection macros and the blocker macro together is what makes it different from the PlanAhead tool.

  Because of the support for slot based reconfiguration style, GoAhead was used for implementing the reconfigurable part of the FPGA.

- *OpenPR*[14] is an open source tool around the Xilinx vendor tool that provides features similar to GoAhead. However, the software does not support the Virtex-6 FPGA family from Xilinx at the time of writing. A Virtex-6 device was used as a target in this thesis, so OpenPR is not applicable for us.
- *FPGA Editor* is a graphical application that is part of the Xilinx Integrated Synthesis Environment (ISE) tool suite. It can be used for viewing and editing implemented designs from Xilinx projects.

  Rather than implementing a complete system, FPGA Editor allows for small netlist manipulation. From these changes differential bitstreams can be generated. For instance, it can be used to change an `AND` gate to an `OR` gate. The tool is not suited for implementing larger changes. However, the exact physical placement of components and routing can be analyzed with FPGA Editor.

  It can open Native Circuit Description (NCD) files produced by the mapping and routing tools. The editor is useful for finding out if a mapped but not routed design looks correct, before passing it to the router. If the routing fails, viewing the NCD of the partly routed design will often reveal what the problem was. Checking the fully routed design is always a good idea when dealing with partial reconfiguration.

All these tools for partial reconfiguration are based on the ISE tool suite from the FPGA vendor Xilinx. At the time of writing this thesis, Xilinx is the only vendor supporting partial reconfiguration of large capacity FPGAs. However, the FPGA vendor Altera announced to also support partial reconfiguration in all future FPGAs and in their design tools[12].

3.5 Implementation Issues

The proxy logic approach allows routing from the static system to cross the partial region. The problem with this is that modules cannot be relocated to any other location, and that they have to be rebuilt for every minor change to the static system.

Blocker macros may be used together with bus macros and connection macros to block routing from the static region to cross over the partial region. This is beneficial because modules can be relocated and built independent of the static system. However, this comes at the expense of a more congested routing situation around the partial region when building the static system. When the logic utilization in the static region becomes higher and the size of the partial region increases, congestion around the partial region becomes a paramount issue that has to be dealt with.

The blocker macro has to block all wires where we do not want routing when building the static region. It must also leave exactly one route for every signal that should connect to them. If there is no route to a connection macro the router will go on forever or for a long time. If there is more than one route to any connection macro it will result in erroneous routing.

The blocker macro in this project is generated by GoAhead. The GoAhead tool is continuously under development, and tweaking the algorithm that generates this filter is out of the hands of the author of this thesis.

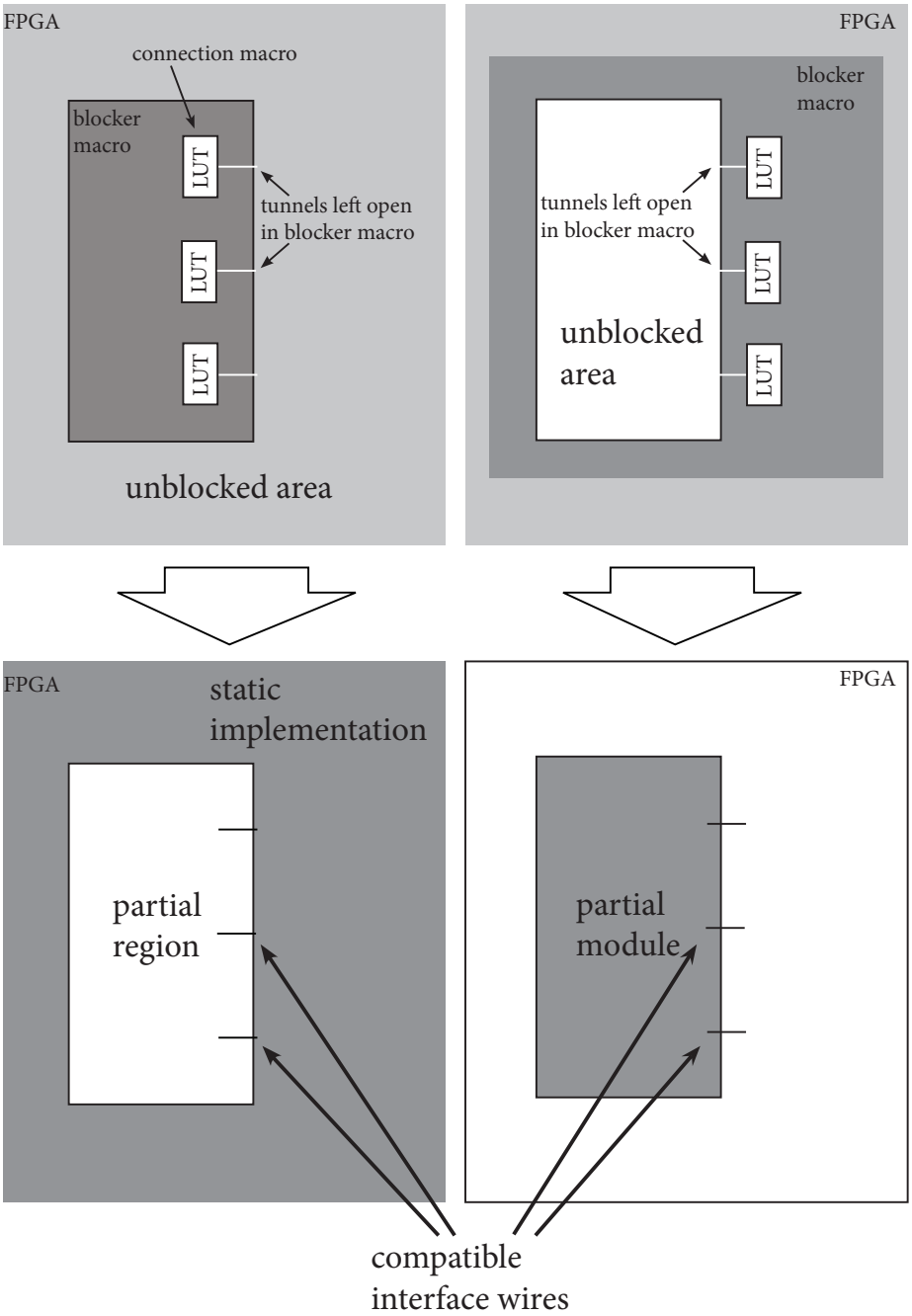

Figure 3.3: The connection macro concept.
Left side: The static system is routed with a blocker macro occupying all wires in the partial region, except the ones we want the router to use.
Right side: For the partial modules, they are routed with a blocker macro 'fence' around the island. This results in matching interfaces for the static system and the partial modules.

Chapter 4

Concepts and Design Decisions

During the process of developing the project there were many decisions that had to be clarified regarding its design and architecture. Each decision having its pros and cons, and every ruling influences the possible subsequent design decisions.

Because of the complexity of the project and starting completely from scratch, some simplifications had to be made. This includes supporting a subset of the SQL language. The supported SQL operators include the `FROM` clause and the `WHERE` clause as well as logical and relational operators.

Chapter Overview In this chapter an evaluation of the components needed for a more complete system is given. We shall consider concepts and design decisions for a possible implementation of a database accelerator.

At first we examine the *floorplanning* of the FPGA. To provide the reader as far as possible with a top-down description of the system it seems most appropriate to start with describing the layout of the system on the device. This gives us an overview of the physical implementation of the system and hopefully eases the understanding of the following sections in this chapter.

Since the system acts transparent to the client there is the need for a way to insert it between the client application and the regular database as shown in Figure 1.1 on page 12. Therefore we will next discuss the de facto way of doing this, with the use of the ODBC protocol. Then we take a look at two open source database implementations and argue why one or the other might be used. Following is an introduction to query plans generated by conventional databases and an explanation for why they cannot be used.

A description of the Maxeler middleware is given next. This software is needed to interface the software part of the database with the Maxeler hardware. Following, a walk through the process of deriving a datapath consisting of modules from the query. In Section 4.5 we can see that there are two different classes of modules and we discuss the use of intermediate signals in between them. Then we look at the *module architecture* options that are given by the query types we target for acceleration. We will decide on implementation options and present an overview of the circuitry.

Finally, we will investigate how to store tables and how to stream them to the modules. We will see that the storage scheme is closely related to the complexity of the static part of the hardware. In a functional accelerator there is the need to handle the storage of multiple tables in the memory of the MAX3 card. Therefore we propose a hardware *file system* for this task in Section 4.8.2.

4.1 Floorplanning

Floorplanning in FPGA development is the process of defining where on the FPGA different components must be placed, and which IO-pins that should be used. This information is defined by the developer and results in user constraints to be followed during the place and route phase by the tools.

The physical implementation of the database accelerator is divided into static and partial regions. The static region is loaded onto the device at startup, but remains unchanged for as long as the accelerator runs. The partial region is where the reconfigurable datapath is implemented, the area in which the modules will be interchanged. Floorplanning in our case involves defining where the partial regions will be, and how they interface with the static system.

In our case, the User Constraint File (UCF) already contains information about IO-pins and other constraints generated by the Maxeler system. This part is left untouched. The *map* program can typically determine the placement without user interaction, but in our case some additional constraints are needed. This includes prohibit constraints for the partial region, such that no static logic is placed in the reconfigurable part that will host the reconfigurable datapath. Furthermore, it is necessary to constrain the routing between the static system and the partial region.

After the mapping phase comes the *place and route* (PAR) phase, that contradictory to its name does not place the components. That was done by the mapping tool. The *par* program, popularly known as the *router*, routes the interconnect wires between the already placed components.

While there exists prohibit constraints for the placement of FPGA primitives, there are no equivalent constraints available for the router. This means that there are no constraints to guide PAR to not use routing resources within the reconfigurable region for implementing the static system.

Therefore we must use *blocker macros*. The blocker macro is a hard macro which consists of a pre-routed dummy component that can be placed on various locations on the FPGA. By placing a blocker macro in the netlist of the placed device description, but before running the PAR phase, the router is forced to ignore the interconnect wires already occupied by the blocker.

These static wires can be deleted after the routing has finished, yielding an area without routing that can be used as a partial reconfigurable island. In order for the modules that are configured to this area to be able to communicate with the static system, the input and output wires must coincide in the physical implementation. That is, the modules and the datapath in the partial region must use the same type of input/output wires, allowing the modules input/output to fit into the common wire pattern. This is accomplished by selectively leaving tunnels in the blocker macros in order to connect to *connection macros*.

Connection macros like the blocker macros are hard macros, but they are assigned

to a specific LUT site, and have a specific instance name. By mapping the datapath through these macro instantiations in the VHDL code, the router is forced to route the signals we want to specific tiles on the FPGA. In the blocker macro only the wires that should be used for connecting to the connection macros are left open. This forces the router to use these wires, since there is no other option. And this leaves a predefined, uniform routing pattern in the partial region.

A summary of the process of creating the static region is as follows:

1. Define the partial region.
2. Prohibit resource usage in the partial region.
3. Place connection macros at specific slice coordinates *within* the partial region. Connect the interface to the partial region to these instances in the VHDL code.
4. Run synthesis, translate, and run the placer (map).
5. Add wire blocking hard macros to the mapped design, covering the partial region.
6. Route the design.
7. Delete dummy nets belonging to the blocker macro.

4.1.1 Module Footprint

The connection macros use *double lines* for interfacing the static and the partial region. These double lines are horizontal wires that jump to every other switch box which is located at every other resource column. They consequently connect to every other CLB), BRAM or multiplier column. This means that the modules and the macros for the static system must be placed so that they always connect to the same wires, and all modules must be a multiple of two columns wide. If the first possible module placement is at one column of resources on the FPGA, the next possible placement would be two columns ahead.

On the Virtex-6 FPGA, the resource columns on either side of the FPGA follow a specific pattern. Figure 4.1 show this pattern where `L` stands for a CLB column with LUTs, `B` for a BRAM column, and `D` for a column with multipliers that are called Digital Signal Processing tiles (DSP48) by Xilinx.

Given a module that is 4 columns wide, this pattern gives us a total of eight possible module footprints. The first and last module placements in the figure have the same footprint, and thus the same module implementation can be used for both sites.

If a module shall be freely relocated to different positions, it requires eight different physical implementations (place and route steps) for the same module. In this example, we assume to use only the routing resources of the BRAM and DSP48 columns. As the routing architecture is identical for BRAM and DSP48 columns and because the routing is encoded identically for there two resources, wildcarding can be used. This allows for example to place a module with a '`BLLL`' footprint at a position providing '`DLLL`' resources[1]. With this trick, four physical implementations would be sufficient to support any placement position in the system.

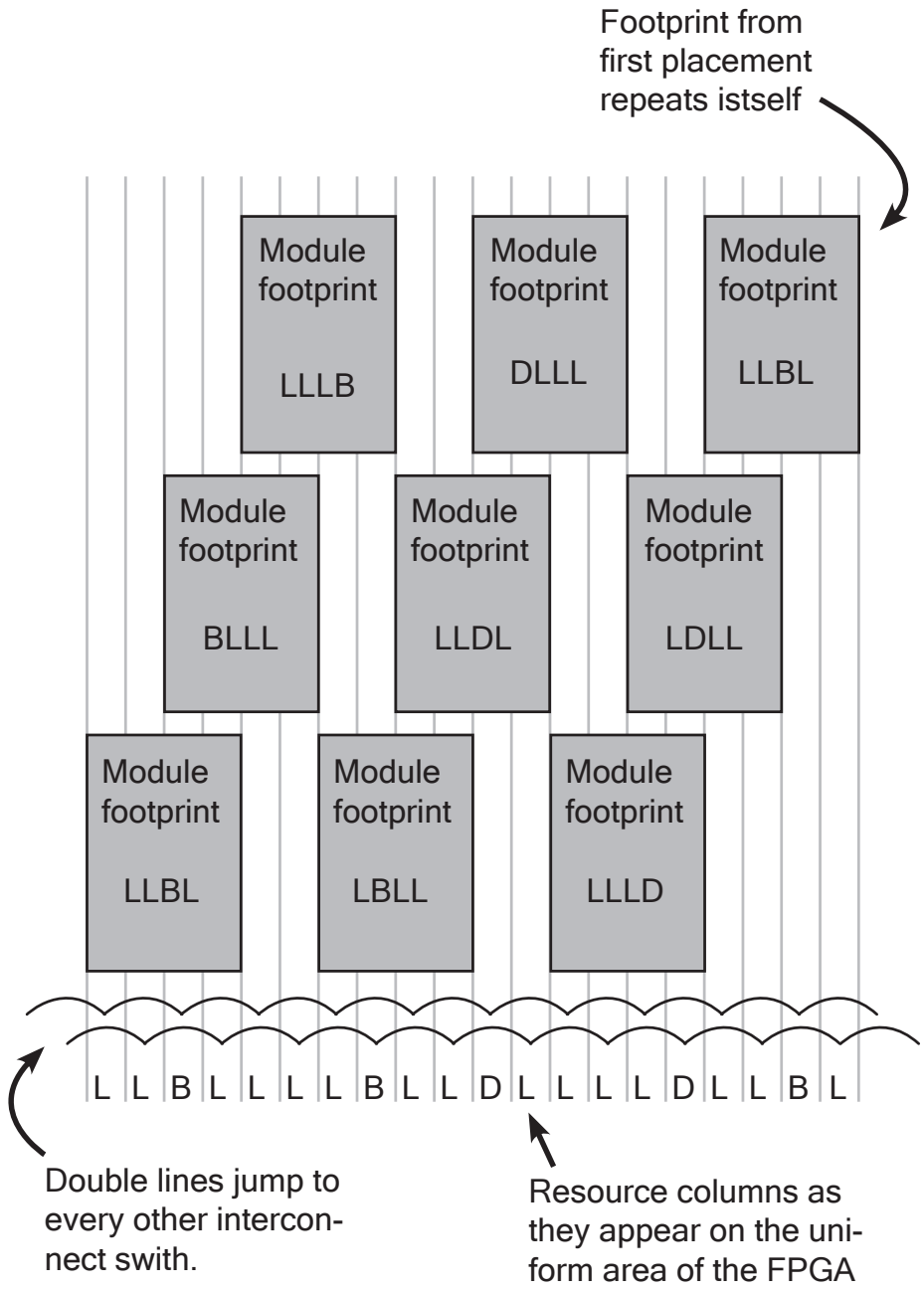

Figure 4.1: Possible placements and footprints for a 4 column wide module. `L` = *LUT,* `B` = *BRAM,* `D` = *DSP.*

4.1.2 Single Island Style

The partial region will host the modules that are reconfigured for every particular query. The interconnected modules form a horizontal datapath with the inputs on one of the east-west sides, and the outputs on the other side. A single island style layout of the FPGA is illustrated in Figure 4.2. Here the data stream originates from the DRAM and is output to the host through the PCI-express interface.

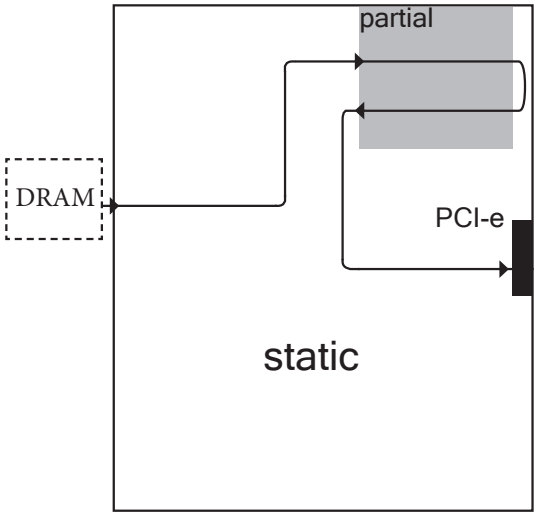

Figure 4.2: A single island style layout. Example with a stream originating in the DRAM memory, passing through the partial region, and to the host computer through the PCI-express interface.

Figure 4.3 shows the concept of routing a stream that enters the partial region on the west side, goes through the modules and exits the partial region on the east side. A problem with such a design is that the output signals have to be routed around the partial island and back again to the area around the input side of the partial region. This is because the inputs and outputs are unsynchronized in terms of clock cycles, but the logic is still related.

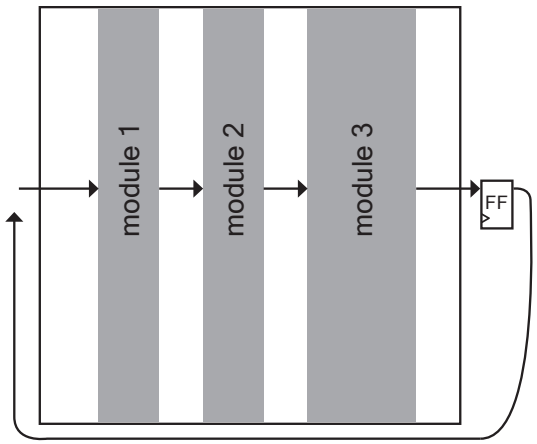

Figure 4.3: Routing output signals around the partial island.

The static output system has to know whether a query is going on or not. This creates two obvious problems. The output signal vector will be more than 500 bits

wide, as we shall see later in this chapter. This means that the vertical routing on the output side of the partial region will consume a large area.

The other obvious problem is that since the static routing is prohibited from going through the partial region, the border around it will already be crammed with other logic and routing. If the design would be routable at all, this would result in a long path for the output signals, and thereby a low maximum clock frequency.

If the modules only do filtering and never alter any records, the output data signals are not really needed. If this is the case, only a one-bit signal, the valid or not valid record signal is needed. Then a solution could be to buffer the input signals in a FIFO on the input side of the partial region, and only route back the valid or not valid result from the output. However, this would destroy the concept of our stream through database. And more importantly, it would prevent us from adding record altering modules like arithmetic operators.

A better and more realistic solution is to add tunnels to the modules and partial region to allow the output signals to return to the input side *through* the partial region. This can be done by adding the return wires as part of the hard macro blockers used when the modules are created. Every connection macro has the same amount of inputs as outputs, and the double lines they use have the same amount of eastbound wires as westbound wires.

The island on the right side of Figure 4.4 shows this kind of routing for a partial region located on the east side of the chip. The input signals go into the first module, then from the output of each module to the input of the next one. From the last module the signals go through a register to ensure that the timing is valid. From there they return through the westbound double lines that are present in every module.

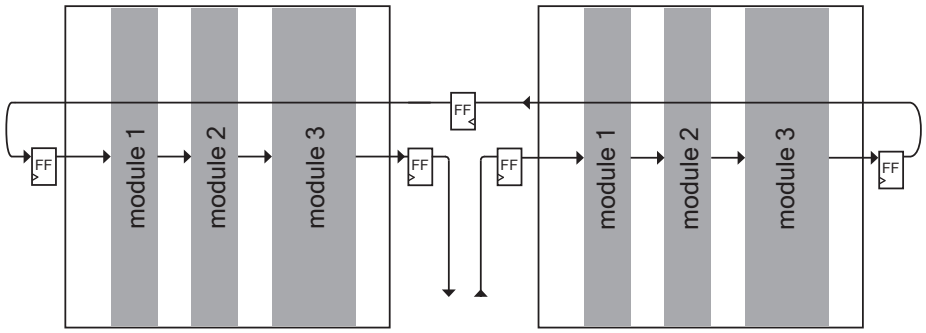

Figure 4.4: Floorplanning streams for the west and east side partial regions. The use of this scheme allows the use of identical modules for partial regions located on both the west and the east side of the device.

4.1.3 Multi Island Style

When using multi island style floorplanning, the output from one partial region is fed into the input of the next one. For partial regions located far away from each other, multiple pipeline stages can be used to meet the timing requirements of the design. In this way, the datapath functions as a single large partial region. But the query planning software must be aware of the boundaries of a partial region, because modules cannot be divided among two different partial regions. This has to be integrated into the query planner running on the host PC.

For a second partial region located on the west side of the chip, the same modules can still be used. The trick is to have the data streaming through the modules always in the same direction to avoid having to build mirror modules. In our system, this means that the datapath always streams from left to right. So since the datapath in the east partial region was eastbound we must ensure that is also eastbound in the west partial region. This is done by first streaming the data through the modules to a register on the far west side of the partial region (left side of Figure 4.4), and then back through the modules in eastbound direction.

Deciding on which module footprint to configure on a given place in the datapath reassembles a pattern matching task. The resource columns are denoted by `L` for LUT, `B` for BRAM and `D` for DSP. In addition, we introduce another symbol `X` that denotes the border between partial regions. When the query planning software searches for a possible placement footprint for a module this icon will not fit any footprint, because this sentinel symbol is never present in any modules footprint. The query planning software will then jump to the next set of columns, and eventually the query planning software will place the module on a location past the sentinel icon and into the next partial region.

Each of the four partial regions have the footprint '`LLBLLLLBLLDLLLLDLLBL`', as shown in Figure 4.1 on page 28. All four partial regions are modeled by one combined string. This string starts with partial region 1, then region 2 and 3, and finally partial region 4. Between the different regions, an '*X*' symbol will be added:

'`LLBLLLLBLLDLLLLDLLBL`*X*`LLBLLLLBLLDLLLLDLLBL`*X*`-`
`LLBLLLLBLLDLLLLDLLBL`*X*`LLBLLLLBLLDLLLLDLLBL`'

By modeling the resources this way, the finding of valid module placement positions is a string match of the module resource string on the string resulting from the four partial regions. Partial modules are placed one after the other, and each module will be placed at the entire leftmost position on the remaining string. This algorithm scales with linear execution time regarding the resource string length and linear with respect to the number of modules.

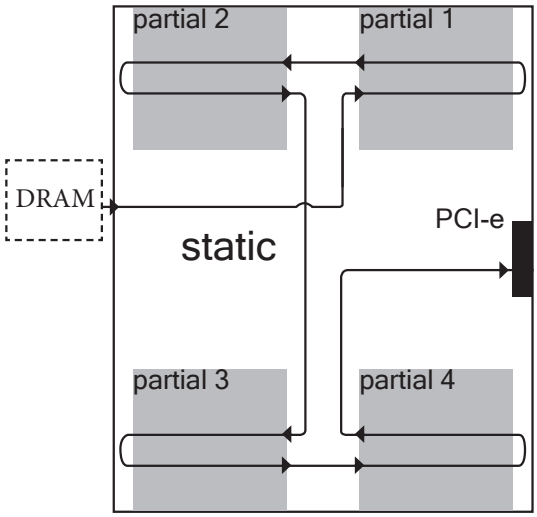

Figure 4.5: Multi island style. Example with stream going from the DRAM memory, through all partial regions, and to the host computer through the PCI-express interface.

Using the routing scheme described above, a four island implementation could have

the overall layout and routing shown in Figure 4.5. The wide data stream originates from the DRAM input pins. It then enters the first module at the first partial region that is located on the north east side of the chip. The stream then takes the horizontal path to the partial region in the north west corner, the stream routing is according to Figure 4.4 on page 30. After this, the stream passes through enough pipeline stages for it to reach the southern part of the chip without timing violations. The last two partial regions on the southern part of the chip complete the datapath. With this routing scheme, the four partial regions can logically function as one larger area.

4.2 Connecting the Accelerator to a Software Database

Open Database Connectivity (ODBC) is a standard software API for communication between database applications and databases. This API acts as a middleware which allows any application to access any database as long as they are both compliant with the ODBC standard, regardless of which operating systems and query languages they use. The ODBC API was originally developed by the SQL Access Group, but later adapted by Microsoft which named the standard ODBC. Microsoft is the maintainer of the ODBC standard, and the latest API version is ODBC 3.8. For this project, the ODBC API is especially interesting because it will not limit the use of the database accelerator to a specific database. If the database accelerator utilizes ODBC, it could in theory be used in conjunction with arbitrary applications and databases.

All major databases ship with an ODBC driver, responsible for translating ODBC calls into the native SQL accessing method for a particular database. Applications do not talk to these drivers directly, but through an ODBC driver manager installed on the operating system that is responsible for loading and forwarding queries to the ODBC driver requested by the application. The Linux ODBC driver manager "unixODBC", as well as the ODBC drivers and database implementations for the popular databases MySQL and PostgreSQL, are all open source and GNU public licensed. This makes them ideal for this project as they can easily be studied, and altered to suit the project if necessary.

In order to incorporate the accelerator with the database, an ODBC proxy is needed. The main task of the proxy is to direct queries to the appropriate execution unit, either the master database, the database accelerator, or both. In operating systems, ODBC drivers manifest themselves as dynamically loaded shared object libraries (DLLs). To allow data to pass through the proxy, it must be installed on the system as an ODBC driver with its own identifier name. The application then requests the driver using the name of the proxy.

In turn the proxy then calls the requested functions in the ODBC driver belonging to the database and passes the function call to it. This way all ODBC calls have to pass through the proxy, and can be intercepted and altered. Data flow is as shown in Figure 4.6. The proxy server is implemented as any other ODBC driver, it is a DLL, and thus functions in it are only executed when called from the application. This means that it cannot provide services in between operations, like for instance loading tables into the accelerator's memory independent of queries. For this reason it might be practical to have the proxy server talk to an always running daemon using shared memory, and have this daemon talk to the accelerator. This complicates the design of the proxy, but gives flexibility in manipulating the accelerator independent of queries.

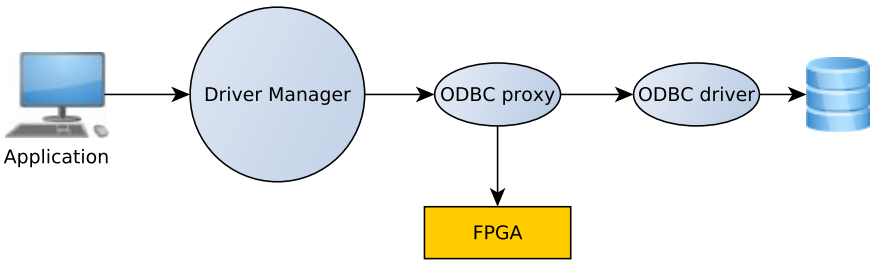

Figure 4.6: Data flow through the proxy.

4.2.1 Executing Queries with ODBC

Using ODBC, when an application attempts to connect to a database, it has to do so through the driver manager by stating the name of the requested driver in a handshaking phase. The application then keeps a pointer to a memory location where a connection handle is stored. This connection handle is then subsequently used by the application to communicate with the ODBC database driver.

There are several ways an application can execute a query and retrieve the results, but the basic approach consists of five calls to the ODBC driver[9].

1. `SQLAllocStmt()`. At this point, a statement handle is requested from the ODBC driver. This is a pointer to a memory location containing driver specific information about the query that is about to be executed. A *query plan* is an ordered set of actions required to execute a query. More on this is explained in Section 4.3.1.
2. `SQLPrepare()`. The function sends the actual SQL query along with the previously allocated statement handle to the database. At this stage the database compiles the query to form the query plan, ready to be executed.
3. `SQLExecute()`. This is the command for the database to execute the query that was prepared in the previous step. The reason for separating the execution step from the prepare/compile step is the case of a query being executed several times, which saves time by storing the precompiled query plans in the database.
4. `SQLFetch()`. This call advances the "cursor" forward, denoting the next row to be retrieved from the results data set generated in the previous step. The term cursor reflects the blinking cursors found in text editors, but in this context it means how much more data should be returned in the next step.
5. `SQLGetData()`. Here a single column, which has to be specified as an argument to this call, is returned from the result set that was fetched in the previous step. A pointer to a memory location to store the results in has to be sent as an argument.

This is the basic approach for executing a query, but this approach only fetches a single table entry at the time. Of course, this only becomes interesting when talking about bulk transfers of result tables, which is why ODBC offers several methods of reducing overhead. Here are two relevant ODBC functions necessary for increased performance:

- `SQLExecDirect()`. Merges the `SQLPrepare()` and `SQLExecute()` calls into

one that is executed immediately.

- `SQLFetchScroll()`. This function moves the cursor to include any number of rows. It is used to specify any number of rows to be transferred in the next `SQLGetData()`. This applies to ODBC version 3.x and later, for earlier versions the `SQLExtendedFetch()`, a similar function, is used.

Another useful function is the `SQLCancel()` function. It can be used to terminate an SQL statement that is being processed. This function may be used in this project by the proxy to terminate query execution in the master database, in cases where the accelerator finishes before the master database. Sending queries to both the accelerator and the master database, and returning the fastest response, is a way of ensuring that the system will never be any slower than if the accelerator was not present.

As for performance with ODBC compared to the regular call level interface of databases, it seems that the added overhead is not significant. One benchmarking test[8] showed that the performance differences when connected to an Oracle database with ODBC compared to the regular Oracle call level interface was less than 5% in 80% of the tests, and less than 10% in the remaining 20% of the tests.

4.2.2 Drawbacks of using ODBC

Since ODBC is a call level interface, and does not have any kind of network support, the ODBC driver and proxy must be on the client system (Figure 4.7). This undermines what was the main reason for using it in the first place; to achieve protocol independence. As long as client and server run on the same (Linux) system, there are no issues. But if the client is running on another Linux or Microsoft system, this is only achievable by using an ODBC-ODBC bridge. There seems to be only one in existence, a commercial non-free product developed by `http://www.easysoft.com/`.

Also, there seems to be very few benchmarking tools that can be used to test an arbitrary ODBC compliant database on Linux. Among the few that can be used, DBT-3 from the Database Test Suite project[15] may be the most relevant one. This is because it uses the TPC-H performance metric which focuses on executing queries on large data volumes with high degree of complexity[13].

4.3 Back-end Database

When choosing which master database to use, the main criterion is that it is an open source implementation that runs on Linux. It has to be well documented and maintained. There are two obvious alternatives; the MySQL database and the PostgreSQL database. Both ship with a variety of console tools and a GUI tool for managing the database.

Both MySQL and PostgreSQL have ODBC drivers available, but the one for MySQL seems to be maintained better and easier to set up. Considering this, and the fact that MySQL is the most widely used of the two DBMSs, MySQL seems the natural the choice for master database in this project.

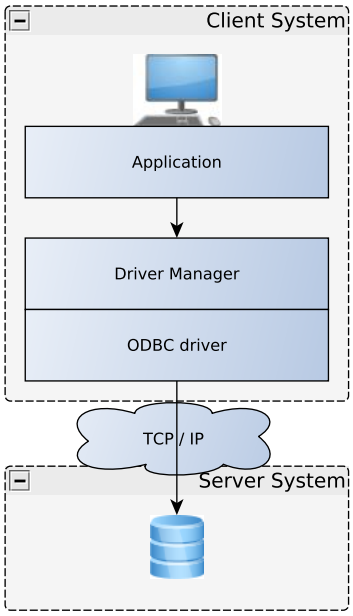

Figure 4.7: Typical ODBC architecture.

4.3.1 Query Plan

In order to determine the data flow to be set up in the FPGA, a data flow graph would be of interest. Such a data flow graph could perhaps be extracted from the query plans used in conventional DBMSs. Query plans are generated in databases prior to execution and show the call hierarchy required to execute a specific query. This also shows what parts of the query can be executed in parallel, and in which order the sequential parts has to be executed. Query plans can be represented textually or graphically. For example, using the pgAdmin III graphical SQL tool for PostgreSQL, the following SQL query generates the query plan shown in Figure 4.8.

```
SELECT
   orderinfo.customer_id
FROM
   orderinfo
     INNER JOIN customer
       ON customer.customer_id = orderinfo.customer_id
ORDER by shipping
```

Examining ODBC statement handles during debugging shows no indication of a query plan being present in it. If this was the case, query plans should at least be present in the statement handle directly after an `SQLPrepare()` had been executed, and right before the `SQLExecute()` was called. If it is not contained in the statement handle, this could mean two things. Either the query plan is stored internally in the MySQL server, or it is only generated at execution time. Unfortunately, MySQL does not build query plans[2] so that they can be reused. This is probably the reason the graphical database tools supplied with the MySQL server cannot generate a query plan, as the pgAdmin tool that ships with PostgreSQL can do.

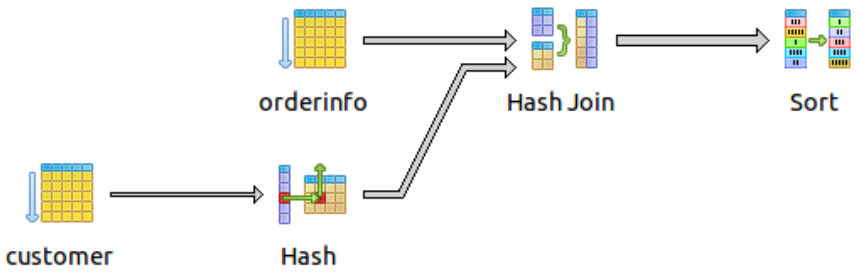

Figure 4.8: An example of a query plan, generated by PostgreSQL.

Using the PostgreSQL database server as master DBMS poses some difficulties due to limitations in the Linux ODBC drivers available for it. The driver is community maintained, and poorly documented. Quite some time has been spent during this project trying to get it working, unsuccessfully. Even if it was working perfectly, there is no guarantee that the statement handles it uses contains the query plans used internally in the PostgreSQL server.

As it is not easily possible to extract a query plan from the master DBMS, it has to be created internally in the proxy. To execute the query, a parser that translates the text representation of a query into a tree of nodes is needed. In this tree, each node is a step that is required to execute the query.

There are many ways to execute most queries. Since SQL is a declarative language it only describes the result set, not how it should be executed. The advantage of using FPGAs is the possibility of a high degree of parallelism, therefore such a query optimizer should opt for having the greatest number of parallel datapaths as possible.

The query plan is interesting for determining the steps that need to be completed for complex queries that involve data aggregation, or that has sub queries in it. However, for building the reconfigurable datapath the query plan from a regular database is not of much help.

Regular query plans serve to determine the order of the operations that have to be executed. It is useful to know that an aggregate function must be performed *before* a data filtering operation from the `WHERE` clause, or that a sub-query must be performed first. But this is not of any help when it comes to finding out what kind of modules that are needed to execute the `WHERE` clause itself. Regular query plans only give us the principal order of operations.

Due to the complexity of the query plan generation and the complexity of the rest of the system, this thesis is restricted mainly to the `FROM` and `WHERE` clauses.

4.4 Maxeler Middleware

For interfacing with the FPGA, Maxeler provides a run-time environment called the MaxelerOS. They also provide a bundle of programs to monitor and configure the FPGA. To interface with the MAX3 card a program must use C library calls from the MaxCompilerRT API. There are two ways to pass information to a custom HDL node on the FPGA; streams and scalar inputs. Both have customizable bus

widths. The streams have flow control and can be used to transfer data, while the scalar inputs can be used to pass a single operand to a custom hardware module (called *HDL node* by Maxeler).

In this project, streams are used to stream data to the HDL node or to the DRAM. A stream is also used by the HDL node to tell the host application that a query has finished and how many records were returned. That is because there is no such thing as a scalar output. Scalar inputs are used to control the HDL node from the host computer. This includes things like resetting, setting the DRAM address, and initiating executions.

There are some relevant MaxCompilerRT functions that are used in this project:

- `max_set_scalar()`. Input a C variable to a register in the kernel or custom HDL node.
- `max_queue_pcie_stream()`. Transfers a statically or dynamically allocated buffer over the specified stream.
- `max_sync_pcie_stream()`. Blocks until a stream has completed.
- `max_setup_pcie_stream_ringbuffer()`. Sets up a ring buffer with read and write markers. In this project it is used to retrieve the results from a query in real time. The reason for using this instead of regular streams, is because the amount of data to receive is not known in advance. Therefore we use ring buffers and a second flow control stream in parallel with the data output stream to tell the host computer that a query has completed.
- `max_reset_device()`. Resets the device. The HDL node uses a different reset by scalar input.
- `max_run()`. A wrapper function that resets, transfers the data and syncs the stream(s). This is used in this project to load tables to the DRAM from the host computer.

4.5 Datapath Generation

The matter of deriving circuits from queries has previously been explored by others, relevant research includes the Glacier compiler[7]. This PhD thesis focuses on the use of FPGAs to accelerate queries on real-time streams of stock market data. In this thesis, a query to hardware compiler called the *Glacier* compiler is presented. This compiler takes a special algebraic representation of a query plan and compiles it into a VHDL hardware description, which in turn is synthesized into a bit level representation (configuration bitstream) that is loaded onto the FPGA.

This approach cannot be directly adopted for use in this project, because for every new query a complete design flow has to be run, which takes far too much time. Even so, the [7] thesis contains some interesting hardware solutions for common database operations. It also offers a discussion of possible ways to get around the compile time problem. It proposes using fixed processing elements (modules) in the FPGA and rearranging the data flow for every query. Alternatively by using programmable modules in a fixed overlay data flow network, which is particularly interesting.

Implementing a fully featured accelerated database cannot be accomplished in a single master thesis project. However, by accelerating the `WHERE` clause one can actually gain a lot of performance in a database system. This is because it in many

cases substantially reduces the table data that has to be further processed by the database system. For example, the `SELECT` clause defines which columns to output, and the `WHERE` clause is used to filter the rows for the result set.

The first thing we want to support in our accelerator is data filtering. The clause that implements filtering in SQL is the `WHERE` clause. The `HAVING` clause does filtering too, but not exclusively. The data aggregation which is allowed in the `HAVING` clause makes it more complicated to implement than the `WHERE` clause. Therefore we first turn our attention to the `WHERE` clause part of the query.

Next we must figure out a way to convert the `WHERE` clause to sequentially placed modules. This part of the query can be represented by a Boolean decision tree where the operands are fields in the input table(s). Consider a query on a table `"numbers"` consisting only of integer values, for simplicity.

```
"SELECT * FROM numbers WHERE intA = 0 AND intB > 0
AND (intC < 0 OR intD = 1) OR intE < 1 OR intF > 1"
```

In this example there are two types of logic operators. There are the relational operators `"<"`, `">"` and `"="`, and there are the logical operators `AND` and `OR`. This implies that there must be two different classes of modules. Firstly, there are the ones that implement the relational operators. These compare a register value to the specified field from the streaming data bus, and the output of previous module stages does not have an effect on the output from these modules. Secondly, there are the modules that implement the logical operators.

Figure 4.9 shows a Boolean decision tree corresponding to the previously stated query. Note that the relational operators and operands have been replaced, `(intA = 0)` by `a`, `(intB > 0)` by `b` and so on. This has been done because the outputs of the modules solving the relational operations are producing the inputs to the modules solving the logical operations. The tree height and structure is determined by the composition of the logical operators, not by the relational ones.

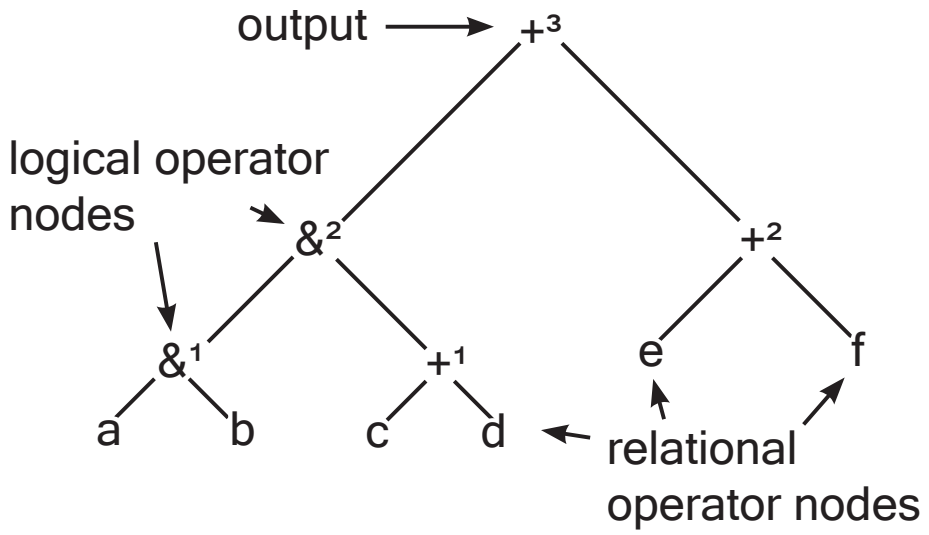

Figure 4.9: Decision tree. Leaf nodes are relational operator nodes and branch nodes are logical operators.

Once the decision tree has been derived from the `WHERE` clause, determining the module placement is simply a matter of traversing the tree from left to right, or

vice versa. The pseudo code in Algorithm 4.1 describes a recursive implementation of this. The curved line in Figure 4.10 shows a graphic representation of tree traversal, and the stars on it denotes placement of modules. This results in a module placement according to Figure 4.11.

Algorithm 4.1 Algorithm for deriving module placement from expression tree.

```
PLACE_MODULES_RECURSIVE( top level root node )
function PLACE_MODULES_RECURSIVE(node_tree)
    node ←node_tree
    if node is relational operator then
        add this node as the next module in the datapath;
    else                                        ▷ node must be logical operator
        PLACE_MODULES_RECURSIVE( this node's left branch)
        PLACE_MODULES_RECURSIVE( this node's right branch)
        add this node as the next module in the datapath
    end if
end function
```

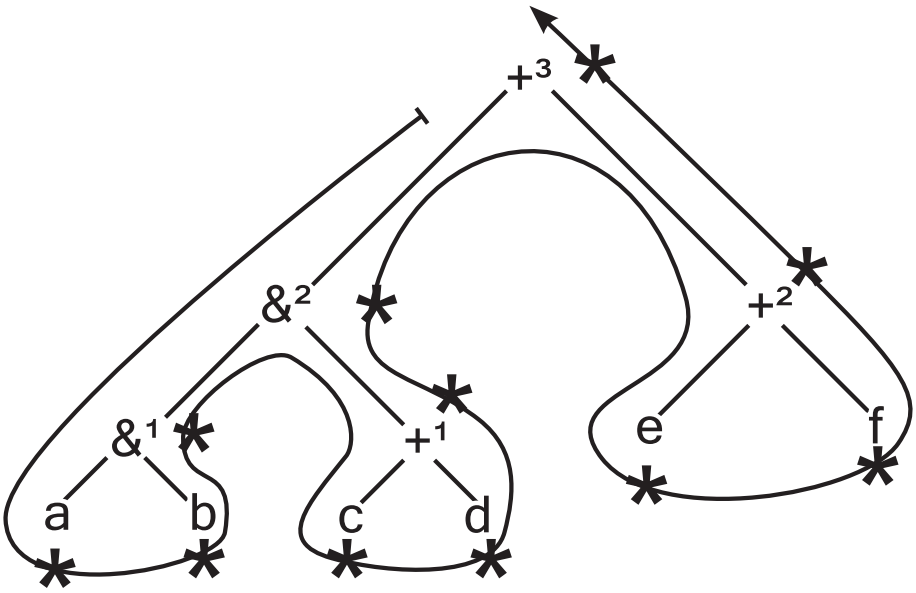

Figure 4.10: Parsing the decision tree. The stars mark the points where a new module is added to the datapath.

The final placement on the FPGA may vary due to resource constraints. However, Figure 4.11 shows the partial order in which modules have to be physically placed to evaluate the `WHERE` clause example.

The figure also shows how every relational operator module adds its result to the top of the results bus, and shifts down all the results from previous modules by one place. When a logical operator module is encountered, the two top results on the results bus are consumed, and replaced by the result from the logical operation.

The height of this tree is directly proportional to the number of intermediate results that has to be propagated along with the data signals. The observant reader might have already figured out that this is not always true, and it is not. In fact, a

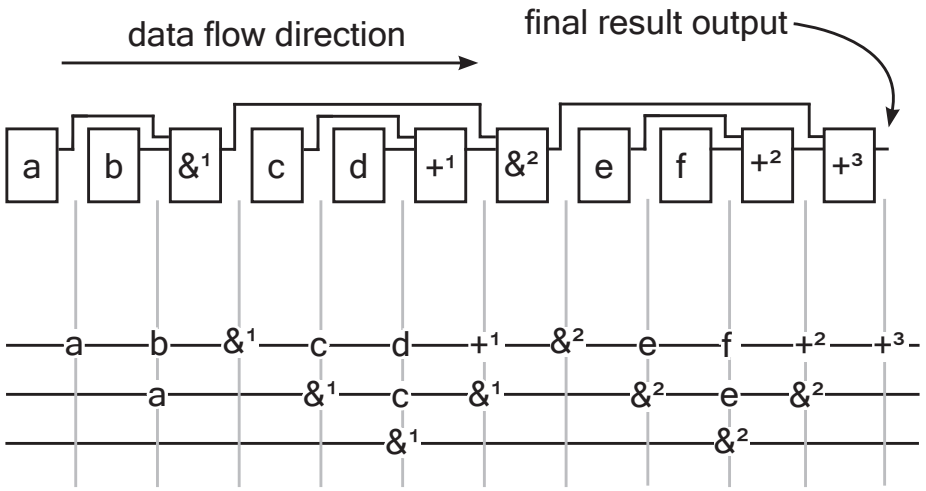

Figure 4.11: Module placement derived from Figure 4.9 and 4.10. Including intermediate signals' placement on the result bus between modules.

tree of any height can be implemented as a path of modules using only one single intermediate signal. This is true as long as the tree is, or can be rewritten as tree consisting of only `AND` operators. But for a balanced tree containing only `OR` operators, the tree height corresponds to the number of needed intermediate signals. The datapath must be functional for all types of trees, so we assume that the tree height is always proportional to the number of intermediate signals.

Reducing the tree height can in some cases be accomplished by applying Boolean algebra to the expression, but this is out of the scope of this project. And as we shall see in the next section, the implementation of the modules can be used to reduce the number of intermediate signals.

The discussion on the needed intermediate signals resembles the problem of evaluating Boolean expressions with a minimal number of temporary variables. A problem that has not been though of since the early days of computing when memory was scarce re-emerges in our project.

To allow modules and datapaths of arbitrary size in terms of latency, the input to and output from a module must be decoupled in the time domain. Consequently, the input and output from the datapath must also be decoupled in the time domain. All modules are synchronized by the same clock source, but they should by able to have an arbitrary amount of clock cycles latency.

The only performance metric that is of any concern is throughput. High volume database operations are throughput driven, and latency in terms of clock cycles is not an issue. Therefore we postulate that the modules and the entire datapath can consume as many clock cycles as needed, but it has to consume and process a new data item every clock cycle.

In this context a data item means a portion of the input table(s) corresponding to the size of the input data bus. This can be referred to as a *chunk*. Further we define a smallest granularity for the data items that can be handled by the accelerator. For a regular database this could be one bit (a Boolean signal), one byte, or more. Dealing with such small atomic units in the modules is undesirable because it will result in complex multiplexers. Instead we dictate that the smallest atomic data item will be 32 bits, and smaller units must be padded to this size. We will refer to

this as a *word*. An integer will be the size of one word, and strings must be padded to fit into a multiple of words.

The modules are required to be able to consume a chunk every clock cycle, but what if there is none available, or if the system is idling? For this purpose an `IDLE` state flag is introduced. If there is no input data or if the system is idling, the `IDLE` state is propagated with every invalid data chunk, and between the modules. When a module sees this flag, it idles all internal processing steps for one clock cycle and propagates the flag to the next module along the datapath. This is effectively forward flow control.

For allowing the modules to keep track of what portion of each record a chunk belongs to, some kind of synchronization is needed. This could be achieved by either propagating some kind of flag with the data chunks that marks the beginning of a record, or it could be a number value that labels each chunk. If the flag option is chosen only one bit signal is needed, but then counters within each module are needed to keep track of the input chunks.

A far better solution is to incorporate the labeling of chunks with the other state flags. There is the previously mentioned `IDLE` state flag, but as described in the next section, there is a need for several further state signals. A bus containing predefined states where some are chunk labels, and others are special state signals are propagated along with the data chunks. This makes the HDL code more manageable and easier to debug.

4.6 Module Concept

Modules implementing SQL operators are the building blocks of the datapath. For example, the operators used in the `WHERE` clause can be turned into modules to execute this part of the query. The library of modules determines what data streaming operations the accelerator can and cannot perform.

In the `WHERE` clause part of a query there are a number of different operators that can occur. There are the logical and relational operators, but there are also others like arithmetic and unary operators. Implementing all of these operators as modules would be too complicated to be carried out in this thesis, and it is not necessary to merely demonstrate the concept of database acceleration using a reconfigurable datapath. Moreover, implementing all variations of these operators would result in an unlikely large module library.

There are also many different data types that can be used in a database. Two of the most common data types are integer numbers and text strings. To reduce the complexity of the software parser and to ease the storage of tables in the accelerator DRAM memory, we limit the number of supported data types to integers and strings.

In a database there is also the possibility to have a NULL value at any record. Implementing this in hardware would require some kind of flag to be stored in the tables, indicating that a particular record does not contain any value. This would further complicate the data storage, parser, and the modules, so we simple say that the accelerator handles only strings and integers.

Note that according to the IEEE standards for floating point representations, a module for comparing floating point modules would be similar to an integer compare

module. It would only have to be changed to handle the special cases of NaN (not a number) and positive and negative zero.

When dealing with strings in databases, *regular expression* is a powerful tool. Regular expressions allow the user to match patterns using wildcards. A wildcard may match a single character or an arbitrary number of characters. A regular expression can be represented by a Deterministic Finite Automaton (DFA) or a Non-deterministic Finite Automaton (NFA)[11]. A DFA graph could be unrolled to modules in the non-blocking datapath like the rest of the `WHERE` clause. This would require a more sophisticated query parser and specialized modules, so regular expressions are omitted in this thesis.

To be able to demonstrate the most common data filtering queries we focus our attention on implementing three different modules. Firstly, the *integer compare module*, for comparing two 32-bit integers with one of the compare operators. Secondly, the *simple pattern match module*, for string pattern matching within one chunk. And thirdly, the *long string pattern match module* which can do pattern matching over multiple chunks.

One key concept of this architecture is that the input and output are unsynchronized in terms of clock cycles. But there has to be some way to tell the host computer that the query has completed, so we introduce the DONE state. This has the same meaning as the IDLE state to the modules, but it tells the output logic on the static system that the query has completed. The number of records returned by the query is then written to the host computer through the control interface. At last we include a RESET state as the fifth and last of the special states. This tells the modules, as the name suggests, to reset its registers to the original state.

4.6.1 Module Initialization

Once the modules have been placed to form the datapath, their internal registers must be set. The pattern matching modules must be set with a match string, and the integer compare module must be set with a compare value. The modules must also know where the data fields are in the stream. This means at which chunk, and where inside a chunk, the word is located. This must also be written to their internal registers after they have been placed on the FPGA.

To avoid confusion between the action of reconfiguring the FPGA and configuring the modules internal register once they have been placed on the FPGA, we shall from now on refer to the latter as *initializing* the modules.

The module initialization is required not only for one, but for several reconfigurable modules that are stitched together in a one dimensional chain as shown in Figure 4.11 on page 40. Consequently, we need a way to write selectively to registers in modules that can be placed at different positions.

To allow the modules to be initialized we introduce two new states, INIT and INIT_PHASE2. We also introduce a new mode of operation on the data bus, the *init mode*. The trick is to design the modules so that if they see the INIT state on the state bus, the first byte on the data bus is decremented by one before it is passed on to the next module. The state bus is as always left unchanged. When a module sees that this decrement byte evaluates to zero, it knows that the initialization data is meant for this particular module. This counter based addressing scheme uses the strict sequential placement order of the reconfigurable modules.

The pattern matching modules can be used to match a string spanning the whole

of a chunk, including the decrement byte. To simplify initialization, a second init state is used, the INIT_PHASE2 state. When a pattern match module sees this state, it knows that the next item on the data bus flagged with INIT_PHASE2 is the match string, which is then written to the appropriate register.

4.6.2 Data Bus

The maximum DRAM memory transfer speed on the MAX3 card is reported to be 38.4GB/sec. The data bus coming from the DRAM can be a multiple of two. The Maxeler compiler will take care of the conversion to the correct bus width. When deciding on a data bus width the maximum memory speed divided by the frequency the modules will be running at defines the upper bound. If we run the modules at 300MHz the calculation shows that $38.4GB * 8/300\text{MHz} = 1024\text{bits}$.

Unfortunately such wide buses make the reconfigurable area too large. This causes difficult routing situations because of the internal vertical routing in a module, and because the partial island is simply too large to route around. So we choose the half size, 512 bits. This is equal to $512/8 * 300\text{MHz} = 19.2\text{GB/sec}$ if we manage to get the modules running at 300MHz. Remember, the smallest atomic data item that is allowed in the data stream is a 32-bit value called a *word*. This means that one chunk will now contain 16 words.

As mentioned before, the modules have to be designed to support a pipeline fill rate of one new data chunk per clock cycle. This implies a computation throughput of 19.2 GB/sec. Reading the DDR memory introduces overhead for DRAM refresh and file system access. Furthermore, sometimes we have to read unnecessary data that will be discarded by the `SELECT` clause. Consequently, the 38.4 GB/sec - to - 19.2 GB/sec data rate ratio is a good match to highly utilize both the memory interface and the reconfigurable datapath.

4.6.3 State Signal Bus

In the previous section the decision was made to merge the synchronization of chunks and the state flags into one state signal bus. This raises the question of how wide this signal bus should be.

There are several things to consider when determining this answer. The number of state signals determines the number of addressable chunks, and thereby sets the maximum size a record (row) in a table can be. State signals are overhead as they consume logic resources that could otherwise be used for the datapath. They must be routed on the FPGA along with the data bus. Also, using a lot of state signals will result in more complex comparators in the modules, since they are used as inputs to modules' multiplexers and state machines.

Another consequence of having more addressable chunks, is that the long string pattern match modules would have to store more match strings. Either that or the situation would be that the maximum record size would be more than the maximum string size, which would be undesirable.

When deciding on such global constants, it is also a good idea to take a look at the FPGA architecture. The Virtex-6 chip is the target device, and according to its datasheet the Look Up Tables (LUTs) can be configured in two different ways. As a single 6-bit LUT, or as two 5-bit LUTs that have separate outputs but common logic inputs. Therefore, choosing a bus width of seven versus six may have some

impact on the logic costs. If we choose a five-bit state signal bus the total number of addressable chunks would be $5^5 - 5 = 27$. Five states are consumed by the IDLE, INIT, INIT_PHASE2, DONE and RESET states. This gives us the maximum record size of $27 * 64\text{B} = 1728$ bytes, which is enough for the tests we have in mind.

With this design decision, a single 6-input LUT can evaluate two different states at the same time, and a single record can be more than one and a half kB long.

4.6.4 Results Signal Bus

In Section 4.5, we learned that the number of intermediate result wires needed is proportional to the tree height of the decision trees we are expecting. We also distinguished between relational operator nodes and logical operator nodes. When designing the modules it appears that the modules dedicated to a logical operator would be much simpler than the ones dedicated to relational operators. In fact any logical gate with two inputs will only take up one LUT. The rest of the logic cost comes from routing and pipelining the input signals to the output of the module.

Therefore, it will be a better solution to incorporate the logical modules into the relational ones. This reduces the amount of modules in the library, and also reduces the need for intermediate wires between them. The operator types have to be given to the module during the initialization. A decision is made to use eight intermediate wires, and allow for eight successive logical modules to be absorbed by one preceding relational node. That should be enough to cover any query we are expecting to test without being too wasteful.

The result from a module is only valid on the output of the last data chunk. It has to be this way because we are only interested in the result at the moment the full record has been processed. Modules interact on the results of the previous result, so using the last chunk of every record for synchronizing this seems like a good choice.

4.7 Module Architecture

We have now defined the input and output buses for the modules. The modules' black box view is shown in Figure 4.12. A simplified overview of the architecture common to all `WHERE` clause module types is shown in Figure 4.13.

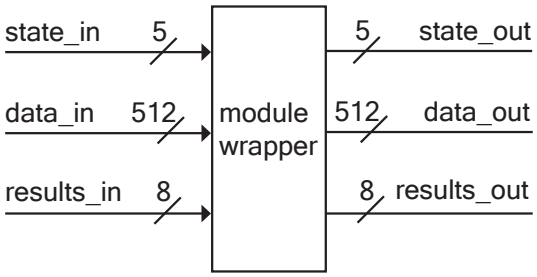

Figure 4.12: Module black box view.

On the input side of the modules, all the signals first go into register flip-flops. This is done to ensure that the timing will be met for the partial region combined with

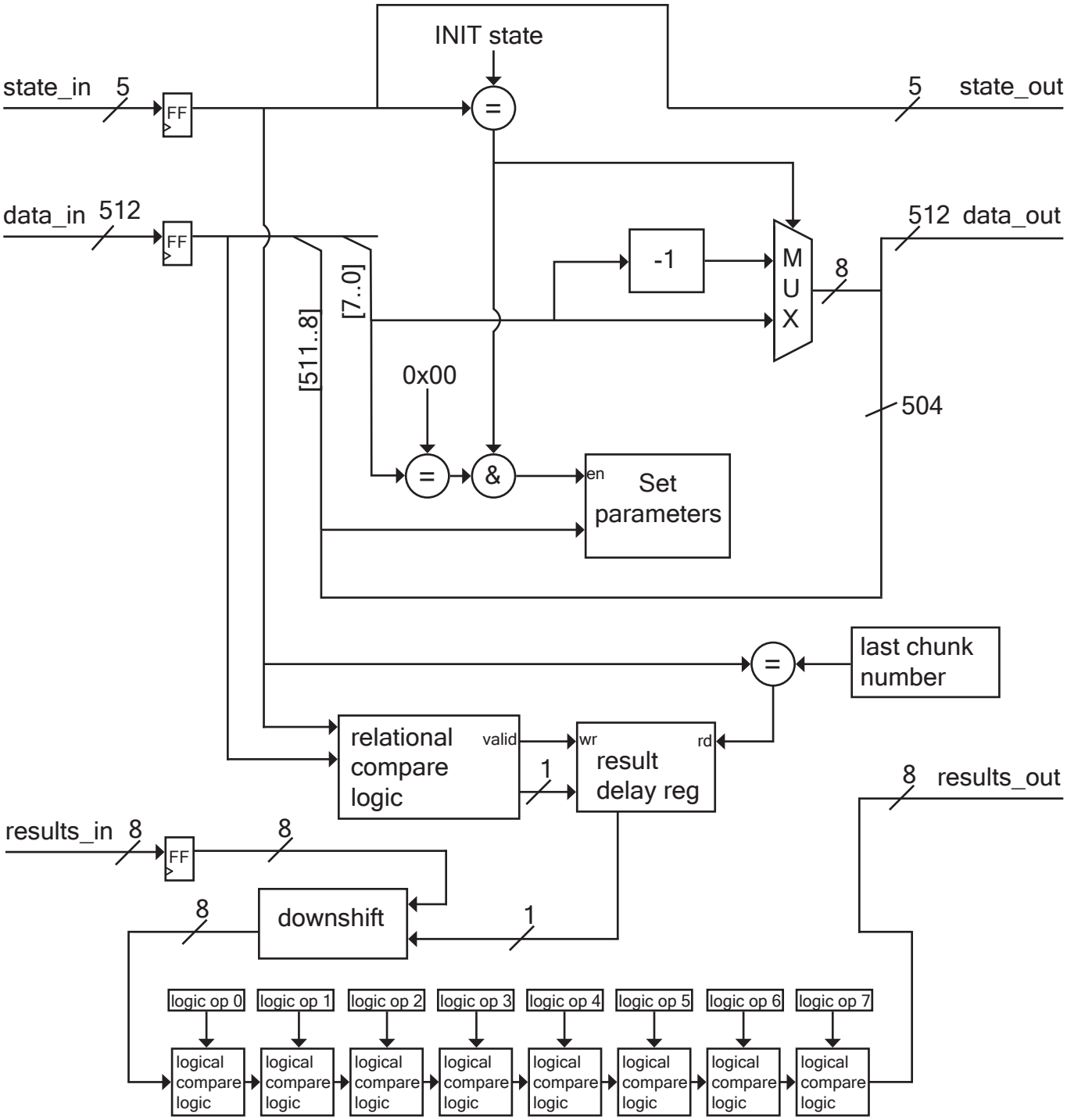

Figure 4.13: Module architectural overview common to all `WHERE` clause modules.

the static system. The router tool will only provide a route with a valid timing score to and from what we define as inputs and outputs for the partial region. Since the modules will be reconfigured dynamically, one must design them in such a way that no path will be longer than the maximum allowable delay once they are stitched together. Therefore, we put flip-flops on the inputs, and also have the signals from the last module going directly into flip-flops in the static system.

On the top half of Figure 4.13 we see the basic functionality of the initialization logic. If the state signal is the INIT state, the first byte is decremented by one. If the decrement byte at the same time evaluates to zero, internal registers are set with values from the data bus. The initialization logic is more complex than this because of the two stage initialization introduced earlier, but the figure shows the basic concept.

The data bus input also goes to the relational compare logic, which of course differs from one module type to another. The result is then delayed until the last chunk is put on the output bus. An integer compare module might have its target field in the first chunk of a record, but as mentioned previously the result from any module is only valid when the last chunk is output from it. Therefore, the result has to be delayed until the last chunk is ready to be output, if the compare field is not at the last chunk already.

Following the result delay logic is a downshift of all the results from previous modules, while the new result from this modules relational compare operation is added to the top of this bus. That is, if there was one result on the result bus from a previous module, it would be at index <0> on the result bus as it enters the current module. This result would then be shifted down to index <1> while the new result replaces its value at index <0>. This is exactly what was illustrated under the nodes `a`, `b`, `c`, `d`, `e` and `f` in Figure 4.11 on page 40.

After the downshift comes the evaluation of the eight logical operators. There are eight of these because the decision was made that every relational operator node should be able to absorb up to eight succeeding logical nodes, and thereby eliminate the need for dedicated logical operator modules. Note that the figures are only conceptual and assumes that the module consumes only one clock cycle, this is not the case in the implementation.

The logic path from the input data bus to the output result as shown in Figure 4.13 is very long, and would result in a long logic path, thus limiting the maximum clock frequency of the module. In the implementation of the modules, pipeline stages are added to divide the logic paths into smaller, faster paths. This is allowed because the requirement of a module is that it accepts a new data item every clock cycle, and outputs one. The database accelerator is throughput driven, latency is not a concern.

4.7.1 Logical Compare Unit

This part of the modules implements the logical node absorption. Every module contains eight consecutive logical compare units. That allows them to absorb up to eight following logical compare nodes from a decision tree, like the one shown in Figure 4.9 on page 38. For simplicity it only implements the most important operators, including `AND`, a `NAND`, an `OR` and a `NOR` gate. The logical operator is specified by a register that is written during the module initialization phase (see Section 4.6.1).

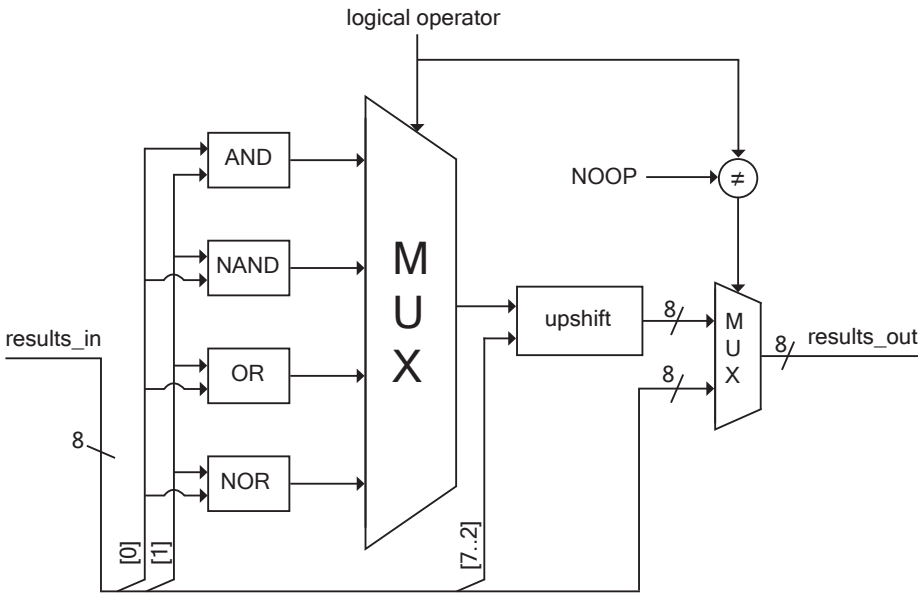

Figure 4.14: Logical compare unit. The MUX on the left side selects the operator based on the value of the `logical operator` *register. The right MUX bypasses the unit if there is no operator (No-Op).*

The path from the input to the output of the `WHERE` clause modules is depicted in Figure 4.13 on page 45. The figure shows that the logic path from input to output goes through eight logical compare units. That is a very long logic path to be evaluated in one clock cycle. It is clear that all `WHERE` clause modules must be pipelined in the final implementation to obtain an adequate maximum frequency rating.

4.7.2 Integer Compare Module

For the *integer compare modules*, the compare logic that fits into the "relational compare logic" box in Figure 4.13 (page 45) is shown in Figure 4.15. There are four register values that must be set during the initialization phase. There is the `compare value` register which holds the integer value to be compared with the values from the target column specified in the query. The `word position` register tells the module at what word the column of interest is found, and the `compare operator` register selects the correct relational operator with the help of a multiplexer. This logic path is active at all times, but the output is only valid when the specific chunk passes through the module. This must be controlled by initializing the `match chunk number` register with the correct chunk number.

All in all, the integer compare logic is fairly simple. It consists of dedicated circuits for all supported relational operators. Which one to use is selected by a multiplexer that reads the `compare operator` initialization register. Implementing the software that generates the initialization data for this module was in fact much trickier than implementing the module itself in VHDL.

Initializing the integer compare module is done in one `INIT_PHASE1`, and takes one clock cycle.

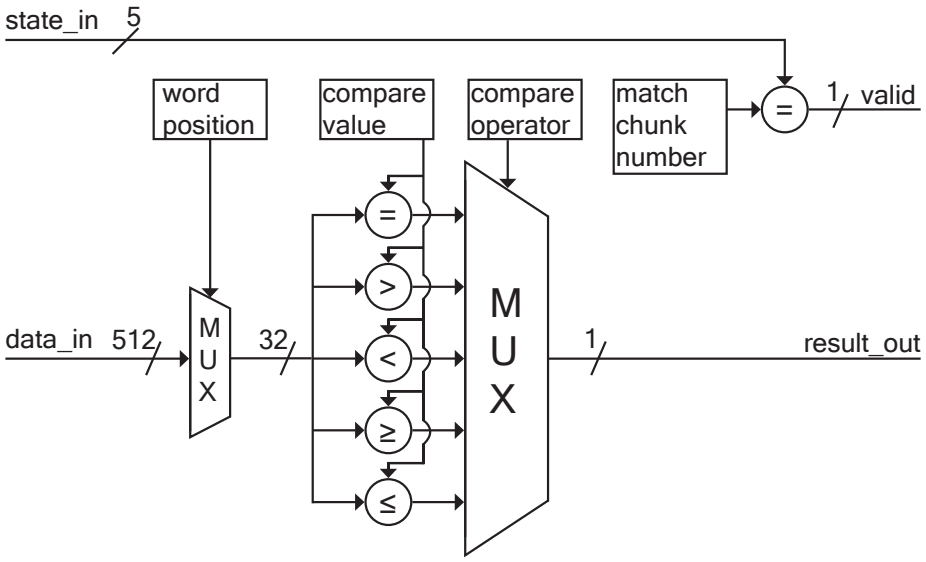

Figure 4.15: Integer compare logic.

4.7.3 Simple Pattern Match Module

This module is designed to match strings that span over no more than one chunk. The query planner software will evaluate the alignment of a text field in the data stream, and decide if it fits into one or more chunks. If it fits completely in one chunk, the *Simple Pattern Match Module* is used, otherwise the *Long String Pattern Match Module* will be used. The Simple Pattern Match Module can therefore compare strings that are $16 * 4 = 64$ characters long.

The reason for having two pattern matching modules is that the one that can span over multiple chunks is more complex, and consumes a larger area on the chip than the simple one. It might have been a good idea to have a pattern match module that can span over only two chunks, because a text string could be short but still span over two chunks. The case could be that it is 8 bytes long (two words), but that the first word is at the end of one chunk and the last is at the beginning of the next chunk.

The pattern matching function is implemented with 16 32-bit comparators as shown in Figure 4.16. They compare words from the stream with register values that have to be written to it during the initialization phase. The active words can be specified by setting the `active words bit mask` register in the initialization phase. This bit mask selectively activates the comparators so that they match the fields' alignment inside the chunk. The result is valid only when the chunk entering the module is equal to the value of the `match chunk number` register. Again, this must be set at initialization time.

As was the case with the Integer Compare Module, all pattern matching circuits are present at all times. The output is governed by the previously mentioned bit mask register. The bit mask approach moves much of the complexity of selecting input words from hardware to software. The philosophy is that implementing a more autonomous hardware module is more costly in terms of resource usage and development time than implementing a complicated software application.

Initializing the *Simple Pattern Match Module* always takes two clock cycles. One chunk tagged with the `INIT_PHASE1` state must first target the module. It sets the

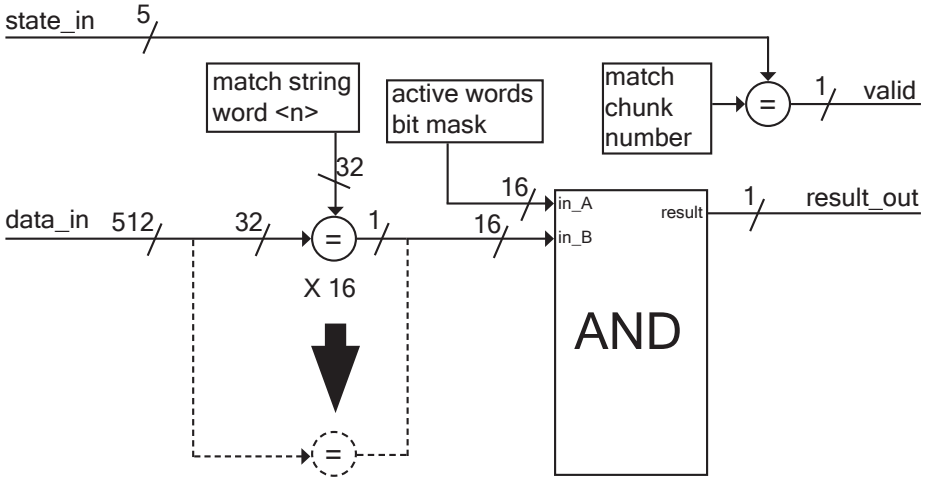

Figure 4.16: Simple Pattern Match Module.

internal registers and instructs the module to wait for the `INIT_PHASE2` state. When it arrives, the match string register is set and the module is operational.

4.7.4 Long String Pattern Match Module

The pattern matching module for long text strings is basically the same as the one for smaller strings. The difference is that it can store a match string that spans over several chunks. Therefore it has to propagate the intermediate result from the first match chunk to the last match chunk. This has to be configured in the initialization phase, as well.

Both pattern matching modules store the match strings in registers (LUTs). This has been done to ease the implementation, but it would be more efficient to store them by using the Block RAM (BRAM) that is available on the chip. With this implementation the BRAM in the partial region is unused and wasted. Also, BRAM can store more data than LUTs in a smaller area. But by looking at the implemented design it seems that the bottleneck for making compact modules is not the LUT usage, but the vertical routing resources. In our case with the 512-bit wide data bus, it is the vertical routing that limits how narrow a module can be on the chip.

Since the decision was made to limit the record (row) size to 27 chunks, there is no gain in using BRAM versus LUTs. The vertical routing problem is a good example of a new limiting factor that is specific to slot based partial reconfiguration using very wide buses.

Initializing the *Long String Pattern Match Module* must be done over several clock cycles. Every chunk that should be matched and its corresponding bit mask is first specified by a state flagged `INIT_PHASE1` followed by an `INIT_PHASE2` state that contains the actual match string for the particular chunk. Initializing this module to its full capacity of 27 chunks would take 54 clock cycles.

4.8 Data Storage and Movement

Input and output from the FPGA is handled by the Maxeler system. The Maxeler system can provide streams with either a push or pull interface. The multiplexing and flow control is also handled by the Maxeler system. We use these auto generated stream interfaces to transfer data to and from our custom VHDL implementation.

The FPGA is mounted on a PCI-express board connected to the host computers motherboard. Interfacing with the host computer is achieved by using the FPGAs PCI-express interface at a maximum data rate of 2GB/s. Maxeler provides a DMA controller for the PCI-express interface that supports up to 15 independent input or output streams. However, all streams to and from the host computer share this bandwidth. The multiplexing is done by the Maxeler system and is transparent to our implementation.

Figure 4.17 shows an overview of the streams included in the design. There are two 512-bit input streams and one 512-bit output stream. One stream goes to the memory, allowing us to store tables in the memory or alternatively stream them to the modules when needed. Another 512-bit stream goes directly to the modules, and is useful for debugging.

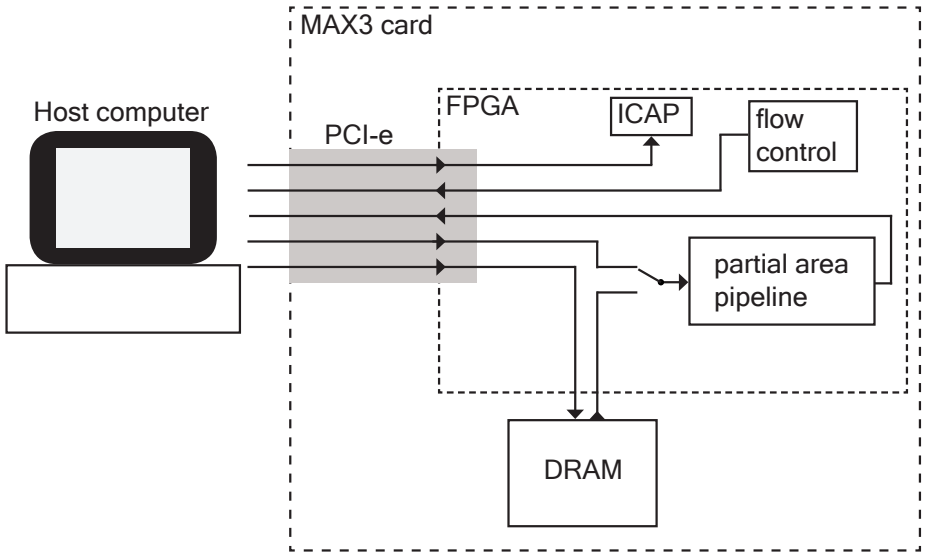

Figure 4.17: An overview of the data streams in the system.

The PCI-express input from the host computer only has a bandwidth of roughly one tenth of the modules, so the tables must be streamed from the memory to achieve the full throughput. The output from the partial region goes back to the host computer. In a more advanced database accelerator there would also be a stream back to the memory. This would be necessary for storing temporary results, and for executing SQL *join* operations.

Along with the output data stream, there is a 32-bit wide flow control stream. This stream is needed to tell the host computer that the query has completed, and to give it the number of returned records. This stream is only active for a few cycles after each query has completed, so it does not consume bandwidth from the output stream.

The last 32-bit stream goes to the FPGA's internal configuration access port (ICAP). This stream is used for reconfiguring the partial region. Config data is streamed directly from the host to the ICAP port, triggering the partial reconfiguration of the device.

4.8.1 Record Management

There are several methods used by databases to store tables in the memory[10]. There is the straightforward implementation where each record spans over one fixed size memory block. The type, length and offset are stored in the table header. This storage scheme wastes space, because each record of the same column must have the same size. For example, most text strings in a database are only a few bytes long, but if one slot contains a 100 bytes long string, all text strings will now consume 100 bytes.

An alternative to fixed sized memory blocks is the variable field length storage scheme. In this scheme, each slot has a length descriptor field. This produces some overhead data, but will in most cases reduce the required storage space. If this storage scheme is used for database streaming, the static system must contain logic to align the slots so that they appear at the correct word positions when streamed to the modules.

The decisions on how to store tables in the memory and the complexity of the streaming part of the static system are closely related. A simplified data storage scheme wastes space, but makes the streaming of the tables straightforward. A more complicated data storage scheme calls for a complex streaming logic to extract the data fields and insert padding where necessary.

A more advanced implementation could store the tables compressed in the memory. This would require compression and decompression logic, but there is also a potential speed gain from this. A decompression scheme could supply the accelerator with data at a faster rate than the DRAM speed[4].

A file system like the one described in the next section would be desirable for abstracting the storage of tables away from the user. However, due to implementation difficulties described in Section 5.2.2 the file system had to be abandoned.

Since the file system design process turned out to be unreasonably time consuming, dumbing down the record management suddenly seems like a good idea. The tables will be stored in the memory padded to the nearest chunk size. In this way no special alignment and flow control logic is needed. The tables can be streamed directly from the memory to the modules, and the flow control will be handled by the Maxeler system.

Figure 4.18 shows an example of how a record with three different sized fields would be stored in memory and streamed over the data bus. The records with padding are stored in the memory, one record following directly after the other. When they are streamed through the modules they already have the correct alignedness. The task of aligning and padding the records is performed by the query parser, and the tables are loaded to the memory using a dedicated PCI-express stream.

4.8.2 File System

A file system is needed to allow storage of large data items in the DRAM memory. To allow tables to be stored and deleted from the memory during run-time, some

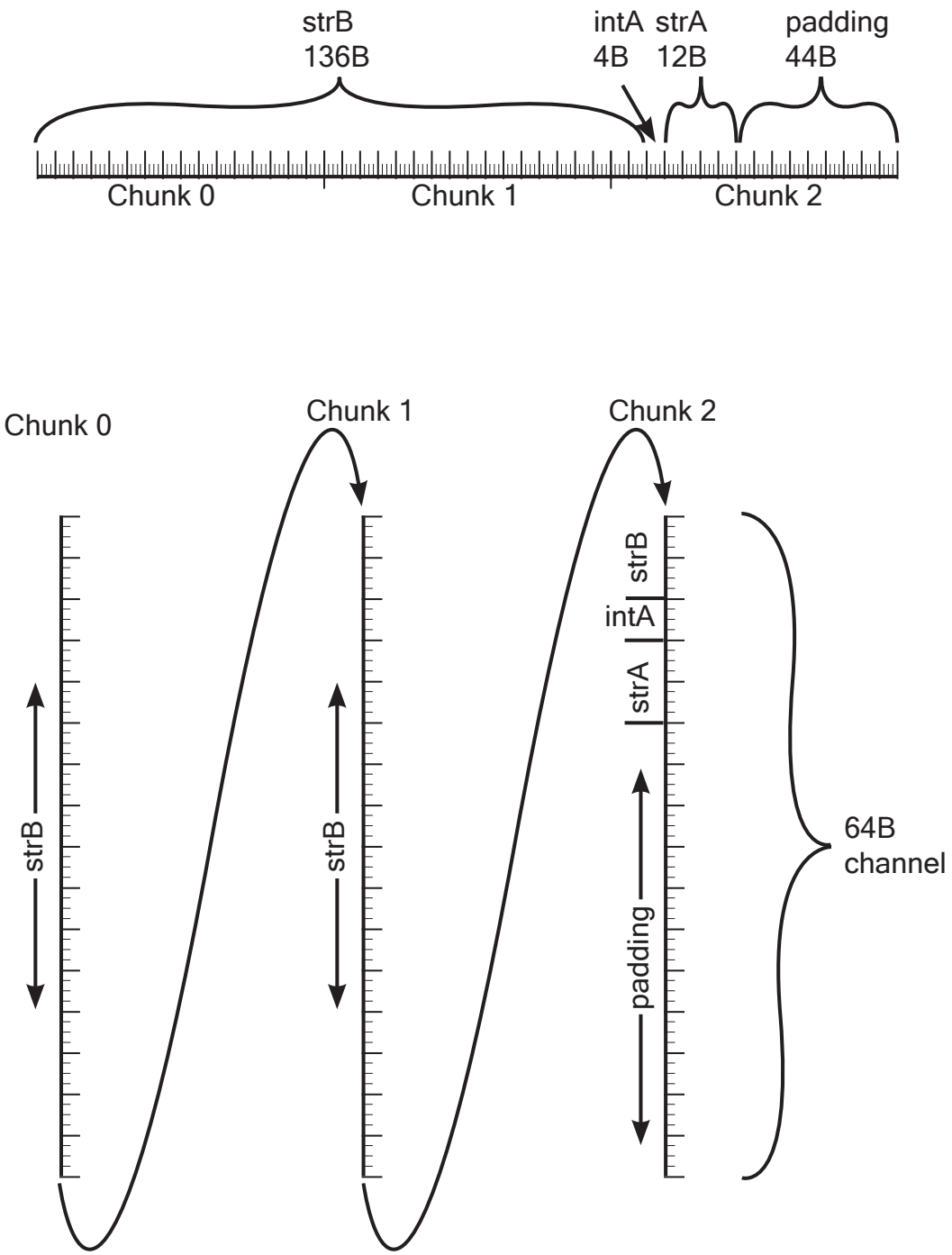

Figure 4.18: Illustration of how a record is streamed over the input bus and stored in the memory (top).

kind of data management scheme that allows fragmentation is needed, and for this we propose a *file system*.

The Linux *ext2* file system is a non-journaling file system with a relatively simple concept. Ext2 splits up the disk into blocks, a file can span over multiple blocks. The file size and pointers to the blocks that make up the file are stored in a record called an *inode*. The inode can also contain a pointer to another pointer structure in case the file is bigger than the maximum number of blocks the inode itself can point to. This second level pointer structure can in turn have a pointer to another pointer structure, and so on. This allows the file to grow and still be fragmented into block sized partitions.

The ext2 file system is more complicated than this, but the outline is very simple. Divide the disk into fixed sized blocks and manage these with pointer structures. Because of its simple pointer structure and the capability to access large files, concepts from the ext2 file system have been selected for an FPGA implementation of the file system.

Instead of having dedicated inode structures, we split the memory into fixed sized blocks that can either be used to store data, or to store pointer structures. The pointer structures contain pointers to data blocks, ordered sequentially. The last pointer in a pointer block is used to point to the next pointer block. This last pointer can be either null (zero in this case), indicating that there is no next pointer structure, or it can be a pointer to the next pointer block that contain pointers to the file's data blocks.

This pointer and data block scheme is illustrated in Figure 4.19. A file's identifier is the pointer address to the first pointer block. This integer number must be mapped to a human readable file name by the software on the host computer. When a file is to be stored in the memory the hardware file system is first given the number of bytes to be stored. Then the file is simply streamed to the hardware file system. At last, a pointer to its first pointer block is returned to the source.

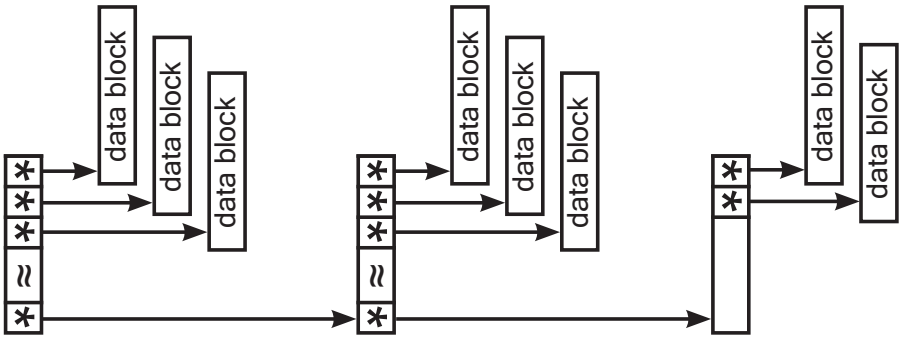

Figure 4.19: Pointer and data block scheme.

When the user wishes to stream a table from the memory to the accelerator, the query planner software must retrieve the pointer associated with the file name (table name). The stream to the accelerator is then set up, and this pointer along with the file size is given to the hardware file system. The transfer of the first pointer initiates the data stream from the memory file system to the accelerator.

The Pointer FIFO The file system must have a way to keep track of which blocks that are used and which that are free. When creating a new file, the file system

must get access to free data blocks. To allow this, a FIFO containing pointers to all the uninitialized blocks is implemented. The memory is split up into blocks, except for the initial part which contains a FIFO of uninitialized pointers (Figure 4.20). Whenever a new block is needed in the process of storing a new file, a pointer to an unused block is pulled from this FIFO.

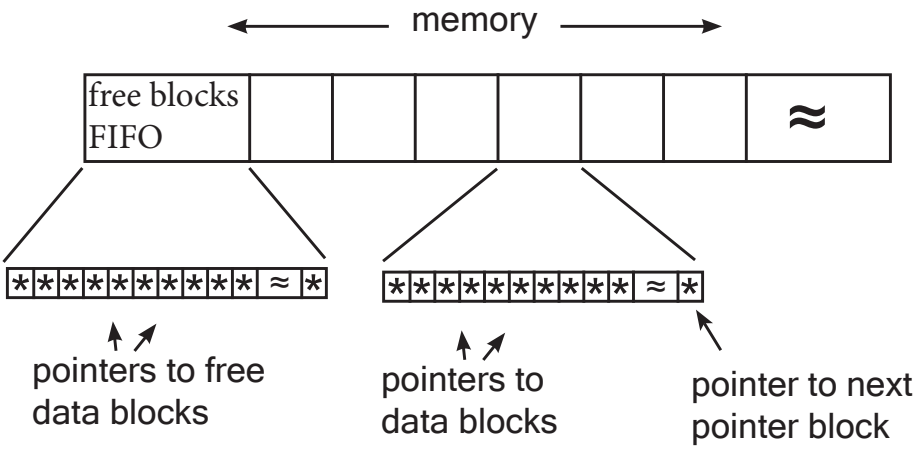

Figure 4.20: Partitioning of the file system. The initial space consists of the FIFO for unused pointers, this FIFO stores pointers to all free blocks. The rest is divided into equally sized blocks, and is used for storing pointer and data blocks.

This scheme of keeping track of free space by using an unused pointer FIFO differs from the ext2 file system which uses a data allocation bitmap for this. One reason for choosing this pointer FIFO is that is simplifies the implementation when targeting hardware. Another reason is that the maximum number of files to store is significantly lower that what is needed on a computer file system. This allows the block sizes to be larger, and this reduces the number of unused pointers that has to be stored. This means that the space used for this meta data structure has less impact than it would have had if the ext2 file system used such a pointer FIFO.

The third and maybe most important reason to use this FIFO is that the pointers can easily be sorted. When a file is deleted the blocks used for it has to be freed. These freed blocks are then pushed onto the FIFO. As the allocation and deallocation of blocks progresses, the memory becomes more fragmented. A simple countermeasure for this would be to sort the unused pointers from time to time. Sorting numbers in FPGAs has been explored by others, and it could even be accomplished by using partial reconfiguration[5]. By using the already present datapath and reconfigurable regions, the extra logic consumed by the sorting hardware would not be significant.

Translating a pointer into the absolute byte address is achieved by multiplying the pointer value by the burst size and adding a fixed offset to it. The offset is for skipping the first part of the memory where the unused pointer FIFO is stored, and the multiplication picks out a unique block. To make the implementation more straightforward, the pointer data type must be a power of two bits wide. Furthermore, they must be large enough to address any block in the memory uniquely.

Considering a file system supporting the 24GB of the MAX3 card, the minimum block size would be $24 * 2^{30}\text{B} \ / 2^{16} = 393.219\text{kB}$. Such large block sizes would fit a file system with only a few large files. Therefore, to be more general, 32-bit pointers are used in this project.

A major design decision regarding the file system is choosing block sizes. It is very difficult to develop complex generic VHDL implementations, that is, a file system implementation that can be built for a variable block size. Therefore, the block sizes will have to be hard coded in the VHDL implementation.

When designing a file system it is necessary to consider the properties of the underlying hardware. The MAX3 card we are using has a minimum memory burst size of 384 bytes. The smallest data size that can be read is 384 bytes, and the memory is only addressable in 384-byte units. The block size should therefore be a multiple of 384 bytes.

The ideal block size depends on the maximum number of files that will be stored. Getting this wrong will result in space being wasted in form of meta data structures. Having a larger number of files means that their average size will be less, and this calls for a smaller block size to minimize internal fragmentation.

With the implementation described above, the space in bytes required for the unused pointer FIFO will be the total number of blocks times four. This is because 32 bits equals four bytes, and because at most, all the data blocks are unused. The amount of used data block pointers consumes the same size, because at most all the data blocks are used. With random file sizes the average wasted space is approximately the size of one block. On average, almost half a pointer block will be unused per file due to *internal fragmentation*. The same applies to data blocks, every file will on average leave close to half a data block unused.

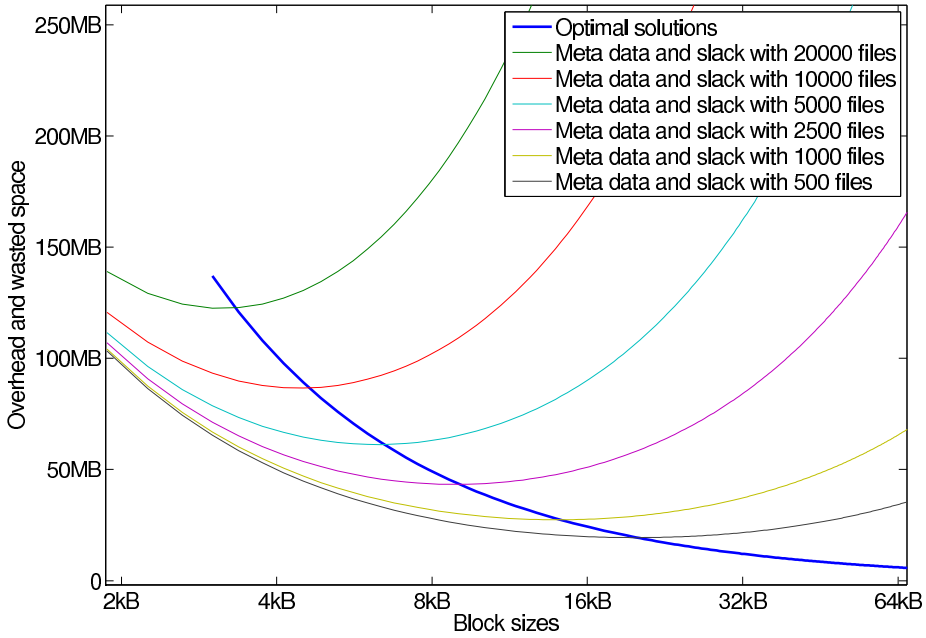

Figure 4.21: Overhead and wasted space as a function of block sizes. The Matlab code for this graph can be viewed on page 90.

These calculations have been plotted into Figure 4.21 to show the overhead and wasted space as a function of block sizes. The graph reveals that even with a file system supporting 20,000 files the overhead and wasted space accounts for only 0.5% of the total memory size. This is sufficient for the type of database the accelerator is meant for. For the final implementation we choose the block size of 3072 bytes (3kB). This is eight times the burst size of the memory controller. Eight is a power of two and this will ease the implementation of the file system.

Chapter 5

Implementation

In this chapter we will examine the system the way it was actually implemented. We will go through the implementation of the entire system and describe the most important features. The focus will be on the parts which give us the best understanding of the overall structure and functionality. We will briefly describe the steps taken to build the complete system, and notable difficulties and pitfalls will be emphasized.

We will start by describing how the software part was implemented. Then we will look at the hardware implementation of the static system. Finally, an implementation of each of the three partial modules will be presented.

5.1 Software

The main software application developed in this project is the *Query Planner*. It is responsible for parsing the queries and turning them into hardware datapaths. This is the front end that a user of the database accelerator sees and interacts with.

In Section 4.2 we discussed the use of ODBC to connect the accelerator to a regular software database. This connection implements only the baseline functionality required to prove the operation of the query accelerator. For a full scale implementation of an ODBC proxy a great deal of work would have to be put into making it compliant with the SQL language and the ODBC call level interface. Such an ODBC-proxy application would have been on top of the query planner in the actual software stack shown in Figure 5.1.

To interface with the MAX3 card and the FPGA an interface application is needed. The FPGA is monitored and controlled by the MaxelerOS, but is accessible to programs through the MaxCompilerRT API. For controlling the streams to and from the FPGA and to and from the memory, an *interface application* was developed. This program provides an abstraction level between the query planner and the Maxeler system.

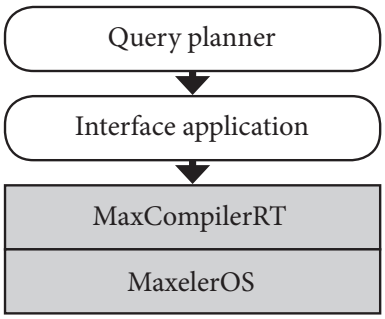

Figure 5.1: Software stack overview.

5.1.1 Query Planner

For processing the queries, a software application was written from scratch, entirely in the C programming language. The decision was made to write the software in C because this seemed more suitable for the existing software. Both the ODBC library and the MaxelerRT library are implemented in C. Writing such an application would preferably be done in a high level object oriented language, but a realistic implementation would require the speed of C or C++ to be functional. The query planner consists of more than 2400 lines of code.

The software is named *Query Planner* although it does not really generate any query plans equivalent to the query plans used in software databases (Section 4.3.1). However, because a query plan is closely related to the final datapath, the name Query Planner was kept.

The query planner features a simple command shell for entering queries and loading tables into the accelerators DRAM memory. It also has functions to create illustrations of the decision tree produced by the input query, and illustrations of the module placement derived from this decision tree. The query planner's parsing function is a bit more advanced than what is needed for this project. This was done to provide a generic solution that can be easily extended in order to provide more SQL operators.

Before any queries can be executed by the accelerator one or more tables have to be uploaded to the MAX3 card memory. The table reading from the database is performed by the query planner, and the transfer to memory is done by the *interface application*. The query planner keeps track of the tables stored in the MAX3 card memory. Therefore, we only upload tables that are not currently mirrored on the FPGA board.

A table could be stored on disk in several different ways. Database implementations typically have their own native storage scheme, and MySQL is no exception. MySQL can also export the tables to an SQL script file so that it can be imported by another database management system.

As an alternative, an export of the tables to comma separated values (CSV) was used. With this approach, information about table relationships and field properties gets lost in the conversion. However, this functionality is not required for a baseline implementation of the accelerator. Moreover, adding a more advanced interface to the MySQL database would have only very little impact on the FPGA hardware.

Currently, we export the tables to CSV files and add a custom header to the files

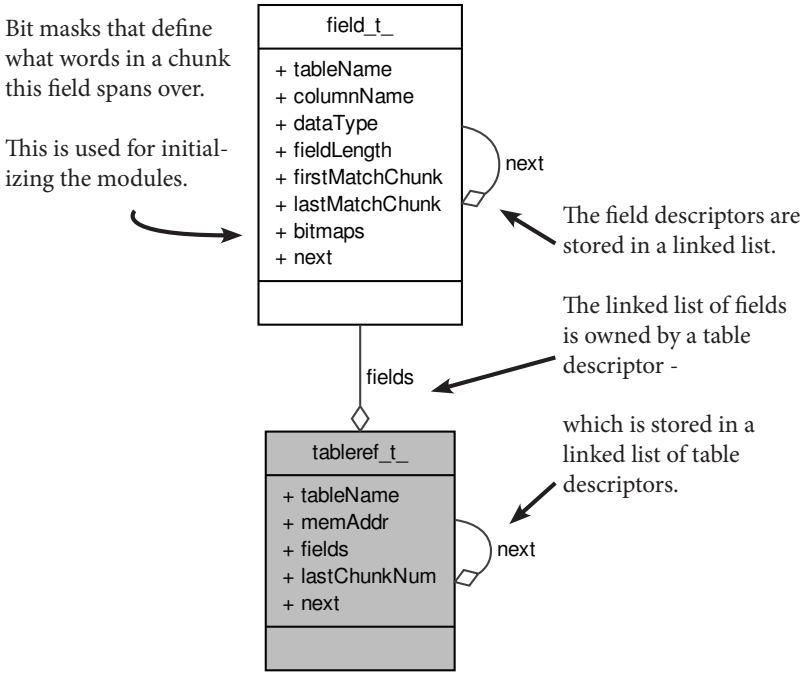

Figure 5.2: Collaboration diagram for the `tableref_t_` struct. It shows the data structure the query planner uses to store information about the tables and its fields.

manually. This custom header states the table name and the data types and lengths for all the fields. This CSV file can then be read by the query planner which aligns the fields and adds padding where necessary, as described in Section 4.8.1.

The transfer to the memory is handled by the interface application described in the next section. Information about the tables and the fields is stored in a data structure shown in Figure 5.2. The `tableref_t_` struct links the table name to a memory address. This information is used by the query planner to figure out the module placement and construct the initialization data stream.

A call graph of the query planner software is shown in Figure 5.3. The command line function has been left out so that the function call order of a typical run can be viewed from top to bottom. At startup, a table reference object is created to accommodate the information from the first table. Then the table is loaded in by the `readCSVtable()` function. The calls to the interface software have been omitted since they do not belong to the query planner.

After the table has been loaded into memory, the accelerator is ready to accept queries. A query is then parsed by the `parseQuery()` function, resulting in a `query_t_` struct. The graph in Figure 5.4 on page 60 shows the hierarchy of the structs that now contain the information in the query. As a feedback to the user a graph is drawn of the decision tree derived from the `WHERE` clause in the query. See Section 4.5 for more details on the `WHERE` clauses.

An example tree produced by the query planner is shown in Figure 5.5 on page 61. This decision tree is realized by the `condition_t_` structs from Figure 5.4 on page Figure 60. These nodes are either relational operators or they are logical operators and have left and right branches.

The next step is for the query planner to derive the module placement from this

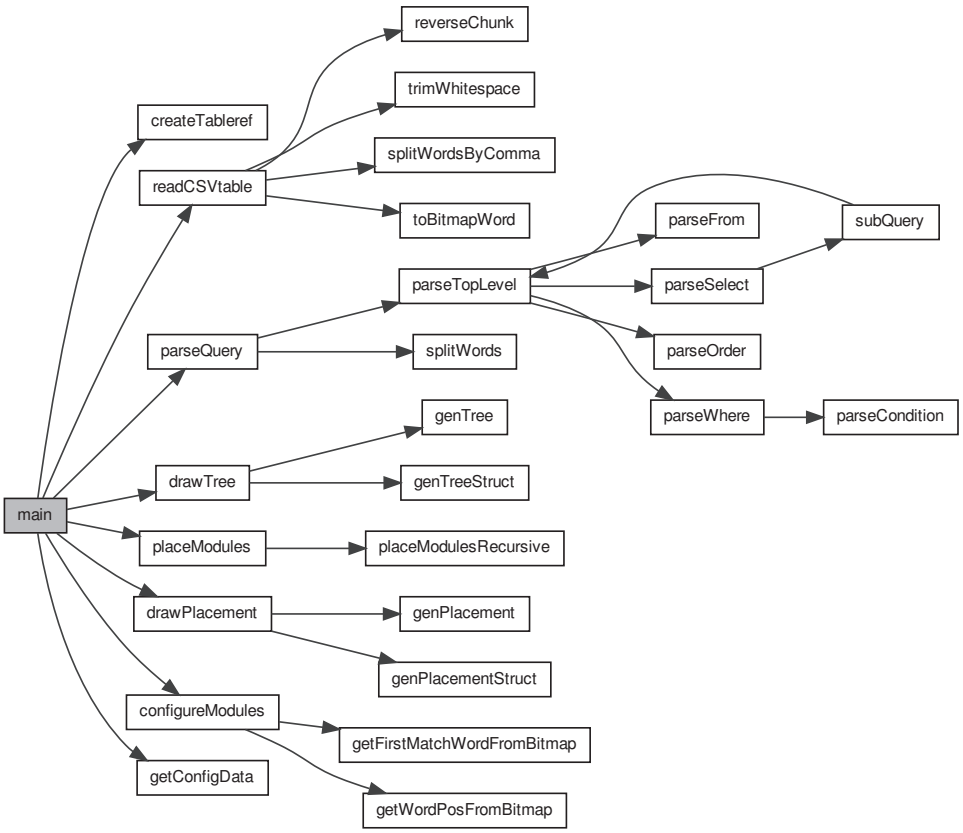

Figure 5.3: Call graph for query planner. Command line and MAX3 interface functions omitted.

decision tree. The `placeModules()` function traverses the decision tree the way it was shown in Figure 4.10 on page 39 to find the module placements. This operation yields a linked list of module objects. The order of these `module_t_` structs (Figure 5.6 p. 61) corresponds to the order in which they must be configured onto the FPGA to form the datapath.

A visual feedback like the one in Figure 5.7 on page 62 is presented to the user. Next, the initialization data that is necessary to set the modules internal register needs to be generated. This is done by the `configureModules()` function. It combines the information from the table reference and the condition nodes to figure out where in the stream the fields of interest will be for every module. This, together with the match values and the ordering of the modules, enables the query planner to generate initialization data for a particular query.

At last, the initialization data can be retrieved by the `getConfigData()` function, ready to be streamed to the modules. Once these steps have been completed and the initialization data has been written to the modules, the table resides in memory and the datapath is set up. The accelerator is now ready to execute the query.

5.1.2 Interface Application

The interface application communicates directly with the MaxelerOS through the MaxCompilerRT run-time library. A typical call graph is shown in Figure 5.8 on

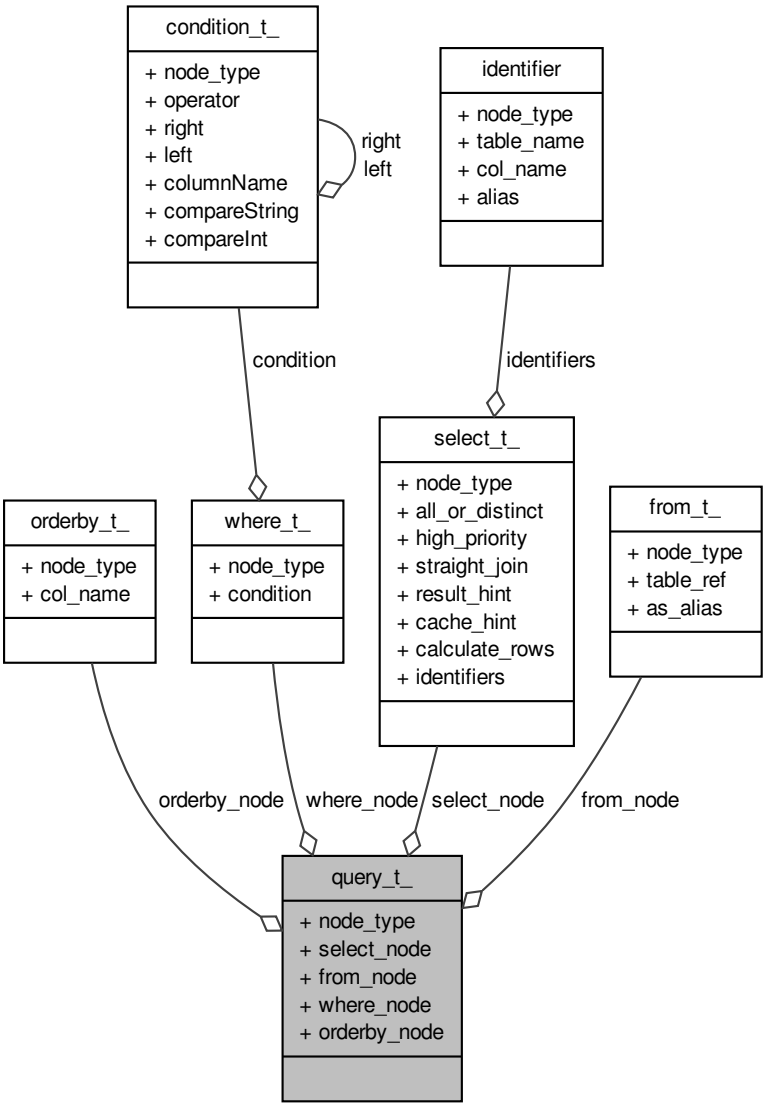

Figure 5.4: Collaboration diagram for the top level struct `query_t_`*. The interesting part in this project is the* `where_t_` *object. It owns one or more* `condition_t_` *nodes. These nodes form the decision tree by pointing to other* `condition_t_` *nodes with its* `left` *and* `right` *branches.*

page 62. The interface application must maintain data structures between every service call from the query planner. This is done by keeping static objects in the application so that they do not change between calls.

The result stream from a query in progress is returned to the host computer in real time. The problem is that the number of records to be returned is not known in advance because the number is data dependent. The MaxCompilerRT offers different ways to handle streams in software, including an asynchronous method using ring buffers. Unfortunately there is no interrupt that can be generated by the hardware for any of the methods, so there is no way for the hardware to tell the accelerator that the query has in fact finished.

To solve this tricky situation we use a control stream in parallel with the output

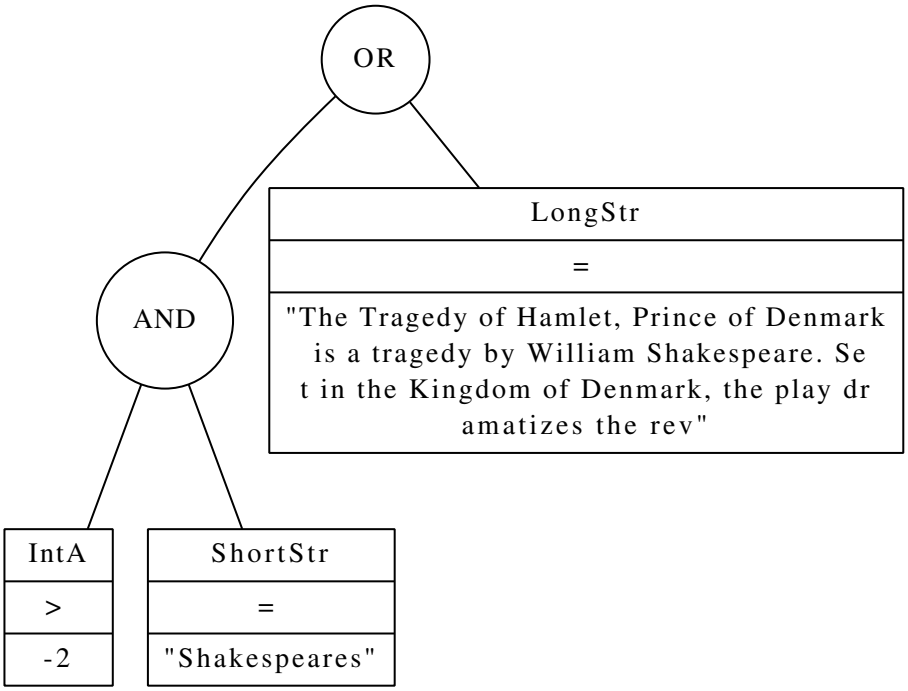

Figure 5.5: Visual feedback from the query planner showing the decision tree from the `WHERE` *clause part of the query:* ***SELECT*** `*` ***FROM*** `table1` ***WHERE*** `IntA>-2` ***AND*** `ShortStr="Shakespeares"` ***OR*** `LongStr="The Tragedy of Hamlet, Prince of Denmark is a tragedy by William Shakespeare. Set in the Kingdom of Denmark, the play dramatizes the rev"`

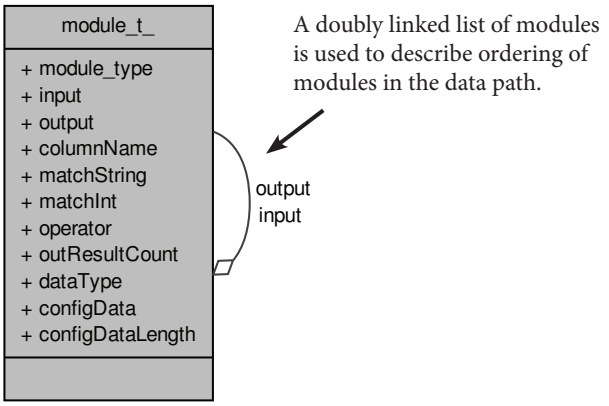

Figure 5.6: Collaboration diagram for the `module_t_` *struct. This data object contains all the information the query planner has on the modules, including the initialization data.*

stream carrying the result records. To avoid deadlocks in the interface application we choose the asynchronous ring buffer stream interface. This allows us to use polling to check if new data has arrived on the streams. Using Algorithm 5.1 to read from the output and control streams ensures that the application continues to fetch results until the query has completed.

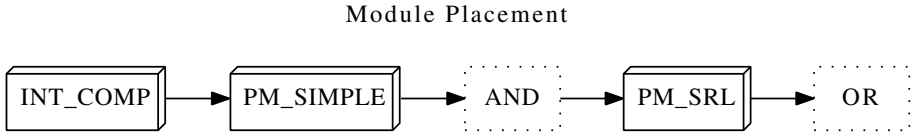

Figure 5.7: Visual feedback from the query planner showing the datapath. The module types and layout is derived from the tree in Figure 5.5. Dotted lines around the `AND` and `OR` modules indicate that they are absorbed by the previous module.

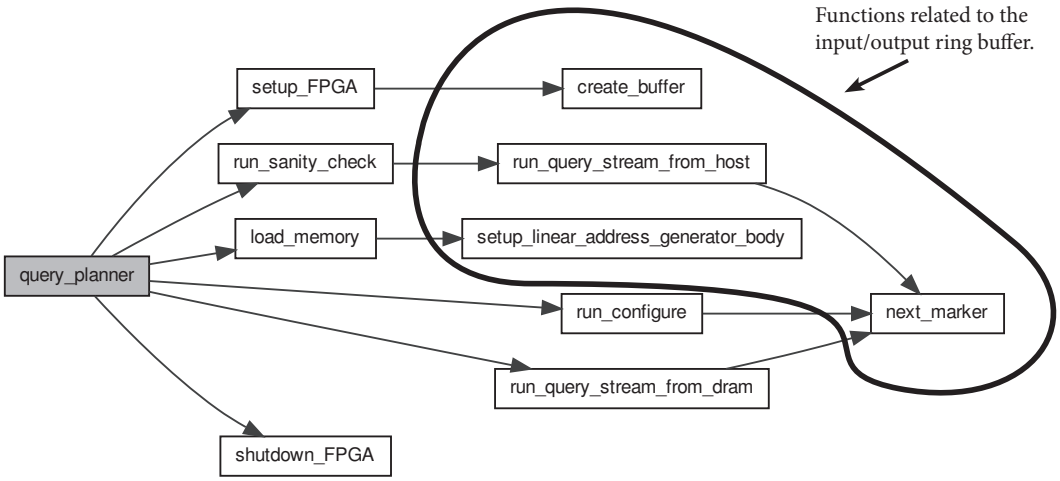

Figure 5.8: Call graph for the interface application showing a typical run.

The first step in getting the hardware operational is to configure the FPGA with the initial bit file. That, and setting up the input and output streams, is done by the `setup_FPGA()` function. For loading tables into the memory the query planner uses the `load_memory()` function. It takes a pointer to a memory location containing the raw table data produced by the query planner and loads it in to a specified location in the MAX3 cards memory.

The next step is to configure the modules onto the FPGA and initialize them. Currently the software does not reconfigure modules, it only writes initialization data to the modules already present in the datapath.

Using the `run_query_stream_from_dram()` function, tables can be streamed from

Algorithm 5.1 Output synchronization algorithm.

```
received ← 0
total ← −1
while total ≠received do
    if output stream has data then
        fetch results
        received ← received + 1
    end if
    if control stream has data then
        total ←stream_item
    end if
end while
```

a memory location. The `run_query_stream_from_host()` function streams tables directly from the host computer. Of course, for accelerating queries only the DRAM streams can be used because streaming from the host computer through PCI-express is too slow. But the host streaming function is still interesting for debugging and testing.

The interface application can also run a sanity check to see if the datapath has been corrupted. This is done by executing a query on a small number of rows that have been hand crafted and that have known results. The sanity check then simply checks the results to see if they match the correct answers.

5.2 Static System

The implemented static system was floorplanned in *single island style* similar to the concept described in Section 4.1.2 and shown in Figure 4.2 on page 29. Due to the 5.5 hours of compile time needed to place and route the single island implementation, and due to challenges regarding the blocker macros, many overnight compilations had to be done. To address the problems with the blocker macro, all the routing attempts were done on the single island style implementation.

Nevertheless, the final implementation of the static system fulfills the conceptual description that was given in Section 4.8. The system includes all the streams sketched in Figure 4.17 on page 50. Ultimately, after a lot of experimenting, the routing in the partial region completed without any violations.

5.2.1 VHDL Implementation

The static system is included in the Maxeler design flow through the inclusion of a *custom HDL node* in the Java code that is compiled with the MaxCompiler. The MaxCompiler was described in Section 2.2, and we learned that is was basically a Java to VHDL compiler. Using a custom HDL node allows us to include our static systems' VHDL description as a component in the Maxeler design flow.

Most of the VHDL code of the static system consists of port mappings and bus assignments. Streams must be connected to the ICAP port, and to and from the connection macros. These connection macros act as anchors inside the partial region in order to force the Xilinx tool to route signals to and from the partial region. The signal assignments between connection macros have to be set up correctly in the VHDL code. With this assignment, we define the physical wire that will later be used to route a specific bit signal.

The static system also has control logic for initiating memory streams and switching between the host and memory as source. It is also responsible for outputting the number of returned data chunks to the host computer upon completion of a query execution. This is output using a dedicated control stream once the last data chunk has been processed. The information is used by the interface application (Section 5.1.2) to synchronize the output results.

Listing 5.1: Defining the interface to the partial region in the VHDL code. Input/output component instantiation.

```
macro : macro_io
port map
(
        clk             => clk,

        in_data         => io_to_first_module_data,
        in_state        => io_to_first_module_state,
        in_results      => io_to_first_module_results,

        out_data        => io_from_last_module_out_data,
        out_state       => io_from_last_module_out_state,
        out_results     => io_from_last_module_out_results
);
```

The port mappings between connection macros have been put in a dedicated VHDL component which is instantiated in the main static VHDL code. This gives us a straightforward VHDL interface to the datapath shown in Listing 5.1. The `macro_io` component file consists of 6500 lines of port mappings, mostly generated by the GoAhead tool.

5.2.2 Memory Management

During the development of this master thesis, an attempt was made to implement a file system for the DRAM to store tables in, but the task was too complex to be completed with satisfactory results within the limited time schedule. The lack of simulation models for the Maxeler system resulted in a situation where debugging had to be done on-chip using Chipscope Logic Analyzer.

With a build time of 75 minutes for every logic change made to the VHDL code debugging was tedious. The idea was to first implement a working file system, and then add pipeline stages to achieve the desired throughput. Looking at the time spent debugging the almost complete file system makes it clear that this is not feasible for one person in any reasonable amount of time.

Another problem with adding this file system to the design was that it consumed a considerable amount of logic and routing resources. This would consume space on the chip, leaving less space for partial regions. It would make routing more difficult and the build time would be considerably longer, or might fail.

The unfinished implementation handles reads reasonably good, it is not far from usable. Implementing reader hardware is simpler than doing its writing counterpart, there are fewer steps involved in reading a file than writing it. A solution could have been to write files directly to the DRAM from the host computer, and only do reads from hardware. But, this would undermine the intent the file system was developed for, namely to allow the hardware to store results and temporary results independently of the host computer.

The Substitute Memory Manager As an alternative to the file system, tables have been stored pre-formatted in the main memory. This was used for maximizing throughput during experiments. Tables are uploaded directly from the host application. The table entries are padded to be aligned with the word and chunk sizes as described in Section 4.8.1 and as depicted in Figure 4.18 on page 52. The uploading is handled solely by the software, it does not involve the custom HDL node.

For managing the stream from memory through the datapath, a memory command writer was implemented. It is controlled from the host computer through the use of scalar inputs[1]. Initiating a memory stream is done from the host by setting the start address and the number of chunks to read, then asserting a `valid` signal that initiates the memory transfer.

The memory command writer issues `memory commands` to the Maxeler memory controller. It requests continuous burst reads of the maximum burst size until the requested number of chunks have been read. The maximum payload size is requested every time because this gives the maximum throughput from the Maxeler memory controller[2]. Because the burst size is 384B and the maximum number of bursts requestable by one memory command is 255, each memory read then results in 384B $*255$ =97.92kB of data. Excess data will have to be discarded by the static system.

5.2.3 Physical Implementation

The on-chip implementation of the database accelerator follows the description from Section 4.1, and consists of a static part and one or more partial regions (a.k.a reconfigurable regions). Naturally, the first attempt in implementing such a design would be to use a single reconfigurable island.

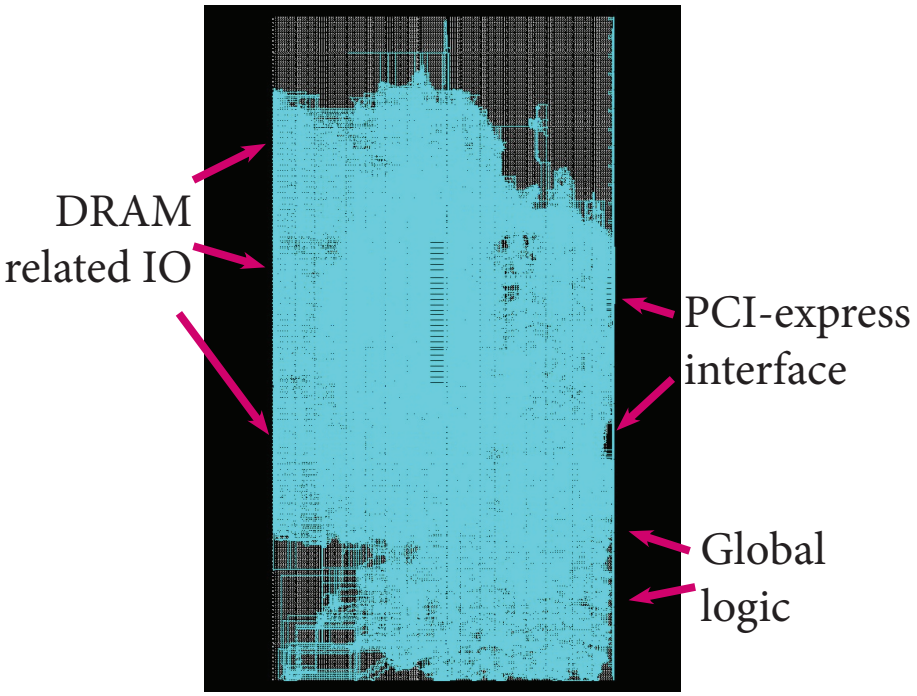

Figure 5.9: An overview of the routing on the FPGA with an example query compiled statically (no partial regions). The light blue colored lines are routed nets (wires).

This raises the question of where to physically place the partial region on the device. For determining this we first look at a statically compiled implementation of the system without any partial regions. By opening the fully routed netlist in the

[1] A scalar input is a MaxelerRT library call that can be used to pass a single operand to a custom hardware module (*HDL node*).

[2] According to Maxeler employees on the Maxeler Developer Exchange message board.

FPGA Editor tool we can inspect the routing and component placements of such a design. Figure 5.9 reveals that the north east corner of the device is virtually empty. Consequently, this is a favorable area to place the first partial region.

In this thesis the GoAhead floorplanning tool was used for creating the user constraints and the connection and blocker macros. The first step towards completing the floorplanning of the static system is to define the partial area in GoAhead's graphical tool. In Section 3.2 we saw that the smallest granularity of reconfiguration on the Virtex-6 device is one resource column wide, and the height of a clock region.

The data bus is 512 bits wide, the state bus is 5 bits wide and the result bus is 8 bits wide. Therefore, the static/partial interface consists of 525 wires in total. Each of the connection macros we are using can connect to four input and four output signals. As a result of this, we need to place $\lceil 525/4 \rceil = 132$ connection macros.

A connection macro occupies one configurable logic block (CLB). The connection macros use double lines for the communication, which connect to every other resource column (CLB, BRAM or DSP). Because of this, two connection macros can be placed adjacent to each other on every row of CLBs on the FPGA. This results in two nested horizontal routing tracks. Using this information, we see that the total number of rows needed to accommodate the connection macro interface is $132/2 = 66$ rows.

By examining the device in FPGA Editor we determine the height of a clock region, then by opening the device in GoAhead we see that this height has 40 rows of CLBs. To place 66 rows we consequently need to use a partial region that has the height of at least two clock regions.

Figure 5.10 shows a full device view of the Virtex-6 chip as it appears in the GoAhead tool's graphical user interface. We have selected an area for the partial region that is the height of two clock regions in the north east corner of the device.

The width of the partial region is selected based on at least three criteria:

- The width of the partial region is proportional to the latency of the nets crossing it. During the routing phase of the static system, these values will have to be within the maximum latency allowed for the given clock frequency.

  There is a trick to circumvent this problem. By placing flip-flops inside the island the long logic path gets divided over several clock cycles. These flip-flops will then later be overwritten by the modules, that themselves act as pipeline stages. But for this first attempt, the region has a width that gives a latency of 3.2ns for crossing nets. This gives a maximum clock frequency that is 312.5MHz, which is sufficient for our purpose.

- Making the partial region wider than the implementation that is shown in Figure 5.10 will result in a non-uniform resource footprint. This means that more implementations of every module will have to be generated to fit to the raised number of possible resource footprints.

- There must be space left for routing on either side of the partial region. The input and output signals needs to be routed vertically along the west side of the island, and because the buses are very wide, this will require a certain amount of resources. Also, the return flip-flops on the east side of the island must be implemented in a column of CLBs.

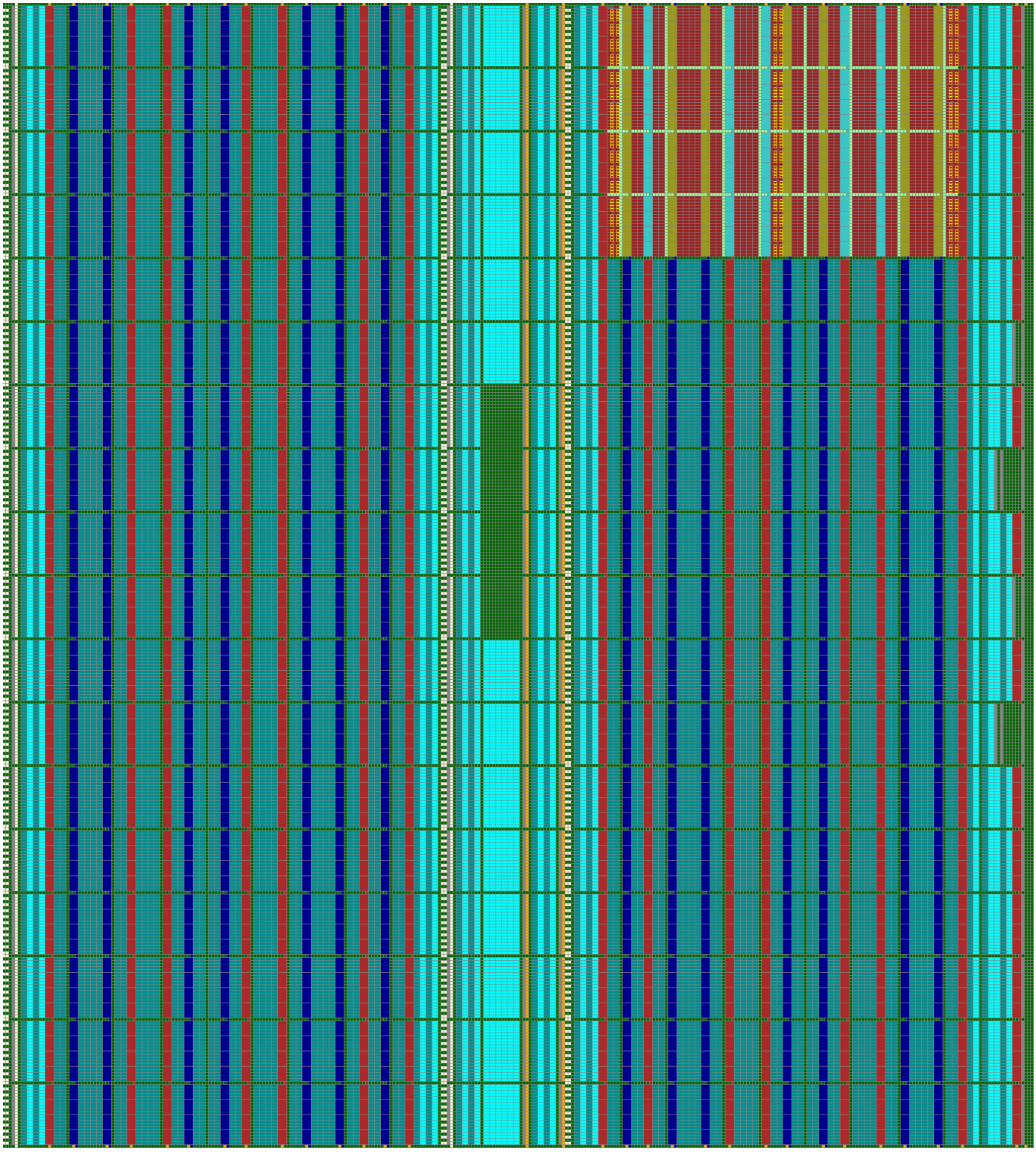

Figure 5.10: Overview of the static system's floorplan as seen in the GoAhead tool. The partial region is the red box in the north east part of the chip.

In the routing scheme for the partial region described in Section 4.1.2 and shown in Figure 4.4 on page 30, the output signals are returned to the input side *through* the partial region. Because each connection macro has the same number of inputs as outputs we organize the routing between the connection macros according to Figure 5.11.

As shown in Figure 5.11, we used three connection macros per routing track. This is necessary because the reconfigurable region has a tunnel for routing signals towards east, and another tunnel to route towards west. Because of the two tunnels, there exist in principle both possibilities, to route first towards east, or to route first towards west. However, due to the routing path between the connection macros CM1 and CM2 (that is completely embedded in the tunnel) this freedom is removed. The router is forced to route into the reconfigurable region in east direction first, while leaving the westward direction for the backward path.

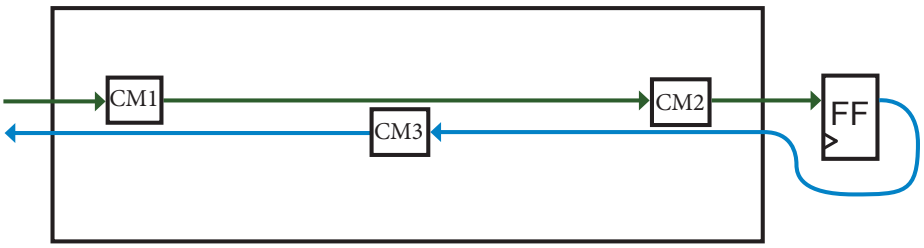

Figure 5.11: The routing scheme between the connection macros in the partial region. The green path will constitute the processing pipeline once the modules have been placed, and the blue path carries the processed results back again.

Implementing the static system in the GoAhead tool is done by using the graphical user interface (GUI). Every action done in the GUI produces a textual command output. These commands can then be copied into a script file that can be used to reproduce all the actions that were performed in the GUI. The final GoAhead script that was used for producing the static system in this project can be found in Appendix on page 102.

A close up view of the partial region as it is seen in GoAhead is shown in Figure 5.12. We can see that the 66 rows of connection macros (marked in yellow) have been distributed over the whole height of the partial region. The three columns of connection macros are identical and implement the routing scheme that was shown in Figure 5.11.

The region marked in red shown in Figure 5.12 marks the partial region, the area GoAhead will produce a blocker macro for. This blocker macro will then be added to the netlist before the routing phase is started. When the router has completed, this blocker macro will be deleted. Then the netlist is re-saved, and the static implementation is complete.

Running the place and route for our implementation of the static system to completion takes about five and a half hours, if the design is routable. A quirk in this process is that it only works if the Xilinx router, `par`, is started with the '`-xe c`' switch. This switch tells `par` to go on forever, even if the router detects an unroutable situation. If we do not specify this switch, the router will exit and output a message telling the user that it has detected an unroutable situation. But from trial and error we know that this is not always the case, so we specify the go-on-forever switch.

Unfortunately, when implementing a new design, sometimes the design *is* un-

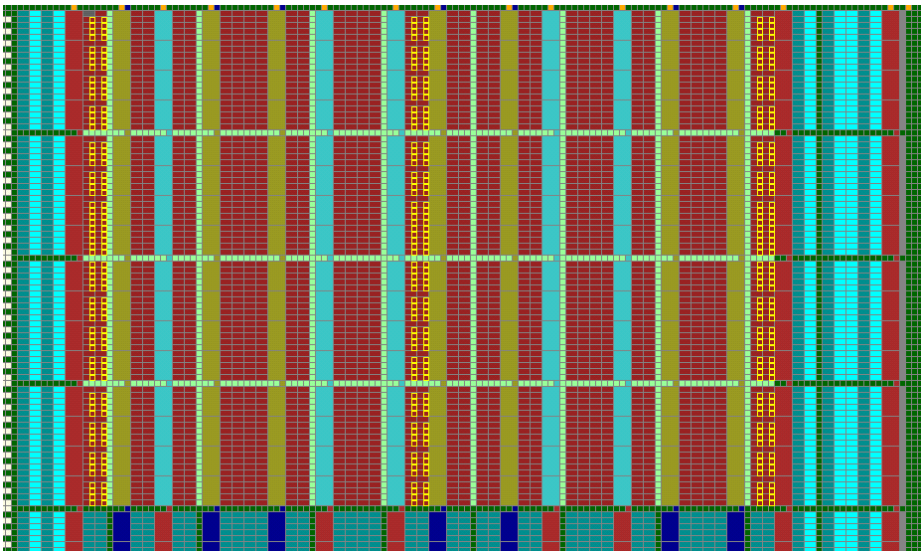

Figure 5.12: A close up view of the partial region as seen in the GoAhead floorplanning tool. The red area is the partial region and the columns of yellow boxes mark the connection macros.

routable because of errors in the blocker macro or because of unfavorable component placements. The problem then is that there is no way predict if the router will complete or not, and the router might continue forever without finding a solution.

The Fully Routed Design Figure 5.13 shows the fully routed static design. In comparison with the non-partial implementation shown in Figure 5.9 on page 65, at first glance, we cannot see that the current routing occupies a larger area. But we can see that the partial region has drawn a lot of the logic from the southern part of the chip towards the north east corner. By examining closer, we see that the areas closest to the partial region are congested with routing. There are still logic resources available in the area.

Figure 5.14 on page 71 shows a close up picture of the partial region. Here we can see the horizontal wires that connect the input to the output. They are composed of the previously mentioned double lines. There is also some vertical routing in the partial region, this is the clock net, a global logic net that connects to all the resources in the partial region.

By the vertical edges of the partial region and in the middle of it we can see the connection macro columns as a slight accumulation of (light blue) routing. This figure shows the same snapshot of the device as the picture from GoAhead in Figure 5.12, only this is the actual physical implementation of it.

Figure 5.15 on page 71 shows a close up of the western border of the partial region. The two middle columns of LUTs (blue boxes) are the connection macros. The blocker macro has been manually deleted, so the occupied wires in the partial region are double wires that go between the connection macros, or they belong to the clock net.

The V-shaped wires inside the partial regions are the double wires entering the switch boxes. There are two sets of double wires in each direction. These two sets

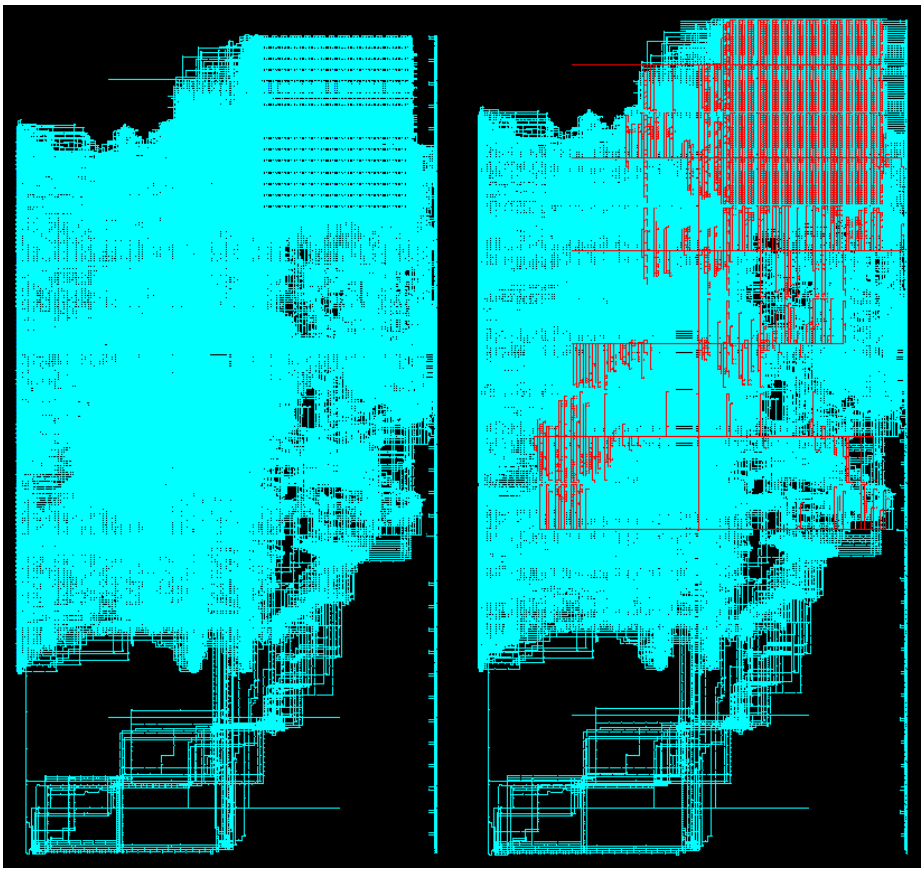

Figure 5.13: The fully routed netlist of the static system opened in the FPGA Editor. All the blue lines are used wires (nets). The leftmost picture shows the routing with the blocker macro deleted. The righmost show the clock net that goes to the partial region marked in red. More details are shown in Figure 5.14 and Figure 5.15.

are interleaved by one column, so that one set always bypass the switchboxes that the other set connects to. This can also be seen in Figure 5.15. At every V-shaped net a set of double wires go into the switch box, while the other set bypasses it on top of the V-shaped routing.

5.2.4 Placer Tool Problem

The Xilinx design flow from *synthesis* to *place and route* is shown in Figure 5.16. In the Map phase design elements get mapped to device resources. Traditionally placement of these elements on the device was done in the PAR phase, thus the name *place and route*. For legacy reasons PAR has kept its name even though placement is now done in the Map phase by the MAP tool.

The design flow for creating the static system containing a partial region is listed on page 27. A notable quirk with this design flow is that a wire blocking hard macro is placed over the partial region *after* the placement phase. This means that all components are placed on the device, but not in the partial region because of the user constraints prohibiting resource usage in this area. After the completion of the MAP tool, a blocker macro is added to the netlist. This blocker macro is a dummy net that occupy all wires that we do not want being used for routing. The blocker

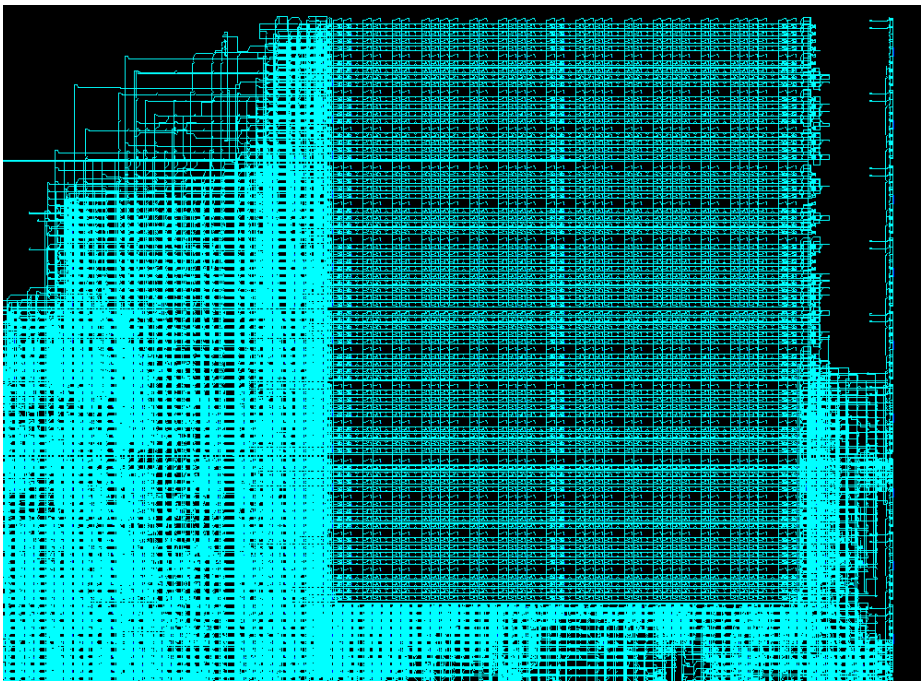

Figure 5.14: A close up view of the partial region from Figure 5.13. All the blue lines are routed wires (nets).

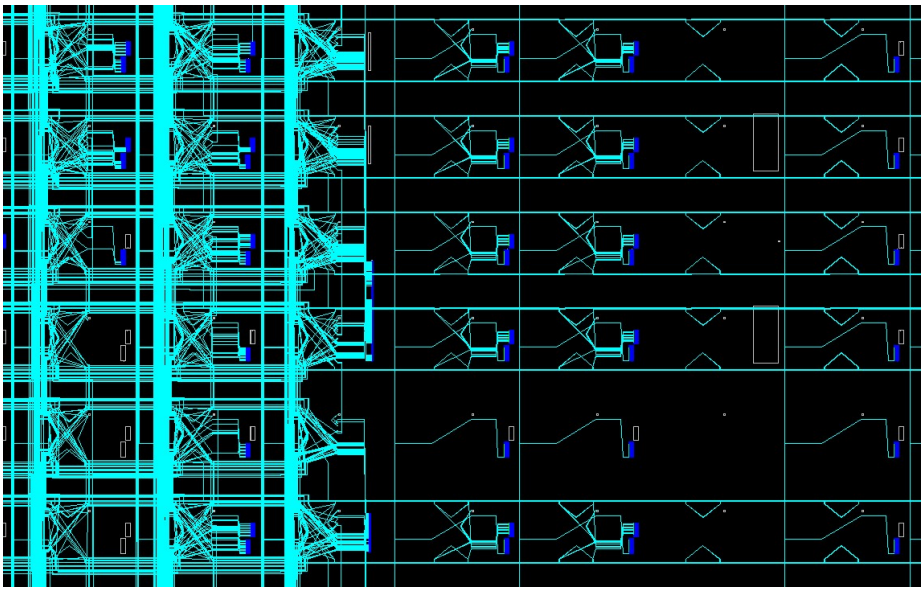

Figure 5.15: A close up view of the western edge of the partial region from Figure 5.14. The two columns of blue boxes slightly to the right of the center are the connection macros. The lines and V-shaped lines to the right of them are the double lines that are used for the static/partial interface and for the return signals. The V-shapes are the double lines entering the switch boxes.

macro effectively acts as a prohibitor for routing. This is necessary because no such wire prohibit statement exists in the regular user constraints.

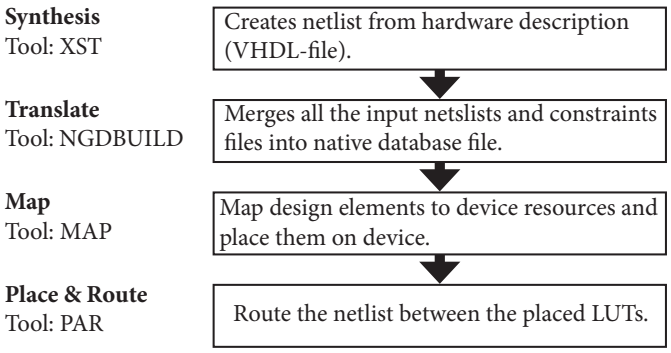

Figure 5.16: An overview of the Xilinx design flow.

The placer tool sees a region on the FPGA that it cannot place logic in, but that apparently can be routed through. The placer tool will therefore attempt to place the components in relation to each other so that the routing distance is as small as possible, as long as all user constraints are met. The placer will consequently not take into account that routing through a partial region will be prohibited for the static routing due to the blocker macro.

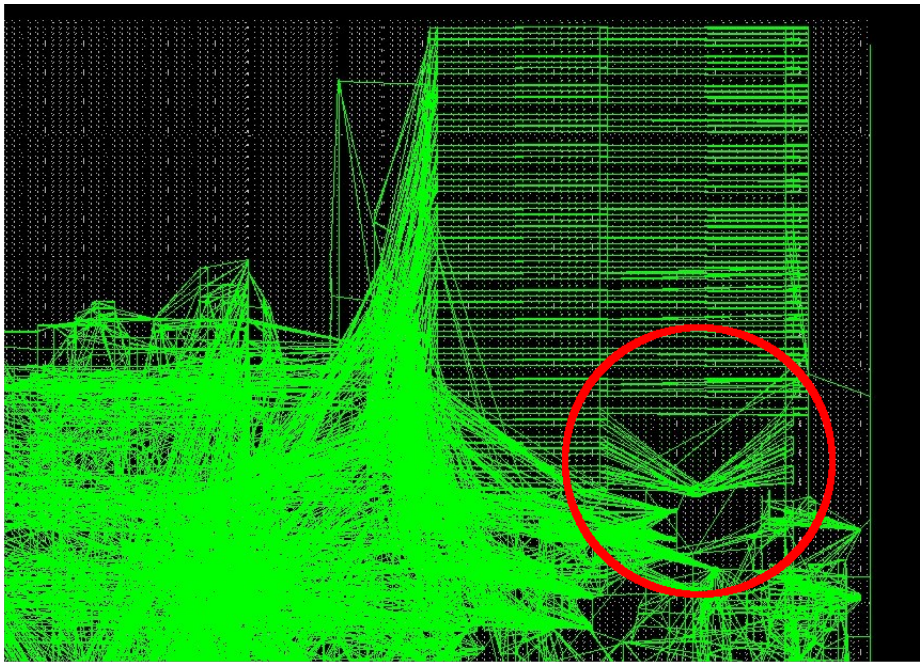

Figure 5.17: A picture from FPGA Editor showing unrouted in the area of the partial region. The circled area contains an erroneous placement of components. An explanation for this is illustrated in Figure 5.18.

An example of such a problem is depicted in Figure 5.17. The image is a screenshot from the FPGA Editor tool displaying the unrouted netlist of the static system. It is zoomed in on the partial region of the static design. The green lines are unrouted nets, and the grey boxes and dots in the background are resources such as block RAM and slices.

Inside the red ring in Figure 5.17 an unlinearity in the grid of unrouted nets can be seen. The only possible path into the partial region will be on the east side of it,

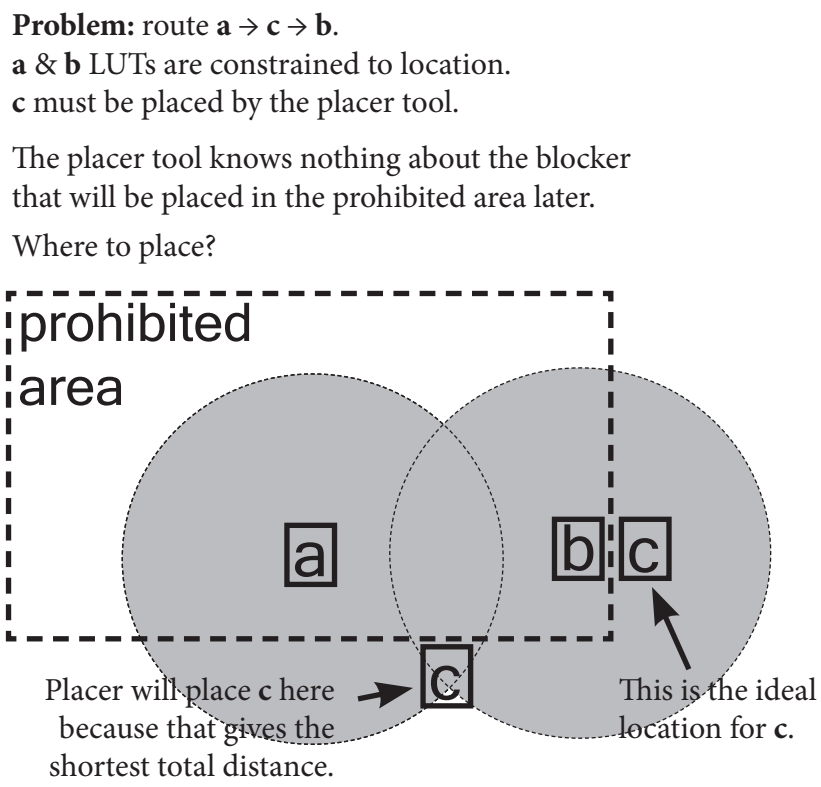

Figure 5.18: An explanation of the placer tool problem.

not on the south side where the components have been placed. Figure 5.18 shows an illustration of what has happened. The return flip-flops have been erroneously placed because the placer tool does not know about the blocker that will be placed there after the Map phase has finished. The result is a netlist that is unroutable. A large number of wires would have to be routed around the corner of the partial region, and back again.

The apparent solution to the specific problem in 5.17 is to add the return flip-flops to an area group. An *area group* is a user constraint that tells the placer tool that a specific component must be placed within a specific region on the device. Adding the return flip-flops to an area group that is adjacent to the east end of the partial region would force the placer tool to place them where we know that they belong.

These problems can be challenging when working with large designs such as our static implementation. There are 525 bit wires streaming through the partial region in each horizontal direction. This draws a lot of logic towards the input and output of the partial region. Some of the logic belongs to our design and some is part of the Maxeler system. Many of these components have to be area constrained in order to get a fully routed design that also meets timing requirements. This was particularly challenging because of the very long tool run time.

The floorplanning has to be done on a component level. Some of the logic might have to be split into different VHDL components to be area constrained. This was the case with the return flip-flops in the example. They were originally a ten-line piece of code before they were placed in a dedicated component. This return flip-flop component was then instantiated in the original VHDL file.

5.3 Partial Modules

Our module library consists of three modules, the *integer compare module*, the *simple pattern match module*, and the *long string pattern match module*. The modules were implemented in VHDL and functionally verified in a simulation environment using Modelsim. Modelsim is a digital simulation and verification tool developed by Mentor Graphics.

The modules were also statically compiled for selected queries, and the on-chip functionality was verified by the use of Chipscope. Chipscope is an embedded software based logic analyzer that is part of the Xilinx tool chain. It is capable of capturing the value of internal vectors in an FPGA and presents them to the user as waveforms.

5.3.1 VHDL Implementation

Modules can be connected in any order to form a datapath, so they must all share the same entity (Listing 5.2). The `in_data` and `out_data` buses form the 512-bit wide datapath that is streamed through the modules. In addition, it is also used for initializing the modules during the module initialization phase.

Listing 5.2: Module entity

```
entity module is
port
(
  clk         : in std_logic;

  in_data     : in std_logic_vector(511 downto 0);
  in_state    : in std_logic_vector(4 downto 0);
  in_results  : in std_logic_vector(7 downto 0);

  out_data    : out std_logic_vector(511 downto 0);
  out_state   : out std_logic_vector(4 downto 0);
  out_results : out std_logic_vector(7 downto 0)
);
end module;
```

The `in_state` and `out_state` signals are state signals that are propagated along with the data chunks. It is a 5 bit signal where the first 27 number values translate into chunk numbers. The top five number values are special states for signaling idle cycles, init states, query done, or reset. The exact state values and other constants are specified in a VHDL library common to all modules. An excerpt from this library is shown in Listing 5.3.

Listing 5.3: State signals (module_pkg.vhd)

```
constant CHUNK_0 : std_logic_vector(4 downto 0) := "00000";
constant CHUNK_1 : std_logic_vector(4 downto 0) := "00001";
constant CHUNK_2 : std_logic_vector(4 downto 0) := "00010";
constant CHUNK_3 : std_logic_vector(4 downto 0) := "00011";
constant CHUNK_4 : std_logic_vector(4 downto 0) := "00100";
constant CHUNK_5 : std_logic_vector(4 downto 0) := "00101";
constant CHUNK_6 : std_logic_vector(4 downto 0) := "00110";
constant CHUNK_7 : std_logic_vector(4 downto 0) := "00111";
-- ... we dont name the rest of the chunks.
-- names are only for debugging, except the last
constant CHUNK_LAST     : std_logic_vector(4 downto 0) := "11010";
constant IDLE : std_logic_vector(4 downto 0) := "11011"; -- 27
constant INIT : std_logic_vector(4 downto 0) := "11100";
constant INIT_PHASE2 : std_logic_vector(4 downto 0) := "11101";
constant DONE : std_logic_vector(4 downto 0) := "11110";
constant RESET : std_logic_vector(4 downto 0) := "11111";
```

All modules consume and output one data item every clock cycle. The data bus, state bus and result bus are synchronized and propagated together. This means that if the data signal is delayed by `n` clock cycles the result and state signals also must be delayed by `n` clock cycles.

Result Signals The `in_results` and `out_results` form the intermediate result signal path. Every module produces one result signal. In Section 4.5 we learned that the number of intermediate signals between two modules is equal to the tree height at the particular node in the corresponding expression tree. A module takes in results from the previous modules. These results are either consumed by this module or another result is added to it before it is forwarded.

Initialization The concept of initialization was described in Section 4.6.1 and meant setting the internal registers of the modules. After the modules have been physically placed on the device, they have to be initialized. This is achieved by using the special `INIT` and `INIT_PHASE2` state signals, and writing the configuration data through the data bus.

Configuration of a module always start with the input state being set to `INIT` and the initialization data for the specific module set on the data bus on the same clock cycle. The 512 bits wide data bus is then interpreted as 16 32-bit initialization words, where each word position has a unique meaning to the modules. For example, one word is used as an address field to individually access the modules.

The DONE State The `DONE` state is pushed onto the datapath immediately after the last input chunk has been put on the datapath. To the modules this state means exactly the same as an `IDLE` state would have meant. This state is used by the static system to flush the pipeline that the datapath constitutes. When the `DONE` state emerges at the output of the datapath it signals that the last chunk has been processed. Then the static system will output the number of returned chunks so that the stream can be synchronized by the interface application as described in Section 5.1.2.

Idle Cycles Since the input to the wrapper can run out of data, and the output can stall at any time, the datapath must have some kind of flow control. In the wrapper it is advantageous that the datapath is designed to work without back

pressure. Instead, there is a special IDLE state that is fed into the datapath when there is no input data. In the modules, this IDLE state acts as a no-operation operand, causing them to do no internal changes. Having the modules accept a new data item every clock cycle means that the flow control problem can be handled entirely by the wrapper, and thereby simplifying the module design.

5.3.2 Speed Optimization

The unoptimized modules had maximum speeds ranging from 175MHz to 224MHz. These numbers are based on the latency reported by the Xilinx synthesis tool. The real maximum frequencies once the modules have been subject to place and route can differ from these. The unoptimized modules are limited in speed by the long logic path from the input to the output. Worst case, because the modules are confined to a tall, narrow area the real maximum speeds could be much lower.

Since there is no way to run timing verification on dynamically reconfigured modules we must ensure that the modules are fast enough to avoid timing violations. First of all every module must be isolated in its own beginning to end latency. This is best done by putting registers at the inputs and outputs so that the first thing on the input side of a module is a flip-flop. Instead of forwarding the output signals directly to the next modules it goes into registers on the output side, and from there out of the module. This consumes logic and adds two clock cycles to the pipeline, but it ensures correct operation.

Component	Min period	Max freq.
module_int_compare	4.459ns	224.248MHz
module_pattern_match_simple	4.929ns	202.901MHz
module_pattern_match_srl	5.714ns	175.000MHz

Table 5.1: Component's timing scores before optimization.

Component	Min period	Max freq.
module_int_compare	1.956ns	511.365MHz
module_pattern_match_simple	2.724ns	367.148MHz
module_pattern_match_srl	2.966ns	337.103MHz

Table 5.2: Component's timing scores after optimization.

To speed up the slow paths inside the modules pipeline stages were added on strategic places, attempting to divide the long paths into smaller ones. Three pipeline stages were added to the Long String Pattern Match Module, two to the Integer Compare Module and one stage was added to the Simple Pattern Match Module. After these changes the Xilinx synthesis tool reports the speeds of the modules to be between 337MHz and 511MHz. Table 5.1 shows the maximum frequency for the unoptimized modules and Table 5.2 shows the improvement after pipeline stages have been added.

With this optimization, the target clock frequency of 300MHz could be achieved for any combination of these modules. These have consequently a processing throughput of close to 19.2GB/s.

5.3.3 Physical Implementation

The floorplan for generating the partial modules was created with the help of the GoAhead tool. The whole concept of connection macros relies on the static region and the partial modules having compatible interfaces. Therefore, the same connection macro placements were used as in the static system. The difference is that they were moved to the area outside the partial region like the concept described in Figure 3.3 on page 24.

Because of the unreasonable long time place and route takes on the large device we are using, the modules were generated using a device description for the smallest Virtex-6 device available. The resource footprint is still the same as on the large device, so the modules can be routed on this smaller device to save time. Subsequently, all the logic and routing belonging to the reconfigurable modules can be *cut out* by using a function in the GoAhead tool. Then differential bitstreams can be generated ready for transfer to the ICAP configuration port of the FPGA.

The process of physically implementing the modules in GoAhead is in a sense a mirror process of the procedure for implementing the static region. Instead of blocking the wires in the partial region, the wires that are *not* in the partial region must now be blocked. In addition, the placer must be prohibited from placing components *outside* of the partial region. The last task is accomplished by a user constraint instructing the placer to place the module components inside the partial region. This constraint can be generated by GoAhead.

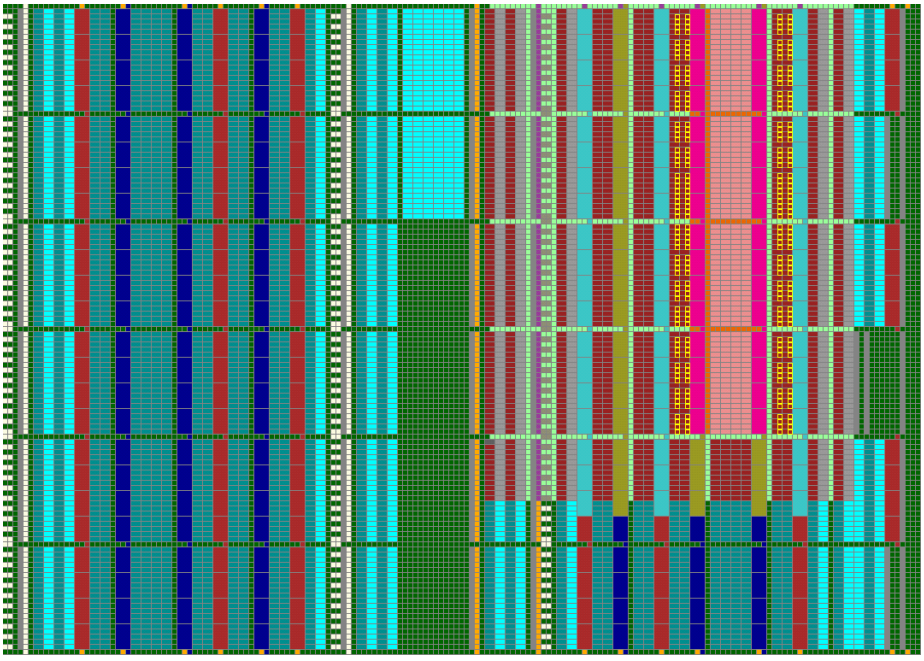

Figure 5.19: A view of the smallest Virtex-6 device in GoAhead. The device is floorplanned for a module implementation.

The task of blocking the wires in the entire region *outside* of the partial region would mean that a very large blocker macro would have to be generated. This can also be done by GoAhead, but it is actually sufficient to use a blocker macro that only extends a short distance away from the partial region.

On Virtex-6 FPGAs, local wires have different lengths. The longest distance that

can be reached from one switch matrix using local wires is another switch matrix in a radius of four switch matrices. Consequently, a fence being at least four switch matrices wide will ensure that no routing outside the module will be used. Furthermore, Virtex-6 FPGAs provide special long lines. If these routing resources shall be blocked, the fence might be designed wider, as a long line connects every switch matrix along its way. However, GoAhead provides commands for automatically creating blocker fences.

The layout used to generate the modules can be seen in Figure 5.19, which is a screenshot from the GoAhead tool. Here we can see the partial region in a lighter red color surrounded by a blocker macro fence, which is the darker red color. The connection macros are now placed just outside of the partial region along the vertical borders. Placing them so close to the border puts more pressure on the placer tool to place components closer to the connection macros that they are connected to.

The GoAhead script used to generate the placement constraints and the blocker macro for the partial modules can be viewed in Appendix on page 105.

Figure 5.20 shows the nets in the fully routed design. This particular module is an integer compare module. The right and left side of the image show exactly the same area of the device. On the left side the blocker macro is still in place. On the right side the blocker macro has been deleted.

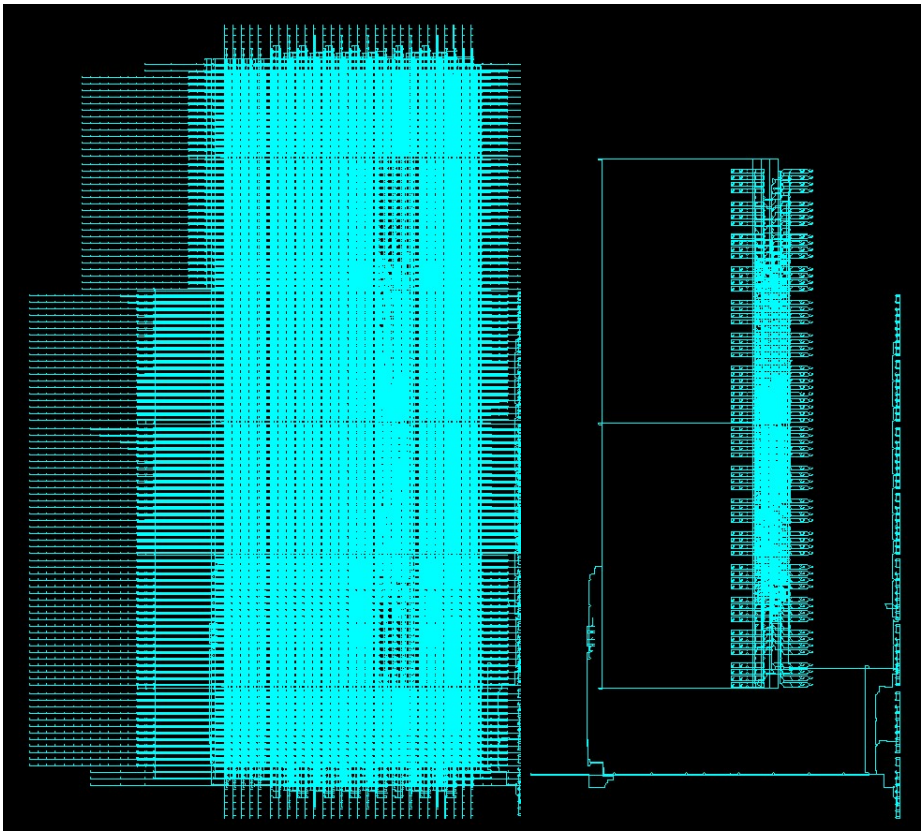

Figure 5.20: A screenshot from FPGA Editor showing the nets of the fully routed integer compare module. On the left side the blocker macro is still present. The right image shows the module after the blocker macro has been deleted.

On the right side of the image we can see the space the module occupies. It is the tall narrow area almost saturated with routing. The horizontal wires protruding

from the module are the double lines that go to the connection macros. These areas on both sides of the module will not be part of it once it has been cut out, only the wire routing will be preserved.

The blocker macro is not meant to be removed before the blocker is cut out of the design, it is removed to allow us to inspect the routing belonging to the module. In fact, if the blocker macros are deleted the accelerator would not function at all. In Section 4.1.2 we concluded that the best way to get the output signals back to the input side of the partial region, was to route them back *through* the partial region.

The 'tunnels' that go through the partial region must also be present in every module for it to work. If not, configuring a module onto the datapath would create a gap in the routing. The return signals use double lines, like all the routing to and from the connection macros does. The GoAhead tool can be used to add these double wires to the blocker macro. In this case, the module would provide the routing for the return datapath as a part of the blocker net.

Figure 5.21 shows a close up view of a switch box somewhere inside the module. The wires marked red belong to the blocker macro. The V-shaped red arcs in the picture are the return wires. In the picture there are other wires marked red that are not related to the return wires. These are other, longer lines that have to be blocked to prevent the router from exiting the partial region. These wires can also be seen on the far left side in Figure 5.20, extending from the blocker.

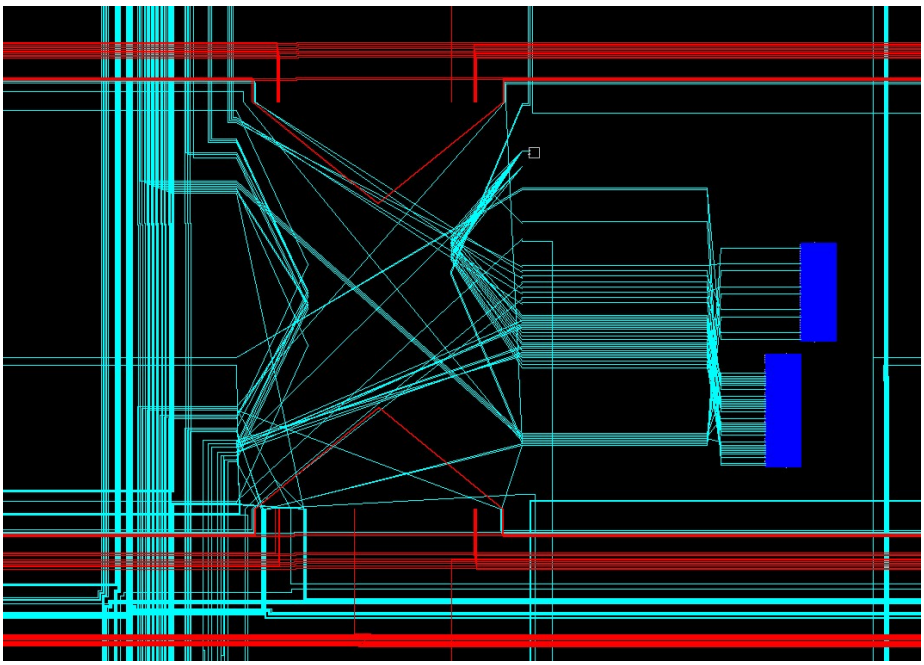

Figure 5.21: A view of a switch box and two slices inside the module. The nets marked in red belong to the blocker macro. The V-shaped red nets are the double lines entering the switch box.

These methods were applied to implement all three modules. Figure 5.22 shows a comparison of these three modules. The integer compare module on the left and the simple pattern match module in the middle have the same width. The long string pattern match module on the right is wider than the other two modules. Because it is more complicated, and thus requires more logic and *routing* resources.

Examining of the implemented modules, we see that we are bound by the *routing* resources, not the logic resources. The modules are tall and narrow, consequently they require a lot of vertical nets to be routed. The shortage of routing resources is one of the challenges encountered when dealing with very large vectors in FPGAs.

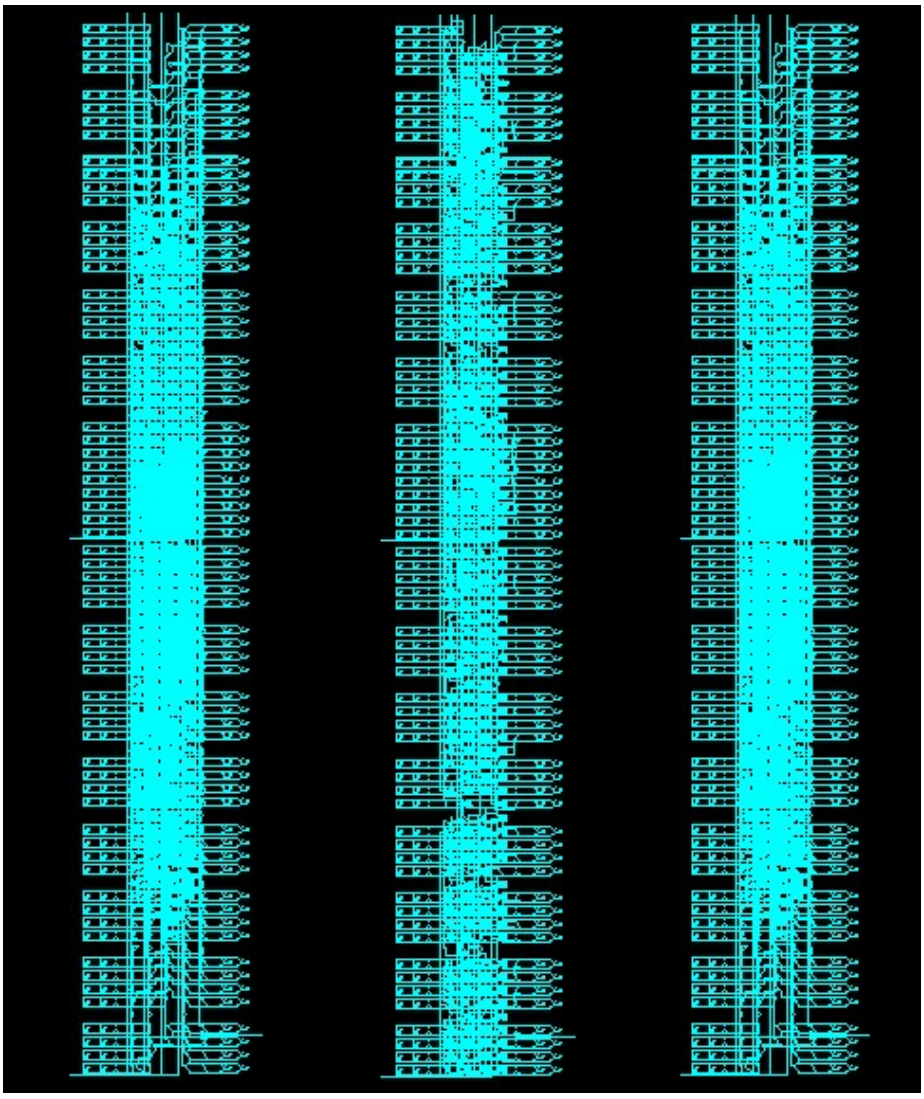

Figure 5.22: From left to right, integer compare module, simple pattern match module and long string pattern match module.

Chapter 6

Results

To partially reconfigure an FPGA, a differential bitstream is required. The differential bitstream describes the logic changes that needs to be made on the FPGA. Due to issues with the relocation of modules, the differential bitstream generation was not successful. Without the correct differential bitstream, the reconfiguration will not work correctly.

The problem is related to the relocation of modules from one placement on the device to another. All modules are built once for every resource footprint, and thus they have specific placement values in the netlist. In our case, they were built on a smaller device with the same resource footprint to save time. Because of this, the module netlist must be 'cut out' from the device netlist that it was built in before the differential bistream can be generated.

The GoAhead tool provides a function for relocating modules. Unfortunately, what worked on the Xilinx Spartan-6 FPGAs did not work correctly on the Virtex-6 architecture that was used in this thesis.

Debugging the GoAhead tool is beyond the scope of the author of this thesis, and the solution to this problem came too close to the thesis deadline for it to be implemented. Therefore the tests were performed on an implementation where the three modules had been compiled statically.

However, the static system was compiled with a stream interface to the internal configuration access port (ICAP) in the FPGA. In Section 3.4 we learned that the Xilinx FPGA Editor tool could be used for creating differential bitstreams for testing purposes. This was done earlier in the project to test the functionality of the ICAP port.

A differential bitstream that changed a logic AND gate to an XOR gate was generated with the use of FPGA Editor. Reconfiguring this simple logic gate in a test design verified that the connection to the ICAP port was working correctly.

Therefore there is reason to believe that it would have worked if the correct differential bitstream had been available. In that case, the results from the following tests would have been *exactly the same.*

A Case Study A query consisting of three modules was compiled. The query contained an `AND` operator, an `OR` operator, a '>' operator and two '=' operators. The '=' operations were performed on string values. One record in the data set (table) spans over three chunks.

The decision tree for this query is similar to the one in Figure 5.5 on page 61. One string is 12 bytes long, while the other is 136 bytes long. This results in a datapath consisting of one module of each sort, and a module placement like the one shown in Figure 5.7 on page 62.

To monitor the execution of the test queries the accelerator was compile with selected signals of interest mapped to an IP core from Chipscope. This enables us to capture the value of internal vectors while the query is executing. The captured data is then downloaded via a JTAG-cable and displayed as wave plots in the Chipscope Logic Analyzer tool.

A Needle in a Haystack In this test we will initialize the modules with matching patterns that only match one record in the entire table. The accelerator must find the one record that matches, the needle in the haystack, and output it to the host application.

A data set consisting of 100,000 chunks (33,333 records) was used. This is equal to $100,000 * 64\text{B} = 6.4\text{MB}$ ($6.4 * 10^6$) of data. Because 131,072 samples is the maximum a Chipscope core can capture in one run, this is about the greatest data set we can use that will fit in this window.

Figure 6.1 on page 84 shows a wave plot from this test. The circled numbers in the figure correspond to numbers in the following list:

1. The query is initiated by the `scalar_dram_addr_valid` signal. This tells the accelerator to start streaming from the DRAM.
2. The accelerator immediately starts to request data from the memory by issuing memory commands.
3. The memory controller responds by deasserting the `empty` signal. Data is then streamed through the module pipeline as fast as the memory controller can produce it.
4. Here, the matching record is found. The `from_last_module_results` signal changes value to tag the matching record. When the static system detects this, the matching record is output to the host computer.
5. The query ends when the last record has been processed and the number of matching chunks is returned to the host application over the control stream.
6. The memory controller keeps outputting data even though the last record has been retrieved. This happens because the accelerator always requests the maximum payload size from the memory controller to maximize throughput. This excess data is discarded by the accelerator.
7. The software part of the accelerator decides that the query has completed and deasserts the `scalar_dram_addr_valid` signal. The system is now in the same state as before the query started.

The query processing took a total of 102354 clock cycles. The time spent processing the query at 100MHz is $(100 * 10^6)^{-1} * 102354 = 1.02354\text{ms}$. In this time 6.4MB of data was processed. This gives us a throughput of $(1.02354 * 10^{-3})^{-1} * 6.4 * 10^6 = 6.25\text{GB/sec}$.

This test was run at 100MHz, but the maximum speed the Maxeler system could support is 300MHz. In Section 2.1 we said that the Maxeler system can provide a memory stream of up to 38.4GB/s. If this is true, and the memory speed scales linearly, upping the frequency would give us a throughput of $3 * 6.25\text{GB/s} = 18.75\text{GB/s}$.

This is close to the theoretical maximum of 19.2GB/s that we determined in Section 4.6.2.

Limited by the PCI-express Interface In this test we used the same hardware setup as in the previous one, but the match values in the query have been slightly altered so that every other row is a match.

To fit the entire sequence in one Chipscope run, a data set of only 10,000 chunks (3,333 records) was used this time. This equals 640kB ($640 * 10^3$) of data. This time the amount of matching records accumulates faster than the PCI-express interface can stream them to the host computer.

Figure 6.2 on page 85 shows a wave plot of a run with this setup. The circled numbers in the figure corresponds to the numbers in this list:

1. The query is initiated by the `scalar_dram_addr_valid` signal. This tells the accelerator to start streaming from the DRAM.
2. At first the memory stream is going through the module pipeline at full speed.
3. The accelerator is producing results at a faster rate than the PCI-express interface can output them. At this point the output buffer is full, and the flow control hardware asserts the `output_stall` signal. The static system propagates idle cycles through the module pipeline when there is no valid data at the input.
4. All the 3,333 records have been streamed through the module pipeline. The number of matching signals is returned to the host computer.
5. The host computer deasserts the `scalar_dram_addr_valid` signal, and the accelerator is back to its original state.

The accelerator spent 108738 clock cycles processing the data, which means that the time spent processing the query which ran at 100MHz was $(100 * 10^6)^{-1} * 108738 = 1.08738$ms. That is almost the same as the time spent in the previous test for processing ten times the amount of data. This gives us a throughput of $(1.08738 * 10^{-3})^{-1} * 640 * 10^3 = 588.5$MB/s.

The reported output speed of the PCI-express interface was 2GB/s. Only every other record in this test was a match, this means that the PCI-express interface must have been transferring data at about half of the throughput rate. 294MB/s is only about 15% of the promised speed.

The decreased speed is probably due to the asynchronous ring buffer that is used by the interface application. The decision to use this interface was explained in Section 5.1.2. Previous experiments have shown that the PCI-express speed comes much closer to the theoretical maximum when transferring directly from the DRAM while *not* using the ring buffer.

A solution to this problem could be to let the accelerator stream the results *back* to the memory. Once the query has completed, the host application will then have to fetch the results from the memory. Another solution could be to define a minimum 'packet' size for the output stream. Then there would have to be some kind of meta data in the packets which the host application could use to locate the end of the stream.

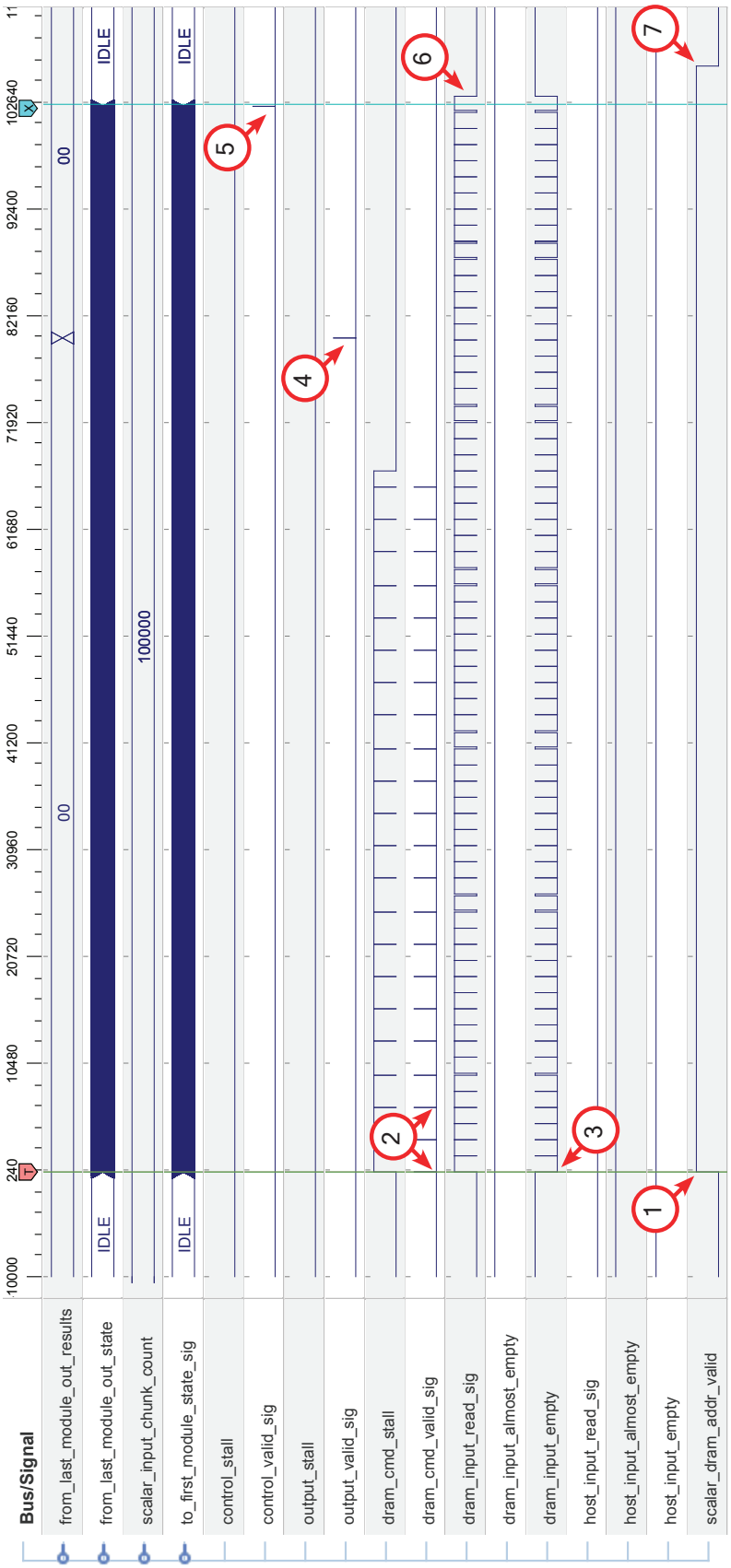

Figure 6.1: A Chipscope wave plot of the accelerator finding a needle in a haystack, one match among 33,333 records spanning over 100,000 chunks. Processing took 102354 clock cycles at 100MHz. That is equal to a throughput 6,25GB/sec.

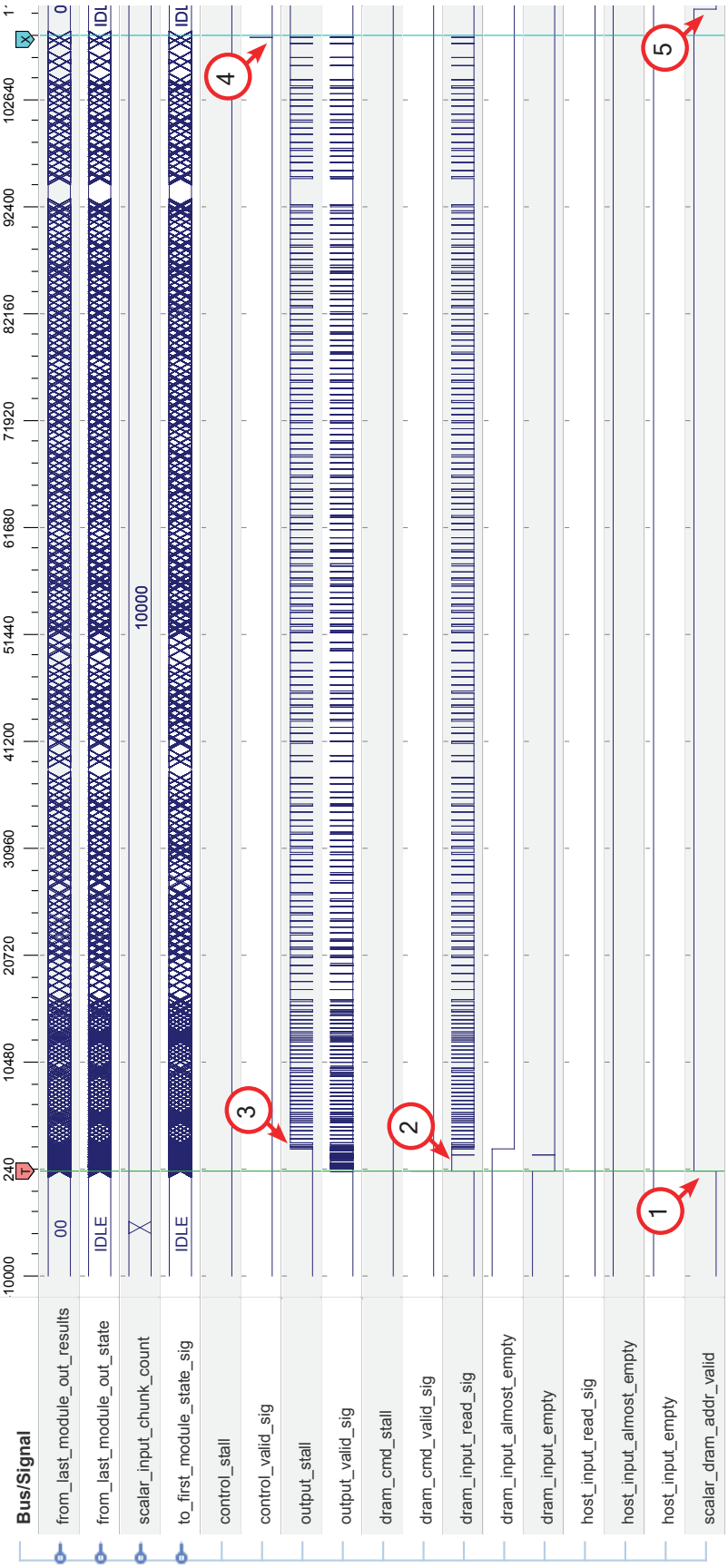

Figure 6.2: A Chipscope wave plot of the accelerator processing a stream where every other record is a match. Output is limited by the PCI-express speed. Processing 3,333 records spanning over 10,000 chunks at 100MHz took 108738 clock cycles. That is equals a throughput of 588,5MB/sec.

Chapter 7

Conclusion

The demonstration in the previous chapter showed that a query could be processed at 6.25GB/s. The memory controller supports speeds of up to 38.4GB/s, and all three modules also support clock frequencies of more than 300MHz. Therefore it is likely that an implementation of our static system running at 300MHz could achieve a throughput of more than 18GB/s.

The results suggest that for some queries the bottleneck is the interface to the host computer. Even with the full 2GB/s, the PCI-express would still become a bottleneck for some queries. The newest solid state disk RAID arrays used in conventional databases are also in the gigabyte per second range. Accordingly, the FPGA accelerator cannot compete solely based on throughput for all usage types.

Performance gain depends heavily on the use case. A more complex data filtering query with many operators would consume more CPU-cycles in a software database, thus slowing it down. However, the throughput in our accelerator would not be negatively affected by adding more logical or relational operators. On the contrary, as long as there is enough space in the partial region to fit all the needed modules, filtering out results would actually speed it up due to less data that needs to be transferred.

The ultimate goal of this thesis was to dynamically stitch modules together at run-time to form a query processing datapath in the FPGA. Although replacing the modules dynamically at run-time was not achieved by the deadline for this thesis, it has been demonstrated that FPGAs *can* be used to accelerate data filtering queries.

We have also shown that a set of generic modules can implement a larger set of SQL operator through the use of a custom initialization protocol. Furthermore, we have shown that the `WHERE` clause part of a query can be translated into a sequence of modules forming a query processing pipeline.

The findings in this thesis indicate that database problems involving complex filtering operations can benefit from this kind of database accelerator.

7.1 Future Work

This thesis has merely scratched the surface on the topic of database acceleration using partial run-time reconfiguration, but it has demonstrated its potential. There is still a lot that could be researched within the subject, starting with how to implement the rest of the SQL operators.

For implementing SQL operators outside of the `WHERE` clause as modules, temporary buffering in the DRAM is needed. If the static system could handle temporary buffering, implementing data aggregate functions like the ones found in the `HAVING` clause would be the next natural step.

Also, processing the `JOIN` clause and the `ORDER BY` clause requires buffering of temporary results. These functions also require sorting functionality to be implemented in the accelerator. This would require having two streams passing the modules simultaneously. One solution could be to define another mode of operation for the datapath, where the 512 bits wide datapath is divided among two 265 bits wide streams.

For further development of the findings presented in this thesis, the author recommends using the concepts described from Section 4.5 and the rest of Chapter 4. It is also recommended to read Chapter 5 and take note of problems that are particularly challenging when designing a partially reconfigurable system using very wide vectors.

Bibliography

[1] T. Becker, W. Luk, and P.Y.K. Cheung. Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration. In *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pages 35–44, april 2007.

[2] P. DuBois. *MySQL: the definitive guide to using, programming, and administering MySQL4*, page 102. Developer's library. Sams Publishing, 2003.

[3] Cindy Kao. Benefits of partial reconfiguration. *Xilinx Xcell Journal*, 55:65–67, 2005.

[4] D. Koch, C. Beckhoff, and J. Teich. Bitstream Decompression for High Speed FPGA Configuration from Slow Memories. In *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, pages 161–168, dec. 2007.

[5] Dirk Koch and Jim Torresen. FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pages 45–54, New York, NY, USA, 2011. ACM.

[6] Dirk Koch, Jim Torresen, Christian Beckhoff, Daniel Ziener, Christopher Dennl, Volker Breuer, Jurgen Teich, Michael Feilen, and Walter Stechele. Partial reconfiguration on FPGAs in practice - Tools and applications. In *ARCS Workshops (ARCS), 2012*, pages 1–12, feb. 2012.

[7] René Müller. *Data Stream Processing on Embedded Devices*. PhD thesis, Zürich : ETH, 2010.

[8] Ken North. Benchmarks of SQL Query Performance for ODBC and Oracle Call Interface. Technical report, Ken North Computing LLC, October 2002.

[9] Roger E. Sanders. *ODBC 3.5 Developers Guide*. McGraw-Hill, 1999.

[10] Edward Sciore. *Database design and implementation*. John Wiley & Sons, Hoboken, NJ, 2009.

[11] R. Sidhu and V.K. Prasanna. Fast Regular Expression Matching Using FPGAs. In *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, pages 227–238, 29 2001-april 2 2001.

[12] Altera Unveils Innovations for 28-nm FPGAs, July 2012. `http://www.altera.com/corporate/news_room/releases/2010/products/nr-innovating-at-28-nm.html`.

[13] Transaction Processing Performance Council, TPC Benchmarks, September 2011. `http://www.tpc.org/information/benchmarks.asp`.

[14] Sourceforge, OpenPR project, July 2012. `http://openpr-vt.sourceforge.net/`.

[15] Sourceforge, Database Test Suite, MediaWiki main page, September 2011. `http://sourceforge.net/apps/mediawiki/osdldbt/`.

Appendix

Matlab Script for File System Overhead

```
clearvars;
clf;

blockSizeBytesStart = 384;
totalMemSizeBytes = 24 * 2^30;
bitSizeOfAllocationIndicator = 1;

i = 1;
blockSizeBytes = blockSizeBytesStart * 5;
while blockSizeBytes < 100 * 2^10

    sectorSize(i) = blockSizeBytes;
    sectorCount = totalMemSizeBytes / sectorSize(i);

    if sectorCount > 2^16
        wordSize = 4; %32 bit
    else
        wordSize = 2; %16 bit
    end

    overheadBytes(i) = sectorCount * wordSize * 2;
    slackSpace20000(i) = 20000 * sectorSize(i) + overheadBytes(i);
    slackSpace10000(i) = 10000 * sectorSize(i) + overheadBytes(i);
    slackSpace5000(i) = 5000 * sectorSize(i) + overheadBytes(i);
    slackSpace2500(i) = 2500 * sectorSize(i) + overheadBytes(i);
    slackSpace1000(i) = 1000 * sectorSize(i) + overheadBytes(i);
    slackSpace500(i) = 500 * sectorSize(i) + overheadBytes(i);

    % get all slack spaces for optimal solutions calculation
    for j = 1:70000
        slackSpace(j,i) = j * sectorSize(i) / 2 + overheadBytes(i);
    end

    i = i+1;
    blockSizeBytes = blockSizeBytes + blockSizeBytesStart;
end
i = i - 1;

smallest = min(slackSpace,[],2);

%find optimal solutions
for a = 1:i
    for b = 1: 70000
        if smallest(b) == slackSpace(b,a)
            optimalFront(a) = smallest(b);
        end
    end
end

optimalX = sectorSize(4:end);
optimalY = optimalFront(4:end);

semilogx(optimalX, optimalY,
  sectorSize, slackSpace20000, sectorSize, slackSpace10000,
  sectorSize, slackSpace5000, sectorSize, slackSpace2500,
  sectorSize, slackSpace1000, sectorSize, slackSpace500)

legend('Optimal solutions',
```

```
    'Meta data and slack with 20000 files',
    'Meta data and slack with 10000 files',
    'Meta data and slack with 5000 files',
    'Meta data and slack with 2500 files',
    'Meta data and slack with 1000 files',
    'Meta data and slack with 500 files')
```

VHDL Code for the Module Library

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

package module_pkg is

  -- constants

  -- init signal bit position constants for all modules
  constant INIT_DECREMENT_POS    : natural := 0; -- word 0
  constant INIT_DECREMENT_LEN    : natural := 8;
  constant INIT_CHUNK_POS        : natural := 32; -- word 1
  constant INIT_CHUNK_LEN        : natural := 5;
  constant INIT_LAST_CHUNK_POS   : natural := 64; -- word 2
  constant INIT_LAST_CHUNK_LEN   : natural := 5;
  constant INIT_LOGIC_OP0_POS    : natural := 96; -- word 3
  constant INIT_LOGIC_OP1_POS    : natural := 128; -- word 4
  constant INIT_LOGIC_OP2_POS    : natural := 160; -- word 5
  constant INIT_LOGIC_OP3_POS    : natural := 192; -- word 6
  constant INIT_LOGIC_OP4_POS    : natural := 224; -- word 7
  constant INIT_LOGIC_OP5_POS    : natural := 256; -- word 8
  constant INIT_LOGIC_OP6_POS    : natural := 288; -- word 9
  constant INIT_LOGIC_OP7_POS    : natural := 320; -- word 10
  constant INIT_LOGIC_OP_LEN     : natural := 3;


  -- init signal constants specific to pattern match module
  constant INIT_PM_BITMAP_POS        : natural := 352; -- word 11
  constant INIT_PM_BITMAP_LEN        : natural := 16;
  constant INIT_PM_IS_PHASE1_POS     : natural := 384;  -- word 12
  constant INIT_PM_IS_PHASE1_LEN     : natural := 1;


  -- init signal constants specific to compare module
  constant INIT_INTC_VAL_POS         : natural := 416; -- word 13
  constant INIT_INTC_VAL_LEN         : natural := 32;
  constant INIT_INTC_OPERATOR_POS    : natural := 448; -- word 14
  constant INIT_INTC_OPERATOR_LEN    : natural := 4;
  constant INIT_INTC_WORD_POS        : natural := 480;  -- word 15
  constant INIT_INTC_WORD_LEN        : natural := 4;

  constant EQUAL_TO                  : std_logic_vector(3 downto 0) := x"0";
  constant GREATER_THAN              : std_logic_vector(3 downto 0) := x"1";
  constant GREATER_THAN_OR_EQUAL_TO  : std_logic_vector(3 downto 0) := x"2";
  constant LESS_THAN                 : std_logic_vector(3 downto 0) := x"3";
  constant LESS_THAN_OR_EQUAL_TO     : std_logic_vector(3 downto 0) := x"4";

  -- logic gates
  constant LOGIC_NOOP    : std_logic_vector(2 downto 0) := "000";
  constant LOGIC_AND     : std_logic_vector(2 downto 0) := "001";
  constant LOGIC_NAND    : std_logic_vector(2 downto 0) := "010";
  constant LOGIC_OR      : std_logic_vector(2 downto 0) := "011";
  constant LOGIC_NOR     : std_logic_vector(2 downto 0) := "100";

  -- states
  constant CHUNK_0       : std_logic_vector(4 downto 0) := "00000"; -- first chunk
  constant CHUNK_1       : std_logic_vector(4 downto 0) := "00001";
  constant CHUNK_2       : std_logic_vector(4 downto 0) := "00010";
  constant CHUNK_3       : std_logic_vector(4 downto 0) := "00011";
  constant CHUNK_4       : std_logic_vector(4 downto 0) := "00100";
  constant CHUNK_5       : std_logic_vector(4 downto 0) := "00101";
  constant CHUNK_6       : std_logic_vector(4 downto 0) := "00110";
  constant CHUNK_7       : std_logic_vector(4 downto 0) := "00111";
  -- ... we dont name the rest of the chunks. names are only for
  --     debugging, except the last chunk
  constant CHUNK_LAST    : std_logic_vector(4 downto 0) := "11010"; -- last chunk, 26
  constant IDLE          : std_logic_vector(4 downto 0) := "11011"; -- 27
  constant INIT          : std_logic_vector(4 downto 0) := "11100"; -- 28
```

```
69    constant INIT_PHASE2   : std_logic_vector(4 downto 0) := "11101"; -- 29
70    constant DONE          : std_logic_vector(4 downto 0) := "11110"; -- 30
71    constant RESET         : std_logic_vector(4 downto 0) := "11111"; -- 31
72
73    -- function declarations
74    function get_word(n : natural; input : std_logic_vector(511 downto 0))
75      return std_logic_vector;
76    function set_word(n : natural ; word : std_logic_vector(31 downto 0)
77      ; input : std_logic_vector(511 downto 0)) return std_logic_vector;
78    function get_part(from : natural; len : natural
79      ; input : std_logic_vector(511 downto 0)) return std_logic_vector;
80    function set_part(from : natural; len : natural; part : std_logic_vector
81      ; input : std_logic_vector(511 downto 0)) return std_logic_vector;
82    function mux_word(n : std_logic_vector(3 downto 0)
83      ; input : std_logic_vector(511 downto 0)) return std_logic_vector;
84    procedure downshift_process
85    (
86      input_sig : in std_logic;
87      input_bus : in  std_logic_vector(7 downto 0);
88      output_bus : out std_logic_vector(7 downto 0)
89    );
90    procedure upshift_process
91    (
92      input_sig : in std_logic;
93      input_bus : in  std_logic_vector(7 downto 0);
94      output_bus : out std_logic_vector(7 downto 0)
95    );
96
97  end;
98
99  package body module_pkg is
100
101   procedure downshift_process
102   (
103     input_sig : in std_logic;
104     input_bus : in  std_logic_vector(7 downto 0);
105     output_bus : out std_logic_vector(7 downto 0)
106   )
107   is
108   begin
109     for i in 0 to 6 loop
110       output_bus(i + 1) := input_bus(i);
111     end loop;
112     output_bus(0) := input_sig;
113   end;
114
115   procedure upshift_process
116   (
117     input_sig : in std_logic;
118     input_bus : in  std_logic_vector(7 downto 0);
119     output_bus : out std_logic_vector(7 downto 0)
120   )
121   is
122   begin
123     output_bus(7) := '0';
124     for i in 2 to 7 loop
125       output_bus(i - 1) := input_bus(i);
126     end loop;
127     output_bus(0) := input_sig;
128   end;
129
130   -- functions
131
132   function get_word(n : natural; input : std_logic_vector(511 downto 0))
133     return std_logic_vector is
134   begin
135     return input(n*32 + 31 downto n*32);
136   end get_word;
137
138   function set_word(n : natural ; word : std_logic_vector(31 downto 0)
139   ; input : std_logic_vector(511 downto 0)) return std_logic_vector is
140     variable ret : std_logic_vector(511 downto 0);
141   begin
142     ret := input;
143     ret(n*32 + 31 downto n*32) := word;
144     return ret;
145   end set_word;
146
147   function get_part(from : natural; len : natural
148     ; input : std_logic_vector(511 downto 0)) return std_logic_vector is
```

```
149   begin
150     return input(from + len - 1 downto from);
151   end get_part;
152
153   function set_part(from : natural; len : natural; part : std_logic_vector
154   ; input : std_logic_vector(511 downto 0)) return std_logic_vector is
155     variable ret : std_logic_vector(511 downto 0);
156   begin
157     ret := input;
158     ret(from + len - 1 downto from) := part;
159     return ret;
160   end set_part;
161
162   function mux_word(n : std_logic_vector(3 downto 0)
163   ; input : std_logic_vector(511 downto 0)) return std_logic_vector is
164   begin
165     case n is
166       when x"0" => return input(31 downto 0);
167       when x"1" => return input(63 downto 32);
168       when x"2" => return input(95 downto 64);
169       when x"3" => return input(127 downto 96);
170       when x"4" => return input(159 downto 128);
171       when x"5" => return input(191 downto 160);
172       when x"6" => return input(223 downto 192);
173       when x"7" => return input(255 downto 224);
174       when x"8" => return input(287 downto 256);
175       when x"9" => return input(319 downto 288);
176       when x"A" => return input(351 downto 320);
177       when x"B" => return input(383 downto 352);
178       when x"C" => return input(415 downto 384);
179       when x"D" => return input(447 downto 416);
180       when x"E" => return input(479 downto 448);
181       when x"F" => return input(511 downto 480);
182       when others => return input(511 downto 480); -- never happens
183     end case;
184   end mux_word;
185
186 end module_pkg;
```

VHDL Code for the Integer Compare Module

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4  use work.module_pkg.all;
5
6  entity module_int_compare is
7    port
8      (
9        clk          : in std_logic;
10
11       in_data    : in std_logic_vector(511 downto 0);
12       in_state   : in std_logic_vector(4 downto 0); -- passed on unchanged
13       in_results  : in std_logic_vector(7 downto 0);
14
15       out_data   : out std_logic_vector(511 downto 0);
16       out_state  : out std_logic_vector(4 downto 0);
17       out_results : out std_logic_vector(7 downto 0)
18     );
19 end module_int_compare;
20
21 architecture behavioural of module_int_compare is
22
23   -- initialization related signals
24
25   -- value to compare input with
26   signal compare_val     : std_logic_vector(31 downto 0);
27   signal compare_operator : std_logic_vector(3 downto 0) := EQUAL_TO;
28   signal word_pos      : std_logic_vector(3 downto 0) := x"0";
29   signal match_chunk     : std_logic_vector(4 downto 0) := IDLE;
30   signal last_chunk      : std_logic_vector(4 downto 0);
31
32   type logic_operators_t is array (0 to 7) of std_logic_vector(2 downto 0);
33   signal logic_operators : logic_operators_t;
34
35   signal intermediate_1_result_delay : std_logic;
36   signal intermediate_1_is_last_chunk : std_logic;
```

```
37    signal intermediate_2_is_last_chunk : std_logic;
38    signal intermediate_1_results : std_logic_vector(7 downto 0);
39    signal intermediate_1_data  : std_logic_vector(511 downto 0);
40    signal intermediate_1_state   : std_logic_vector(4 downto 0);
41    signal intermediate_2_results : std_logic_vector(7 downto 0);
42    signal intermediate_2_data  : std_logic_vector(511 downto 0);
43    signal intermediate_2_state   : std_logic_vector(4 downto 0);
44
45    signal out_results_sig : std_logic_vector(7 downto 0);
46
47  begin
48
49    out_results <= out_results_sig;
50
51    -- This process passes the signals to the next module.
52    -- Except from out_results_sig, which is updated in match_p.
53    output_p : process(clk)
54      variable in_data_var : std_logic_vector(511 downto 0);
55      variable decrement : std_logic_vector(7 downto 0);
56    begin
57      if rising_edge(clk) then
58        in_data_var := in_data;
59
60        case in_state is
61          when INIT =>
62            -- decrement first byte
63            decrement :=
64              get_part(INIT_DECREMENT_POS, INIT_DECREMENT_LEN, in_data_var);
65            decrement :=
66              std_logic_vector(unsigned(decrement) - 1);
67            in_data_var :=
68              set_part(INIT_DECREMENT_POS, INIT_DECREMENT_LEN, decrement, in_data_var);
69          when others =>
70        end case;
71
72        intermediate_1_data <= in_data_var;
73        intermediate_1_state <= in_state;
74
75        intermediate_2_data <= intermediate_1_data;
76        intermediate_2_state <= intermediate_1_state;
77
78        out_data <= intermediate_2_data;
79        out_state <= intermediate_2_state;
80      end if; -- rising edge
81    end process;
82
83    -- This process is triggered when the input state is INIT and the "init decrement byte"
84    -- is 0. It sets the result_delay_var string, and which words should be active.
85    initialize_p : process(clk)
86    begin
87      if rising_edge(clk) then
88        if in_state = INIT
89        and unsigned(get_part(INIT_DECREMENT_POS, INIT_DECREMENT_LEN, in_data)) = 0 then
90          -- this call is for me!
91          match_chunk <= get_part(INIT_CHUNK_POS, INIT_CHUNK_LEN, in_data);
92          last_chunk <= get_part(INIT_LAST_CHUNK_POS, INIT_LAST_CHUNK_LEN, in_data);
93
94          -- set the bitmap telling which words should be active
95          word_pos <= get_part(INIT_INTC_WORD_POS, INIT_INTC_WORD_LEN, in_data);
96          compare_val <= get_part(INIT_INTC_VAL_POS, INIT_INTC_VAL_LEN, in_data);
97          compare_operator <=
98            get_part(INIT_INTC_OPERATOR_POS, INIT_INTC_OPERATOR_LEN, in_data);
99          logic_operators(0) <= get_part(INIT_LOGIC_OP0_POS, INIT_LOGIC_OP_LEN, in_data);
100         logic_operators(1) <= get_part(INIT_LOGIC_OP1_POS, INIT_LOGIC_OP_LEN, in_data);
101         logic_operators(2) <= get_part(INIT_LOGIC_OP2_POS, INIT_LOGIC_OP_LEN, in_data);
102         logic_operators(3) <= get_part(INIT_LOGIC_OP3_POS, INIT_LOGIC_OP_LEN, in_data);
103         logic_operators(4) <= get_part(INIT_LOGIC_OP4_POS, INIT_LOGIC_OP_LEN, in_data);
104         logic_operators(5) <= get_part(INIT_LOGIC_OP5_POS, INIT_LOGIC_OP_LEN, in_data);
105         logic_operators(6) <= get_part(INIT_LOGIC_OP6_POS, INIT_LOGIC_OP_LEN, in_data);
106         logic_operators(7) <= get_part(INIT_LOGIC_OP7_POS, INIT_LOGIC_OP_LEN, in_data);
107       elsif in_state = RESET then
108         match_chunk <= IDLE;
109         match_chunk <= IDLE;
110         logic_operators <= (others => (others => '0'));
111       end if;
112     end if; -- rising edge
113   end process;
114
115   match_stage_1_p : process(clk)
116     variable in_word : std_logic_vector(31 downto 0); -- bit map
```

```
117       variable result_delay_var : std_logic; -- true or false
118     begin
119       if rising_edge(clk) then
120         result_delay_var := intermediate_1_result_delay;
121
122         if in_state = last_chunk then
123           intermediate_1_is_last_chunk <= '1';
124         else
125           intermediate_1_is_last_chunk <= '0';
126         end if;
127
128         if in_state <= CHUNK_LAST then
129           if in_state = match_chunk then
130             in_word := mux_word(word_pos, in_data);
131             result_delay_var := '0';
132
133             case compare_operator is
134               when EQUAL_TO =>
135                 if in_word = compare_val then
136                   result_delay_var := '1';
137                 end if;
138               when GREATER_THAN =>
139                 if signed(in_word) > signed(compare_val) then
140                   result_delay_var := '1';
141                 end if;
142               when GREATER_THAN_OR_EQUAL_TO =>
143                 if signed(in_word) >= signed(compare_val) then
144                   result_delay_var := '1';
145                 end if;
146               when LESS_THAN =>
147                 if signed(in_word) < signed(compare_val) then
148                   result_delay_var := '1';
149                 end if;
150               when LESS_THAN_OR_EQUAL_TO =>
151                 if signed(in_word) <= signed(compare_val) then
152                   result_delay_var := '1';
153                 end if;
154               when others =>
155             end case;
156           end if;
157         end if;
158
159         intermediate_1_results <= in_results;
160         intermediate_1_result_delay <= result_delay_var;
161       end if; -- rising edge
162     end process;
163
164     match_stage_2_p : process(clk)
165       variable result_delay_var : std_logic; -- true or false
166       variable out_results_var : std_logic_vector(7 downto 0);
167       variable tmp : std_logic;
168     begin
169       if rising_edge(clk) then
170         result_delay_var := intermediate_1_result_delay;
171
172           if intermediate_1_is_last_chunk = '1' then
173             downshift_process(result_delay_var, intermediate_1_results, out_results_var);
174             for i in 0 to 2 loop
175               case logic_operators(i) is
176                 when LOGIC_AND =>
177                   tmp := out_results_var(0) and out_results_var(1);
178                   upshift_process(tmp, out_results_var, out_results_var);
179                 when LOGIC_NAND =>
180                   tmp := out_results_var(0) nand out_results_var(1);
181                   upshift_process(tmp, out_results_var, out_results_var);
182                 when LOGIC_OR =>
183                   tmp := out_results_var(0) or out_results_var(1);
184                   upshift_process(tmp, out_results_var, out_results_var);
185                 when LOGIC_NOR =>
186                   tmp := out_results_var(0) nor out_results_var(1);
187                   upshift_process(tmp, out_results_var, out_results_var);
188                 when others => null; -- NOOP
189               end case;
190             end loop;
191
192             intermediate_2_results <= out_results_var;
193           end if;
194
195         intermediate_2_is_last_chunk <= intermediate_1_is_last_chunk;
196       end if; -- rising edge
```

```
197   end process;
198
199   match_stage_3_p : process(clk)
200     variable out_results_var : std_logic_vector(7 downto 0);
201     variable tmp : std_logic;
202   begin
203     if rising_edge(clk) then
204       out_results_var := intermediate_2_results;
205
206       if intermediate_2_is_last_chunk = '1' then
207         for i in 2 to 7 loop
208           case logic_operators(i) is
209             when LOGIC_AND =>
210               tmp := out_results_var(0) and out_results_var(1);
211               upshift_process(tmp, out_results_var, out_results_var);
212             when LOGIC_NAND =>
213               tmp := out_results_var(0) nand out_results_var(1);
214               upshift_process(tmp, out_results_var, out_results_var);
215             when LOGIC_OR =>
216               tmp := out_results_var(0) or out_results_var(1);
217               upshift_process(tmp, out_results_var, out_results_var);
218             when LOGIC_NOR =>
219               tmp := out_results_var(0) nor out_results_var(1);
220               upshift_process(tmp, out_results_var, out_results_var);
221             when others => null; -- NOOP
222           end case;
223         end loop;
224         out_results_sig <= out_results_var;
225       end if;
226     end if; -- rising edge
227   end process;
228
229
230 end behavioural;
```

VHDL Code for the Simple Pattern Match Module

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  use IEEE.numeric_std.all;
4  use work.module_pkg.all;
5
6  entity module_pattern_match_simple is
7    port
8      (
9        clk          : in std_logic;
10
11       in_data    : in std_logic_vector(511 downto 0);
12       in_state   : in std_logic_vector(4 downto 0); -- passed on unchanged
13       in_results  : in std_logic_vector(7 downto 0);
14
15       out_data   : out std_logic_vector(511 downto 0);
16       out_state  : out std_logic_vector(4 downto 0);
17       out_results : out std_logic_vector(7 downto 0)
18     );
19 end module_pattern_match_simple;
20
21 architecture behavioural of module_pattern_match_simple is
22
23   type init_state_t is ( PHASE1, PHASE2 );
24
25   -- the maximum number of chunks string can span over
26   constant MAX_CHUNKS : natural := 29;
27
28
29   -- initialization related signals
30   signal init_state : init_state_t := PHASE1;
31   signal match_words : std_logic_vector(15 downto 0); -- bit map
32   signal match_string : std_logic_vector(511 downto 0);
33   signal match_chunk : std_logic_vector(4 downto 0) := IDLE;
34   signal last_chunk : std_logic_vector(4 downto 0);
35
36   type logic_operators_t is array (0 to 7) of std_logic_vector(2 downto 0);
37   signal logic_operators : logic_operators_t;
38
```

```
39    signal result_delay : std_logic;
40
41    signal intermediate_is_last_chunk : std_logic;
42    signal intermediate_results : std_logic_vector(7 downto 0);
43    signal intermediate_data  : std_logic_vector(511 downto 0);
44    signal intermediate_state   : std_logic_vector(4 downto 0);
45
46    signal in_data_reg : std_logic_vector(511 downto 0);
47    signal in_state_reg : std_logic_vector(4 downto 0);
48    signal in_results_reg : std_logic_vector(7 downto 0);
49
50  begin
51
52    input_FFs_p : process(clk)
53    begin
54      if rising_edge(clk) then
55        in_data_reg <= in_data;
56        in_state_reg <= in_state;
57        in_results_reg <= in_results;
58      end if; -- rising edge
59    end process;
60
61    -- This process passes the signals to the next module.
62    -- Except from out_results, which is updated in match_p.
63    output_p : process(clk)
64      variable in_data_var : std_logic_vector(511 downto 0);
65      variable decrement : std_logic_vector(7 downto 0);
66    begin
67      if rising_edge(clk) then
68        in_data_var := in_data_reg;
69
70        case in_state_reg is
71            when INIT =>
72              -- decrement first byte
73              decrement :=
74                get_part(INIT_DECREMENT_POS, INIT_DECREMENT_LEN, in_data_var);
75              decrement :=
76                std_logic_vector(unsigned(decrement) - 1);
77              in_data_var :=
78                set_part(INIT_DECREMENT_POS, INIT_DECREMENT_LEN, decrement, in_data_var);
79            when others =>
80          end case;
81
82          intermediate_data <= in_data_var;
83          intermediate_state <= in_state_reg;
84
85          out_data <= intermediate_data;
86          out_state <= intermediate_state;
87      end if; -- rising edge
88    end process;
89
90    -- This process is triggered when the input state is INIT and the "init decrement byte"
91    -- is 0. It sets the match string, and which words should be active.
92    initialize_p : process(clk)
93    begin
94      if rising_edge(clk) then
95        if in_state_reg = INIT
96        and unsigned(get_part(INIT_DECREMENT_POS, INIT_DECREMENT_LEN, in_data_reg)) = 0
97        and init_state = PHASE1 then
98          -- this call is for me!
99          -- set the bitmap telling which words should be active
100         match_chunk <= get_part(INIT_CHUNK_POS, INIT_CHUNK_LEN, in_data_reg);
101         last_chunk <= get_part(INIT_LAST_CHUNK_POS, INIT_LAST_CHUNK_LEN, in_data_reg);
102         match_words <= get_part(INIT_PM_BITMAP_POS, INIT_PM_BITMAP_LEN, in_data_reg);
103         logic_operators(0) <= get_part(INIT_LOGIC_OP0_POS, INIT_LOGIC_OP_LEN, in_data_reg);
104         logic_operators(1) <= get_part(INIT_LOGIC_OP1_POS, INIT_LOGIC_OP_LEN, in_data_reg);
105         logic_operators(2) <= get_part(INIT_LOGIC_OP2_POS, INIT_LOGIC_OP_LEN, in_data_reg);
106         logic_operators(3) <= get_part(INIT_LOGIC_OP3_POS, INIT_LOGIC_OP_LEN, in_data_reg);
107         logic_operators(4) <= get_part(INIT_LOGIC_OP4_POS, INIT_LOGIC_OP_LEN, in_data_reg);
108         logic_operators(5) <= get_part(INIT_LOGIC_OP5_POS, INIT_LOGIC_OP_LEN, in_data_reg);
109         logic_operators(6) <= get_part(INIT_LOGIC_OP6_POS, INIT_LOGIC_OP_LEN, in_data_reg);
110         logic_operators(7) <= get_part(INIT_LOGIC_OP7_POS, INIT_LOGIC_OP_LEN, in_data_reg);
111         init_state <= PHASE2;
112       elsif in_state_reg = INIT_PHASE2 and init_state = PHASE2 then
113           -- set the match string
114           match_string <= in_data_reg;
115           init_state <= PHASE1;
116       elsif in_state_reg = RESET then
117         match_chunk <= IDLE;
118         last_chunk <= IDLE;
```

```
119         logic_operators <= (others => (others => '0'));
120       end if;
121     end if; -- rising edge
122   end process;
123
124   match_stage_1_p : process(clk)
125     variable result_delay_var : std_logic;
126     variable out_results_var : std_logic_vector(7 downto 0);
127     variable tmp_results_var : std_logic_vector(7 downto 0);
128     variable tmp : std_logic;
129   begin
130     if rising_edge(clk) then
131       result_delay_var := result_delay;
132       intermediate_is_last_chunk <= '0';
133
134       if in_state_reg = CHUNK_0 then
135         result_delay_var := '1'; -- new record, reset this signal
136       end if;
137
138       if in_state_reg <= CHUNK_LAST then
139         if in_state_reg = match_chunk then
140           for i in 0 to 15 loop
141             if match_words(i) = '1'
142             and get_word(i, in_data_reg) /= get_word(i, match_string) then
143               result_delay_var := '0'; -- mismatch
144             end if;
145           end loop;
146         end if;
147
148         if in_state_reg = last_chunk then
149           downshift_process(result_delay_var, in_results_reg, out_results_var);
150
151           for i in 0 to 2 loop
152             case logic_operators(i) is
153               when LOGIC_AND =>
154                 tmp := out_results_var(0) and out_results_var(1);
155                 upshift_process(tmp, out_results_var, out_results_var);
156               when LOGIC_NAND =>
157                 tmp := out_results_var(0) nand out_results_var(1);
158                 upshift_process(tmp, out_results_var, out_results_var);
159               when LOGIC_OR =>
160                 tmp := out_results_var(0) or out_results_var(1);
161                 upshift_process(tmp, out_results_var, out_results_var);
162               when LOGIC_NOR =>
163                 tmp := out_results_var(0) nor out_results_var(1);
164                 upshift_process(tmp, out_results_var, out_results_var);
165               when others => null; -- NOOP
166             end case;
167           end loop;
168
169           intermediate_is_last_chunk <= '1';
170           intermediate_results <= out_results_var;
171         end if;
172       end if;
173
174       result_delay <= result_delay_var; -- delay to sync with last chunk
175     end if; -- rising edge
176   end process;
177
178   match_stage_2_p : process(clk)
179     variable result_delay_var : std_logic;
180     variable out_results_var : std_logic_vector(7 downto 0);
181     variable tmp : std_logic;
182   begin
183     if rising_edge(clk) then
184       result_delay_var := result_delay;
185       out_results_var := intermediate_results;
186
187       if intermediate_is_last_chunk = '1' then
188         for i in 3 to 7 loop
189           case logic_operators(i) is
190             when LOGIC_AND =>
191               tmp := out_results_var(0) and out_results_var(1);
192               upshift_process(tmp, out_results_var, out_results_var);
193             when LOGIC_NAND =>
194               tmp := out_results_var(0) nand out_results_var(1);
195               upshift_process(tmp, out_results_var, out_results_var);
196             when LOGIC_OR =>
197               tmp := out_results_var(0) or out_results_var(1);
198               upshift_process(tmp, out_results_var, out_results_var);
```

```
199               when LOGIC_NOR =>
200                 tmp := out_results_var(0) nor out_results_var(1);
201                 upshift_process(tmp, out_results_var, out_results_var);
202               when others => null; -- NOOP
203             end case;
204           end loop;
205           out_results <= out_results_var;
206         end if;
207       end if; -- rising edge
208     end process;
209
210
211   end behavioural;
```

VHDL Code for the Long String Pattern Match Module

```
1   library IEEE;
2   use IEEE.std_logic_1164.all;
3   use IEEE.numeric_std.all;
4   use work.module_pkg.all;
5
6   entity module_pattern_match_srl is
7     port
8       (
9         clk          : in std_logic;
10
11        in_data    : in std_logic_vector(511 downto 0);
12        in_state   : in std_logic_vector(4 downto 0); -- passed on unchanged
13        in_results   : in std_logic_vector(7 downto 0);
14
15        out_data   : out std_logic_vector(511 downto 0);
16        out_state  : out std_logic_vector(4 downto 0);
17        out_results : out std_logic_vector(7 downto 0)
18      );
19  end module_pattern_match_srl;
20
21  architecture behavioural of module_pattern_match_srl is
22
23    -- the maximum number of chunks string can span over
24    constant MAX_CHUNKS : natural := 27;
25
26    type match_record_t is
27      record
28        match_words : std_logic_vector(15 downto 0);
29        match_string: std_logic_vector(511 downto 0);
30      end record;
31
32    --first define the type of array.
33    type match_records_t is array (0 to MAX_CHUNKS - 1) of match_record_t;
34    type init_state_t is ( PHASE1, PHASE2 );
35
36    -- initialization related signals
37    signal init_state : init_state_t := PHASE1;
38    signal match_records : match_records_t;
39    signal first_match_chunk : std_logic_vector(4 downto 0) := IDLE;
40    signal last_match_chunk : std_logic_vector(4 downto 0);
41    signal last_chunk : std_logic_vector(4 downto 0);
42    signal match_chunk_count : integer range 0 to MAX_CHUNKS := 0;
43
44    signal tmp_result : std_logic;
45    signal chunk_counter : integer range 0 to MAX_CHUNKS := 0;
46
47    type logic_operators_t is array (0 to 7) of std_logic_vector(2 downto 0);
48    signal logic_operators : logic_operators_t;
49
50    signal intermediate_1_data  : std_logic_vector(511 downto 0);
51    signal intermediate_1_state : std_logic_vector(4 downto 0);
52    signal intermediate_1_results : std_logic_vector(7 downto 0);
53
54    signal intermediate_2_data  : std_logic_vector(511 downto 0);
55    signal intermediate_2_state : std_logic_vector(4 downto 0);
56    signal intermediate_2_results : std_logic_vector(7 downto 0);
57
58    signal intermediate_1_is_data_chunk : std_logic;
59    signal intermediate_1_is_last_chunk : std_logic;
```

```
60    signal intermediate_1_is_first_match_chunks : std_logic;
61
62    signal intermediate_2_is_last_chunk : std_logic;
63
64    signal intermediate_1_match_words : std_logic_vector(15 downto 0);
65    signal intermediate_1_match_records : std_logic_vector(15 downto 0);
66
67    signal in_data_reg : std_logic_vector(511 downto 0);
68    signal in_state_reg : std_logic_vector(4 downto 0);
69    signal in_results_reg : std_logic_vector(7 downto 0);
70
71  begin
72
73    input_FFs_p : process(clk)
74    begin
75      if rising_edge(clk) then
76        in_data_reg <= in_data;
77        in_state_reg <= in_state;
78        in_results_reg <= in_results;
79      end if; -- rising edge
80    end process;
81
82    -- This process passes the signals to the next module.
83    -- Except from out_results, which is updated in match_p.
84    output_p : process(clk)
85      variable in_data_var : std_logic_vector(511 downto 0);
86      variable decrement : std_logic_vector(7 downto 0);
87    begin
88      if rising_edge(clk) then
89        in_data_var := in_data_reg;
90
91        case in_state_reg is
92            when INIT =>
93              -- decrement first byte
94              decrement :=
95                get_part(INIT_DECREMENT_POS, INIT_DECREMENT_LEN, in_data_var);
96              decrement :=
97                std_logic_vector(unsigned(decrement) - 1);
98              in_data_var :=
99                set_part(INIT_DECREMENT_POS, INIT_DECREMENT_LEN, decrement, in_data_var);
100           when others =>
101         end case;
102
103         intermediate_1_state <= in_state_reg;
104         intermediate_1_data <= in_data_var;
105
106         intermediate_2_state <= intermediate_1_state;
107         intermediate_2_data <= intermediate_1_data;
108
109         out_state <= intermediate_2_state;
110         out_data <= intermediate_2_data;
111
112
113     end if; -- rising edge
114   end process;
115
116   -- This process is triggered when the input state is INIT and the "init decrement byte"
117   -- is 0. It sets the match string, and which words should be active.
118   initialize_p : process(clk)
119     variable match_chunk : std_logic_vector(4 downto 0);
120   begin
121     if rising_edge(clk) then
122       if in_state_reg = INIT
123       and unsigned(get_part(INIT_DECREMENT_POS, INIT_DECREMENT_LEN, in_data_reg)) = 0
124       and init_state = PHASE1 then
125         -- this call is for me!
126
127         -- save first and last chunk
128         match_chunk := get_part(INIT_CHUNK_POS, INIT_CHUNK_LEN, in_data_reg);
129         last_chunk <= get_part(INIT_LAST_CHUNK_POS, INIT_LAST_CHUNK_LEN, in_data_reg);
130         if first_match_chunk = IDLE then
131           first_match_chunk <= match_chunk;
132         end if;
133         last_match_chunk <= match_chunk;
134         logic_operators(0) <= get_part(INIT_LOGIC_OP0_POS, INIT_LOGIC_OP_LEN, in_data_reg);
135         logic_operators(1) <= get_part(INIT_LOGIC_OP1_POS, INIT_LOGIC_OP_LEN, in_data_reg);
136         logic_operators(2) <= get_part(INIT_LOGIC_OP2_POS, INIT_LOGIC_OP_LEN, in_data_reg);
137         logic_operators(3) <= get_part(INIT_LOGIC_OP3_POS, INIT_LOGIC_OP_LEN, in_data_reg);
138         logic_operators(4) <= get_part(INIT_LOGIC_OP4_POS, INIT_LOGIC_OP_LEN, in_data_reg);
139         logic_operators(5) <= get_part(INIT_LOGIC_OP5_POS, INIT_LOGIC_OP_LEN, in_data_reg);
```

```
140         logic_operators(6) <= get_part(INIT_LOGIC_OP6_POS, INIT_LOGIC_OP_LEN, in_data_reg);
141         logic_operators(7) <= get_part(INIT_LOGIC_OP7_POS, INIT_LOGIC_OP_LEN, in_data_reg);
142
143         -- set the match words for this chunk
144         match_records(match_chunk_count).match_words <=
145           get_part(INIT_PM_BITMAP_POS, INIT_PM_BITMAP_LEN, in_data_reg);
146         init_state <= PHASE2;
147       elsif in_state_reg = INIT_PHASE2 and init_state = PHASE2 then
148           -- set the match string for this chunk
149           match_records(match_chunk_count).match_string <= in_data_reg;
150           match_chunk_count <= match_chunk_count + 1;
151           init_state <= PHASE1;
152       elsif in_state_reg = DONE then
153         -- reset intermediates
154         match_chunk_count <= 0;
155       elsif in_state_reg = RESET then
156         match_chunk_count <= 0;
157         first_match_chunk <= IDLE;
158         last_match_chunk <= IDLE;
159         logic_operators <= (others => (others => '0'));
160       end if;
161     end if; -- rising edge
162   end process;
163
164   match_stage_1_p : process(clk)
165     variable tmp : std_logic;
166   begin
167     if rising_edge(clk) then
168       intermediate_1_is_data_chunk <= '0';
169       intermediate_1_is_last_chunk <= '0';
170       intermediate_1_is_first_match_chunks <= '0';
171
172       if in_state_reg = first_match_chunk then -- if this is the first chunk
173         intermediate_1_is_first_match_chunks <= '1';
174       end if;
175
176
177       for i in 0 to 15 loop
178         if match_records(chunk_counter).match_words(i) = '1' then
179           intermediate_1_match_words(i) <= '1';
180         else
181           intermediate_1_match_words(i) <= '0';
182         end if;
183       end loop;
184
185       for i in 0 to 15 loop
186         if get_word(i, in_data_reg) =
187         get_word(i, match_records(chunk_counter).match_string) then
188           intermediate_1_match_records(i) <= '1';
189         else
190           intermediate_1_match_records(i) <= '0';
191         end if;
192       end loop;
193
194       if in_state_reg <= CHUNK_LAST then
195         if in_state_reg >= first_match_chunk and in_state_reg <= last_match_chunk then
196           intermediate_1_is_data_chunk <= '1';
197           chunk_counter <= chunk_counter + 1;
198         end if;
199       end if;
200
201       if in_state_reg = last_chunk then
202         chunk_counter <= 0;
203         intermediate_1_is_last_chunk <= '1';
204       end if;
205
206       intermediate_1_results <= in_results_reg;
207     end if; -- rising edge
208   end process;
209
210   match_stage_2_p : process(clk)
211     variable tmp_result_var : std_logic;
212     variable out_results_var : std_logic_vector(7 downto 0);
213     variable tmp : std_logic;
214   begin
215     if rising_edge(clk) then
216       tmp_result_var := tmp_result;
217
218       if intermediate_1_is_first_match_chunks = '1' then
219         tmp_result_var := '1'; -- reset result
```

```
220         end if;
221
222         if intermediate_1_is_data_chunk = '1' then
223           for i in 0 to 15 loop
224             if intermediate_1_match_words(i) = '1'
225             and intermediate_1_match_records(i) = '0' then
226               tmp_result_var := '0'; -- mismatch
227             end if;
228           end loop;
229         end if;
230
231         if intermediate_1_is_last_chunk = '1' then
232           downshift_process(tmp_result_var, intermediate_1_results, out_results_var);
233           for i in 0 to 2 loop
234             case logic_operators(i) is
235               when LOGIC_AND =>
236                 tmp := out_results_var(0) and out_results_var(1);
237                 upshift_process(tmp, out_results_var, out_results_var);
238               when LOGIC_NAND =>
239                 tmp := out_results_var(0) nand out_results_var(1);
240                 upshift_process(tmp, out_results_var, out_results_var);
241               when LOGIC_OR =>
242                 tmp := out_results_var(0) or out_results_var(1);
243                 upshift_process(tmp, out_results_var, out_results_var);
244               when LOGIC_NOR =>
245                 tmp := out_results_var(0) nor out_results_var(1);
246                 upshift_process(tmp, out_results_var, out_results_var);
247               when others => null; -- NOOP
248             end case;
249           end loop;
250         end if;
251
252         tmp_result <= tmp_result_var;
253         intermediate_2_is_last_chunk <= intermediate_1_is_last_chunk;
254         intermediate_2_results <= out_results_var;
255       end if; -- rising edge
256     end process;
257
258     match_stage_3_p : process(clk)
259       variable out_results_var : std_logic_vector(7 downto 0);
260       variable tmp : std_logic;
261     begin
262       if rising_edge(clk) then
263         out_results_var := intermediate_2_results;
264
265         if intermediate_2_is_last_chunk = '1' then
266           for i in 3 to 7 loop
267             case logic_operators(i) is
268               when LOGIC_AND =>
269                 tmp := out_results_var(0) and out_results_var(1);
270                 upshift_process(tmp, out_results_var, out_results_var);
271               when LOGIC_NAND =>
272                 tmp := out_results_var(0) nand out_results_var(1);
273                 upshift_process(tmp, out_results_var, out_results_var);
274               when LOGIC_OR =>
275                 tmp := out_results_var(0) or out_results_var(1);
276                 upshift_process(tmp, out_results_var, out_results_var);
277               when LOGIC_NOR =>
278                 tmp := out_results_var(0) nor out_results_var(1);
279                 upshift_process(tmp, out_results_var, out_results_var);
280               when others => null; -- NOOP
281             end case;
282           end loop;
283         end if;
284
285         out_results <= out_results_var;
286       end if; -- rising edge
287     end process;
288
289 end behavioural;
```

GoAhead Script for Producing the Static System

```
1 LoadFPGA FileName=E:\GoAhead\xc6vsx475t.binFPGA;
2
3 ClearSelection;
4 AddToSelectionXY UpperLeftX=200 UpperLeftY=1 LowerRightX=315 LowerRightY=83;
```

```
5  ExpandSelection;
6  StoreCurrentSelectionAs UserSelectionType=PartialArea;
7
8  AddMacro MacroName=RBB_blocker;
9  SelectMacro MacroName=RBB_blocker;
10
11 AddLibraryElementFromXDL XDLMacro=E:\GoAhead\BM\Connect4_V6_double.xdl
12   ParseXDLPortStatements=True;
13
14 # Left macros
15 ClearSelection;
16 AddToSelectionXY UpperLeftX=200 UpperLeftY=80 LowerRightX=203 LowerRightY=83;
17 AddToSelectionXY UpperLeftX=200 UpperLeftY=75 LowerRightX=203 LowerRightY=78;
18 AddToSelectionXY UpperLeftX=200 UpperLeftY=70 LowerRightX=203 LowerRightY=73;
19 AddToSelectionXY UpperLeftX=200 UpperLeftY=65 LowerRightX=203 LowerRightY=68;
20 AddToSelectionXY UpperLeftX=200 UpperLeftY=59 LowerRightX=203 LowerRightY=62;
21 AddToSelectionXY UpperLeftX=200 UpperLeftY=54 LowerRightX=203 LowerRightY=57;
22 AddToSelectionXY UpperLeftX=200 UpperLeftY=49 LowerRightX=203 LowerRightY=52;
23 AddToSelectionXY UpperLeftX=200 UpperLeftY=44 LowerRightX=203 LowerRightY=47;
24 AddToSelectionXY UpperLeftX=200 UpperLeftY=38 LowerRightX=203 LowerRightY=41;
25 AddToSelectionXY UpperLeftX=200 UpperLeftY=33 LowerRightX=203 LowerRightY=36;
26 AddToSelectionXY UpperLeftX=200 UpperLeftY=28 LowerRightX=203 LowerRightY=31;
27 AddToSelectionXY UpperLeftX=200 UpperLeftY=23 LowerRightX=203 LowerRightY=26;
28 AddToSelectionXY UpperLeftX=200 UpperLeftY=17 LowerRightX=203 LowerRightY=20;
29 AddToSelectionXY UpperLeftX=200 UpperLeftY=12 LowerRightX=203 LowerRightY=15;
30 AddToSelectionXY UpperLeftX=200 UpperLeftY=7 LowerRightX=203 LowerRightY=10;
31 AddToSelectionXY UpperLeftX=200 UpperLeftY=2 LowerRightX=203 LowerRightY=5;
32 ExpandSelection;
33 AddMacroInstantiationInSelectedTiles SliceNumber=1
34   InstanceName=left_data MacroName=Connect4_V6_double SortyBy=row SortOrder=asc;
35 StoreCurrentSelectionAs UserSelectionType=LeftData;
36
37
38
39 ClearSelection;
40 AddToSelectionXY UpperLeftX=200 UpperLeftY=43 LowerRightX=203 LowerRightY=43;
41 ExpandSelection;
42 AddMacroInstantiationInSelectedTiles SliceNumber=1
43   InstanceName=left_results MacroName=Connect4_V6_double SortyBy=row SortOrder=asc;
44 StoreCurrentSelectionAs UserSelectionType=LeftResults;
45
46 ClearSelection;
47 AddToSelectionXY UpperLeftX=200 UpperLeftY=37 LowerRightX=203 LowerRightY=37;
48 ExpandSelection;
49 AddMacroInstantiationInSelectedTiles SliceNumber=1
50   InstanceName=left_state MacroName=Connect4_V6_double SortyBy=row SortOrder=asc;
51 StoreCurrentSelectionAs UserSelectionType=LeftState;
52
53 # Middle macros
54 ClearSelection;
55 AddToSelectionXY UpperLeftX=254 UpperLeftY=80 LowerRightX=257 LowerRightY=83;
56 AddToSelectionXY UpperLeftX=254 UpperLeftY=75 LowerRightX=257 LowerRightY=78;
57 AddToSelectionXY UpperLeftX=254 UpperLeftY=70 LowerRightX=257 LowerRightY=73;
58 AddToSelectionXY UpperLeftX=254 UpperLeftY=65 LowerRightX=257 LowerRightY=68;
59 AddToSelectionXY UpperLeftX=254 UpperLeftY=59 LowerRightX=257 LowerRightY=62;
60 AddToSelectionXY UpperLeftX=254 UpperLeftY=54 LowerRightX=257 LowerRightY=57;
61 AddToSelectionXY UpperLeftX=254 UpperLeftY=49 LowerRightX=257 LowerRightY=52;
62 AddToSelectionXY UpperLeftX=254 UpperLeftY=44 LowerRightX=257 LowerRightY=47;
63 AddToSelectionXY UpperLeftX=254 UpperLeftY=38 LowerRightX=257 LowerRightY=41;
64 AddToSelectionXY UpperLeftX=254 UpperLeftY=33 LowerRightX=257 LowerRightY=36;
65 AddToSelectionXY UpperLeftX=254 UpperLeftY=28 LowerRightX=257 LowerRightY=31;
66 AddToSelectionXY UpperLeftX=254 UpperLeftY=23 LowerRightX=257 LowerRightY=26;
67 AddToSelectionXY UpperLeftX=254 UpperLeftY=17 LowerRightX=257 LowerRightY=20;
68 AddToSelectionXY UpperLeftX=254 UpperLeftY=12 LowerRightX=257 LowerRightY=15;
69 AddToSelectionXY UpperLeftX=254 UpperLeftY=7 LowerRightX=257 LowerRightY=10;
70 AddToSelectionXY UpperLeftX=254 UpperLeftY=2 LowerRightX=257 LowerRightY=5;
71 ExpandSelection;
72 AddMacroInstantiationInSelectedTiles SliceNumber=1
73   InstanceName=middle_data MacroName=Connect4_V6_double SortyBy=row SortOrder=asc;
74 StoreCurrentSelectionAs UserSelectionType=MiddleData;
75
76 ClearSelection;
77 AddToSelectionXY UpperLeftX=254 UpperLeftY=43 LowerRightX=257 LowerRightY=43;
78 ExpandSelection;
79 AddMacroInstantiationInSelectedTiles SliceNumber=1
80   InstanceName=middle_results MacroName=Connect4_V6_double SortyBy=row SortOrder=asc;
81 StoreCurrentSelectionAs UserSelectionType=MiddleResults;
82
83 ClearSelection;
84 AddToSelectionXY UpperLeftX=254 UpperLeftY=37 LowerRightX=257 LowerRightY=37;
```

```
85  ExpandSelection;
86  AddMacroInstantiationInSelectedTiles SliceNumber=1
87    InstanceName=middle_state MacroName=Connect4_V6_double SortyBy=row SortOrder=asc;
88  StoreCurrentSelectionAs UserSelectionType=MiddleState;
89
90  # Right macros
91  ClearSelection;
92  AddToSelectionXY UpperLeftX=312 UpperLeftY=80 LowerRightX=315 LowerRightY=83;
93  AddToSelectionXY UpperLeftX=312 UpperLeftY=75 LowerRightX=315 LowerRightY=78;
94  AddToSelectionXY UpperLeftX=312 UpperLeftY=70 LowerRightX=315 LowerRightY=73;
95  AddToSelectionXY UpperLeftX=312 UpperLeftY=65 LowerRightX=315 LowerRightY=68;
96  AddToSelectionXY UpperLeftX=312 UpperLeftY=59 LowerRightX=315 LowerRightY=62;
97  AddToSelectionXY UpperLeftX=312 UpperLeftY=54 LowerRightX=315 LowerRightY=57;
98  AddToSelectionXY UpperLeftX=312 UpperLeftY=49 LowerRightX=315 LowerRightY=52;
99  AddToSelectionXY UpperLeftX=312 UpperLeftY=44 LowerRightX=315 LowerRightY=47;
100 AddToSelectionXY UpperLeftX=312 UpperLeftY=38 LowerRightX=315 LowerRightY=41;
101 AddToSelectionXY UpperLeftX=312 UpperLeftY=33 LowerRightX=315 LowerRightY=36;
102 AddToSelectionXY UpperLeftX=312 UpperLeftY=28 LowerRightX=315 LowerRightY=31;
103 AddToSelectionXY UpperLeftX=312 UpperLeftY=23 LowerRightX=315 LowerRightY=26;
104 AddToSelectionXY UpperLeftX=312 UpperLeftY=17 LowerRightX=315 LowerRightY=20;
105 AddToSelectionXY UpperLeftX=312 UpperLeftY=12 LowerRightX=315 LowerRightY=15;
106 AddToSelectionXY UpperLeftX=312 UpperLeftY=7 LowerRightX=315 LowerRightY=10;
107 AddToSelectionXY UpperLeftX=312 UpperLeftY=2 LowerRightX=315 LowerRightY=5;
108 ExpandSelection;
109 AddMacroInstantiationInSelectedTiles SliceNumber=1
110   InstanceName=right_data MacroName=Connect4_V6_double SortyBy=row SortOrder=asc;
111 StoreCurrentSelectionAs UserSelectionType=RightData;
112
113 ClearSelection;
114 AddToSelectionXY UpperLeftX=312 UpperLeftY=43 LowerRightX=315 LowerRightY=43;
115 ExpandSelection;
116 AddMacroInstantiationInSelectedTiles SliceNumber=1
117   InstanceName=right_results MacroName=Connect4_V6_double SortyBy=row SortOrder=asc;
118 StoreCurrentSelectionAs UserSelectionType=RightResults;
119
120 ClearSelection;
121 AddToSelectionXY UpperLeftX=312 UpperLeftY=37 LowerRightX=315 LowerRightY=37;
122 ExpandSelection;
123 AddMacroInstantiationInSelectedTiles SliceNumber=1
124   InstanceName=right_state MacroName=Connect4_V6_double SortyBy=row SortOrder=asc;
125 StoreCurrentSelectionAs UserSelectionType=RightState;
126
127 ClearSelection;
128 AddToSelectionXY UpperLeftX=200 UpperLeftY=2 LowerRightX=315 LowerRightY=5;
129 AddToSelectionXY UpperLeftX=200 UpperLeftY=7 LowerRightX=315 LowerRightY=10;
130 AddToSelectionXY UpperLeftX=200 UpperLeftY=12 LowerRightX=315 LowerRightY=15;
131 AddToSelectionXY UpperLeftX=200 UpperLeftY=17 LowerRightX=315 LowerRightY=20;
132 AddToSelectionXY UpperLeftX=200 UpperLeftY=23 LowerRightX=315 LowerRightY=26;
133 AddToSelectionXY UpperLeftX=200 UpperLeftY=28 LowerRightX=315 LowerRightY=31;
134 AddToSelectionXY UpperLeftX=200 UpperLeftY=33 LowerRightX=315 LowerRightY=41;
135 AddToSelectionXY UpperLeftX=200 UpperLeftY=43 LowerRightX=315 LowerRightY=47;
136 AddToSelectionXY UpperLeftX=200 UpperLeftY=49 LowerRightX=315 LowerRightY=52;
137 AddToSelectionXY UpperLeftX=200 UpperLeftY=54 LowerRightX=315 LowerRightY=57;
138 AddToSelectionXY UpperLeftX=200 UpperLeftY=59 LowerRightX=315 LowerRightY=62;
139 AddToSelectionXY UpperLeftX=200 UpperLeftY=65 LowerRightX=315 LowerRightY=68;
140 AddToSelectionXY UpperLeftX=200 UpperLeftY=70 LowerRightX=315 LowerRightY=73;
141 AddToSelectionXY UpperLeftX=200 UpperLeftY=75 LowerRightX=315 LowerRightY=78;
142 AddToSelectionXY UpperLeftX=200 UpperLeftY=80 LowerRightX=315 LowerRightY=83;
143 ExpandSelection;
144 StoreCurrentSelectionAs UserSelectionType=blockWithoutEE2andWW2;
145
146 SelectUserSelection UserSelectionType=PartialArea Keep=False;
147 PrintProhibitStatementsForSelection =True
148   FileName=X:\blocker_test\prohibit.ucf Append=False;
149
150 SelectUserSelection UserSelectionType=blockWithoutEE2andWW2 Keep=False;
151
152 MarkPortsInSelectionAsUsedByRegexp PortNameRegexp=WW2
153   CheckForExistence=False IncludeAllPorts=False;
154 MarkPortsInSelectionAsUsedByRegexp PortNameRegexp=EE2
155   CheckForExistence=False IncludeAllPorts=False;
156
157
158 SelectUserSelection UserSelectionType=PartialArea Keep=False;
159 BlockSelection PrintUnblockedPorts=False
160   Prefix=RBB_blocker BlockWithEndPips=True SliceNumber=0;
161 GenerateXDL FileName=X:\blocker_test\blocker.xdl MacroNames=RBB_blocker
162   IncludePorts=False IncludeDummyNets=True IncludeDesignStatement=False
163   IncludeModuleHeader=False IncludeModuleFooter=False DesignName=_XILINX_NMC_MACRO;
164 GenerateXDL FileName=X:\blocker_test\blocker_FE.xdl MacroNames=RBB_blocker
```

```
  IncludePorts=False IncludeDummyNets=True IncludeDesignStatement=True
  IncludeModuleHeader=True IncludeModuleFooter=True DesignName=_XILINX_NMC_MACRO;

PrintLocationConstraintsForPlacedMacros Prefix=
  InstancePrefix= FileName=X:\blocker_test\prohibit.ucf Append=True;

# add east end FFs to area group, or map will route badly
# change GROUP and PLACE in UCF file to OPEN manually
ClearSelection;
AddToSelectionXY UpperLeftX=319 UpperLeftY=1 LowerRightX=333 LowerRightY=83;
ExpandSelection;
PrintAreaConstraint InstanceName=east_end_FFs_inst
  FileName=X:\blocker_test\prohibit.ucf Append=True;

# and area group for output fifo
ClearSelection;
AddToSelectionXY UpperLeftX=132 UpperLeftY=1 LowerRightX=199 LowerRightY=83;
ExpandSelection;
PrintAreaConstraint InstanceName=fifo_w512_d128_inst
  FileName=X:\blocker_test\prohibit.ucf Append=True;

ShowGUI;
```

GoAhead Script for Producing the Partial Modules

```
LoadFPGA FileName=E:\GoAhead\xc6vlx75tff484-3.binFPGA;

ClearSelection;
AddToSelectionXY UpperLeftX=95 UpperLeftY=0 LowerRightX=165 LowerRightY=0; #north
AddToSelectionXY UpperLeftX=95 UpperLeftY=84 LowerRightX=165 LowerRightY=96; #south
AddToSelectionXY UpperLeftX=95 UpperLeftY=1 LowerRightX=133 LowerRightY=83; #west
AddToSelectionXY UpperLeftX=149 UpperLeftY=1 LowerRightX=165 LowerRightY=83; #east
ExpandSelection;
StoreCurrentSelectionAs UserSelectionType=BlockerArea;

AddMacro MacroName=RBB_blocker;
SelectMacro MacroName=RBB_blocker;

ClearSelection;
AddToSelectionXY UpperLeftX=134 UpperLeftY=1 LowerRightX=148 LowerRightY=83;
ExpandSelection;
StoreCurrentSelectionAs UserSelectionType=ModuleArea;

AddLibraryElementFromXDL XDLMacro=~/GoAhead/BM/Connect4_V6_double.xdl
  ParseXDLPortStatements=True;

# Left macros
ClearSelection;
AddToSelectionXY UpperLeftX=130 UpperLeftY=80 LowerRightX=133 LowerRightY=83;
AddToSelectionXY UpperLeftX=130 UpperLeftY=75 LowerRightX=133 LowerRightY=78;
AddToSelectionXY UpperLeftX=130 UpperLeftY=70 LowerRightX=133 LowerRightY=73;
AddToSelectionXY UpperLeftX=130 UpperLeftY=65 LowerRightX=133 LowerRightY=68;
AddToSelectionXY UpperLeftX=130 UpperLeftY=59 LowerRightX=133 LowerRightY=62;
AddToSelectionXY UpperLeftX=130 UpperLeftY=54 LowerRightX=133 LowerRightY=57;
AddToSelectionXY UpperLeftX=130 UpperLeftY=49 LowerRightX=133 LowerRightY=52;
AddToSelectionXY UpperLeftX=130 UpperLeftY=44 LowerRightX=133 LowerRightY=47;
AddToSelectionXY UpperLeftX=130 UpperLeftY=38 LowerRightX=133 LowerRightY=41;
AddToSelectionXY UpperLeftX=130 UpperLeftY=33 LowerRightX=133 LowerRightY=36;
AddToSelectionXY UpperLeftX=130 UpperLeftY=28 LowerRightX=133 LowerRightY=31;
AddToSelectionXY UpperLeftX=130 UpperLeftY=23 LowerRightX=133 LowerRightY=26;
AddToSelectionXY UpperLeftX=130 UpperLeftY=17 LowerRightX=133 LowerRightY=20;
AddToSelectionXY UpperLeftX=130 UpperLeftY=12 LowerRightX=133 LowerRightY=15;
AddToSelectionXY UpperLeftX=130 UpperLeftY=7  LowerRightX=133 LowerRightY=10;
AddToSelectionXY UpperLeftX=130 UpperLeftY=2  LowerRightX=133 LowerRightY=5;
ExpandSelection;
AddMacroInstantiationInSelectedTiles SliceNumber=1
  InstanceName=left_data MacroName=Connect4_V6_double SortyBy=row SortOrder=asc;
StoreCurrentSelectionAs UserSelectionType=LeftData;

ClearSelection;
AddToSelectionXY UpperLeftX=130 UpperLeftY=43 LowerRightX=133 LowerRightY=43;
ExpandSelection;
AddMacroInstantiationInSelectedTiles SliceNumber=1
  InstanceName=left_results MacroName=Connect4_V6_double SortyBy=row SortOrder=asc;
StoreCurrentSelectionAs UserSelectionType=LeftResults;
```

```
51
52  ClearSelection;
53  AddToSelectionXY UpperLeftX=130 UpperLeftY=37 LowerRightX=133 LowerRightY=37;
54  ExpandSelection;
55  AddMacroInstantiationInSelectedTiles SliceNumber=1
56    InstanceName=left_state MacroName=Connect4_V6_double SortyBy=row SortOrder=asc;
57  StoreCurrentSelectionAs UserSelectionType=LeftState;
58
59  # Right macros
60  ClearSelection;
61  AddToSelectionXY UpperLeftX=150 UpperLeftY=80 LowerRightX=153 LowerRightY=83;
62  AddToSelectionXY UpperLeftX=150 UpperLeftY=75 LowerRightX=153 LowerRightY=78;
63  AddToSelectionXY UpperLeftX=150 UpperLeftY=70 LowerRightX=153 LowerRightY=73;
64  AddToSelectionXY UpperLeftX=150 UpperLeftY=65 LowerRightX=153 LowerRightY=68;
65  AddToSelectionXY UpperLeftX=150 UpperLeftY=59 LowerRightX=153 LowerRightY=62;
66  AddToSelectionXY UpperLeftX=150 UpperLeftY=54 LowerRightX=153 LowerRightY=57;
67  AddToSelectionXY UpperLeftX=150 UpperLeftY=49 LowerRightX=153 LowerRightY=52;
68  AddToSelectionXY UpperLeftX=150 UpperLeftY=44 LowerRightX=153 LowerRightY=47;
69  AddToSelectionXY UpperLeftX=150 UpperLeftY=38 LowerRightX=153 LowerRightY=41;
70  AddToSelectionXY UpperLeftX=150 UpperLeftY=33 LowerRightX=153 LowerRightY=36;
71  AddToSelectionXY UpperLeftX=150 UpperLeftY=28 LowerRightX=153 LowerRightY=31;
72  AddToSelectionXY UpperLeftX=150 UpperLeftY=23 LowerRightX=153 LowerRightY=26;
73  AddToSelectionXY UpperLeftX=150 UpperLeftY=17 LowerRightX=153 LowerRightY=20;
74  AddToSelectionXY UpperLeftX=150 UpperLeftY=12 LowerRightX=153 LowerRightY=15;
75  AddToSelectionXY UpperLeftX=150 UpperLeftY=7  LowerRightX=153 LowerRightY=10;
76  AddToSelectionXY UpperLeftX=150 UpperLeftY=2  LowerRightX=153 LowerRightY=5;
77  ExpandSelection;
78  AddMacroInstantiationInSelectedTiles SliceNumber=1
79    InstanceName=right_data MacroName=Connect4_V6_double SortyBy=row SortOrder=asc;
80  StoreCurrentSelectionAs UserSelectionType=RightData;
81
82  ClearSelection;
83  AddToSelectionXY UpperLeftX=150 UpperLeftY=43 LowerRightX=153 LowerRightY=43;
84  ExpandSelection;
85  AddMacroInstantiationInSelectedTiles SliceNumber=1
86    InstanceName=right_results MacroName=Connect4_V6_double SortyBy=row SortOrder=asc;
87  StoreCurrentSelectionAs UserSelectionType=RightResults;
88
89  ClearSelection;
90  AddToSelectionXY UpperLeftX=150 UpperLeftY=37 LowerRightX=153 LowerRightY=37;
91  ExpandSelection;
92  AddMacroInstantiationInSelectedTiles SliceNumber=1
93    InstanceName=right_state MacroName=Connect4_V6_double SortyBy=row SortOrder=asc;
94  StoreCurrentSelectionAs UserSelectionType=RightState;
95
96  ClearSelection;
97  AddToSelectionXY UpperLeftX=130 UpperLeftY=2 LowerRightX=133  LowerRightY=5;
98  AddToSelectionXY UpperLeftX=130 UpperLeftY=7 LowerRightX=133  LowerRightY=10;
99  AddToSelectionXY UpperLeftX=130 UpperLeftY=12 LowerRightX=133 LowerRightY=15;
100 AddToSelectionXY UpperLeftX=130 UpperLeftY=17 LowerRightX=133 LowerRightY=20;
101 AddToSelectionXY UpperLeftX=130 UpperLeftY=23 LowerRightX=133 LowerRightY=26;
102 AddToSelectionXY UpperLeftX=130 UpperLeftY=28 LowerRightX=133 LowerRightY=31;
103 AddToSelectionXY UpperLeftX=130 UpperLeftY=33 LowerRightX=133 LowerRightY=41;
104 AddToSelectionXY UpperLeftX=130 UpperLeftY=43 LowerRightX=133 LowerRightY=47;
105 AddToSelectionXY UpperLeftX=130 UpperLeftY=49 LowerRightX=133 LowerRightY=52;
106 AddToSelectionXY UpperLeftX=130 UpperLeftY=54 LowerRightX=133 LowerRightY=57;
107 AddToSelectionXY UpperLeftX=130 UpperLeftY=59 LowerRightX=133 LowerRightY=62;
108 AddToSelectionXY UpperLeftX=130 UpperLeftY=65 LowerRightX=133 LowerRightY=68;
109 AddToSelectionXY UpperLeftX=130 UpperLeftY=70 LowerRightX=133 LowerRightY=73;
110 AddToSelectionXY UpperLeftX=130 UpperLeftY=75 LowerRightX=133 LowerRightY=78;
111 AddToSelectionXY UpperLeftX=130 UpperLeftY=80 LowerRightX=133 LowerRightY=83;
112 AddToSelectionXY UpperLeftX=149 UpperLeftY=2 LowerRightX=153  LowerRightY=5;
113 AddToSelectionXY UpperLeftX=149 UpperLeftY=7 LowerRightX=153  LowerRightY=10;
114 AddToSelectionXY UpperLeftX=149 UpperLeftY=12 LowerRightX=153 LowerRightY=15;
115 AddToSelectionXY UpperLeftX=149 UpperLeftY=17 LowerRightX=153 LowerRightY=20;
116 AddToSelectionXY UpperLeftX=149 UpperLeftY=23 LowerRightX=153 LowerRightY=26;
117 AddToSelectionXY UpperLeftX=149 UpperLeftY=28 LowerRightX=153 LowerRightY=31;
118 AddToSelectionXY UpperLeftX=149 UpperLeftY=33 LowerRightX=153 LowerRightY=41;
119 AddToSelectionXY UpperLeftX=149 UpperLeftY=43 LowerRightX=153 LowerRightY=47;
120 AddToSelectionXY UpperLeftX=149 UpperLeftY=49 LowerRightX=153 LowerRightY=52;
121 AddToSelectionXY UpperLeftX=149 UpperLeftY=54 LowerRightX=153 LowerRightY=57;
122 AddToSelectionXY UpperLeftX=149 UpperLeftY=59 LowerRightX=153 LowerRightY=62;
123 AddToSelectionXY UpperLeftX=149 UpperLeftY=65 LowerRightX=153 LowerRightY=68;
124 AddToSelectionXY UpperLeftX=149 UpperLeftY=70 LowerRightX=153 LowerRightY=73;
125 AddToSelectionXY UpperLeftX=149 UpperLeftY=75 LowerRightX=153 LowerRightY=78;
126 AddToSelectionXY UpperLeftX=149 UpperLeftY=80 LowerRightX=153 LowerRightY=83;
127 ExpandSelection;
128 StoreCurrentSelectionAs UserSelectionType=blockWithoutEE2andWW2;
129
130 SelectUserSelection UserSelectionType=ModuleArea Keep=False;
```

```
131 InvertSelection;
132 PrintProhibitStatementsForSelection ExcludeUsedSlices=True
133   FileName=X:\blocker_test\partial_small_device.ucf Append=False;
134
135 SelectUserSelection UserSelectionType=blockWithoutEE2andWW2 Keep=False;
136 MarkPortsInSelectionAsUsedByRegexp PortNameRegexp=EE2
137   CheckForExistence=False IncludeAllPorts=False;
138
139 SelectUserSelection UserSelectionType=BlockerArea Keep=False;
140 BlockSelection PrintUnblockedPorts=False
141   Prefix=RBB_Blocker BlockWithEndPips=True SliceNumber=0;
142
143 SelectUserSelection UserSelectionType=ModuleArea Keep=False;
144 AddArcs From=WW2END0 To=WW2BEG0;
145 AddArcs From=WW2END1 To=WW2BEG1;
146 AddArcs From=WW2END2 To=WW2BEG2;
147 AddArcs From=WW2END3 To=WW2BEG3;
148
149 GenerateXDL FileName=X:\blocker_test\partial_blocker_small_device.xdl
150   MacroNames=RBB_blocker IncludePorts=False IncludeDummyNets=True
151   IncludeDesignStatement=False IncludeModuleHeader=False
152   IncludeModuleFooter=False DesignName=__XILINX_NMC_MACRO;
153 GenerateXDL FileName=X:\blocker_test\partial_blocker_small_device_FE.xdl
154   MacroNames=RBB_blocker IncludePorts=False IncludeDummyNets=True
155   IncludeDesignStatement=True IncludeModuleHeader=True
156   IncludeModuleFooter=True DesignName=__XILINX_NMC_MACRO;
157
158 PrintLocationConstraintsForPlacedMacros Prefix= InstancePrefix=
159   FileName=X:\blocker_test\partial_small_device.ucf Append=True;
160
161 SelectUserSelection UserSelectionType=ModuleArea Keep=False;
162 PrintAreaConstraint InstanceName=*module_wrapper*
163   FileName=X:\blocker_test\partial_small_device.ucf Append=True;
164
165 ShowGUI;
```