A System for Fast Dynamic Partial Reconfiguration using GoAhead

Anders Hauk Fritzell

12th August 2013
Abstract

Reconfigurable hardware in the form of Field Programmable Gate Arrays (FPGAs) was brought to market almost three decades ago, but many designs are still not using the full potential of the devices. By designing and implementing systems using partial runtime reconfiguration of the device, it is possible to achieve more efficient use of resources. For example, the impact of static power consumption could be reduced by using a PR design flow to make a FPGA design fit onto a smaller device.

The goal of this thesis is to design and implement an open-source system for partial reconfiguration (PR) using the GoAhead tool flow. This system is optimized for several different FPGA platforms that are based on many different Xilinx devices, including commonly used academic board. It is also designed to provide good reconfiguration speeds, since one of the biggest hurdles in PR is the overhead resulting from the reconfiguration time. This system can also act as a platform for further studies on PR by students and researchers interested in reconfiguration using the GoAhead framework.

The implemented system is built around a baseline MIPS CPU, designed particularly for this project. Reconfigurable custom instructions are added as an extension to the MIPS using GoAhead. Connected to the MIPS over the system bus is a configuration controller capable of writing configuration data to the device from both a host-PC and external flash memory. Furthermore, the implemented configuration controller has support for compressed bitstreams and module relocation. A large PR region with a streaming interface was implemented at the top of the device to allow for PR modules in the system.

The final system is capable of partial reconfiguration of custom instructions and PR modules. The use of custom instructions is observed to save 518 cycles compared to a software implementation of the same function. By using compressed bitstreams and a 100 MHz system clock, the configuration controller is able to produce a configuration data throughput of 97 MB/s directly from external flash memory. Furthermore, the configuration controller supports two-dimensional relocation without any CPU intervention during the configuration process.
# Contents

1 **Introduction**  
1.1 Motivation ............................................. 2  
1.2 Chapter overview .................................. 3  

2 **Background**  
2.1 Field Programmable Gate Arrays .................. 5  
2.1.1 Technology ........................................ 5  
2.1.2 Configuration details ............................ 6  
2.1.3 Architecture ...................................... 6  
2.2 Reconfigurable hardware ......................... 9  
2.3 Partial reconfiguration ............................. 10  
2.3.1 Styles for module placement .................. 11  
2.3.2 Module footprint ................................ 14  
2.3.3 Spartan-6 configuration ....................... 15  
2.3.4 Relocation of partial module bitstreams ..... 17  
2.3.5 Algorithm for bitstream relocation ........ 17  
2.3.6 Bitstream compression for reconfiguration 19  
2.4 Design flows for partial reconfiguration .... 19  
2.4.1 Xilinx PlanAhead PR flow ..................... 20  
2.4.2 OpenPR .......................................... 22  
2.4.3 GoAhead ......................................... 22  
2.4.4 GoAhead compared to PlanAhead and OpenPR 23  
2.5 Runtime reconfigurable modules ............... 25  
2.5.1 Benefits of using runtime reconfigurable modules 25  
2.6 MIPS architecture ................................. 27  
2.6.1 RISC ............................................ 27  
2.6.2 MIPS overview .................................. 27  
2.6.3 MIPS instruction set ............................ 28  
2.7 Reconfigurable instruction set extension .... 28  
2.7.1 Custom instructions in hardware ............ 29  
2.7.2 Custom instructions in software ............. 30  
2.7.3 Reconfiguration of custom instructions .... 30  

3 **System implementation**  
3.1 System overview .................................... 33  
3.2 Hardware platform .................................. 35  
3.3 System bus ......................................... 36  

v
3.3.1 Module testing ...................................... 36
3.4 Baseline MIPS ....................................... 37
   3.4.1 Implementation ................................ 38
   3.4.2 Software and testing ............................ 40
3.5 Configuration controller ............................. 41
   3.5.1 Flash controller ................................ 42
   3.5.2 SPI flash reader ................................. 42
   3.5.3 Decompression module ............................ 45
   3.5.4 UART receiver .................................. 45
   3.5.5 ICAP .............................................. 46
3.6 Configuration controller for module relocation ..... 46
   3.6.1 Implementation ................................ 46
3.7 Custom instructions .................................. 48
   3.7.1 Hardware implementation ....................... 48
   3.7.2 Software implementation ....................... 53
   3.7.3 Reconfigurable decoder for custom instructions 54
3.8 Partial reconfigurable region ......................... 55
   3.8.1 Implementation ................................ 55
3.9 Video input and output ................................ 58
   3.9.1 Video buffering ................................ 58
3.10 Challenges during implementation .................... 59
   3.10.1 Custom instructions ............................. 59
   3.10.2 DDC2 module for plug-and-play video ........ 60

4 Test cases and results ................................. 61
   4.1 Custom instructions ................................ 61
      4.1.1 Introduction .................................. 61
      4.1.2 Setup .......................................... 61
      4.1.3 Configuration controller ....................... 63
      4.1.4 Custom instructions ............................. 63
      4.1.5 Reconfiguration results ........................ 64
   4.2 Placing and relocating PR modules ................... 65
      4.2.1 Introduction .................................. 65
      4.2.2 Setup .......................................... 66
      4.2.3 Test module .................................... 66
      4.2.4 Placement and bitstream generation ............ 67
      4.2.5 Relocation results ............................. 69

5 Conclusion and further work .......................... 71
   5.1 System performance ................................ 71
   5.2 Future work ....................................... 72

A MIPS custom instructions with inline assembly ....... 73
   A.1 Introduction ....................................... 73
   A.2 Modifying binutils source .......................... 73
   A.3 Building crosscompiler .............................. 74
B VHDL code
   B.1 MIPS CPU .............................................. 75
   B.2 SPI flash reader ....................................... 85

C Scripts
   C.1 Batch-script ........................................... 91
   C.2 GOA-script ............................................. 93
List of Figures

2.1 3-input LUT implementing the function $Y = (A_0 \cdot \overline{A_2}) + (A_0 \oplus A_1) \cdot A_2$ ................................. 7

2.2 The logic fabric of the FPGA presented in a simplified way. The interconnect is displayed as black and grey buses connecting to the switch matrices. The logic is presented as CLBs on the right side of each switch matrix, and each CLB contains two slices. ................................. 8

2.3 Differences between island style and slot style PR. Slot style allows for a more efficient use of resources. In slot style placement, the slot not used by M1 can be used by another module (M2). In island style placement, the area not used by M1 would be left unused as internal fragmentation (which would be even higher with M2 placed in the island). ........ 11

2.4 Graph representation of equation 2.1 taken from "Partial Reconfiguration on FPGAs"[16]. The graph displays how communication cost and slot size affect average overhead for a random set of modules being 300-10000 LUTs in size. .... 13

2.5 External fragmentation. M3 can’t be placed because of poor placement of M2, not because of too little resources. .... 13

2.6 Resource footprint. Each module can only be placed at positions within the PR region were the module footprint fits the FPGA footprint. ................................. 14

2.7 3 ways of implementing the interface between static- and partial system. Red colour represents the the routing and logic generated during implementation of the partial modules, light blue represents everything added during implementation of the static system. The dark blue wires are the signals creating the interface between static- and partial system. ................................. 21

2.8 GoAhead GUI, with the FPGA resources represented as colored tiles. ................................. 24

2.9 Tool flow for GoAhead and ISE. Figure taken from [6] .... 24
2.10 A) Time and area consumption for a static implementation with module x, y and z. B) Implementation using reconfiguration, only one module is placed on the device at one time. This allows for a smaller device, since area requirements are lower. Modules are changed using reconfiguration (indicated with red in the graph). C) An implementation that utilizes the full size the device used for the static implementation, but only for one module at a time. This allows more resources to spent per module, allowing for speed-up of execution for some problems.

2.11 Flow chart of a possible solution for reconfiguration and handling of custom instructions by using trap handlers and software emulation of CIs.

3.1 System

3.2 Atlys development board

3.3 An overview of the MIPS. Only the most important signals are included in this figure.

3.4 An overview of the Program Counter. The address nextPC is passed on to the instruction memory instead of PC to allow for execution of one instruction per clock cycle. The register for delayed branch is pure overhead in our case, because of the lack of pipelining in the design.

3.5 Figure of the critical combinatorial path for the multiplication instruction which is allow two clock cycles for execution.

3.6 Block view of configuration controller.

3.7 Flash controller ASM diagram. The signal names correspond to the ones in Figure 3.6.

3.8 This figure show how the flash controller is changed to allow for module relocation. It replaces the flash controller in Figure 3.6 to form a complete configuration controller capable of module relocation.

3.9 The MIPS ALU extended with custom instructions (CIs). The two slots for CIs acts as a extension of the ALU, allowing for two more instructions that can be reconfigured at any time.

3.10 Drawing of hard macro placement and routing for two slots west of CPU. White boxes are connection macros placed in CLB columns. Blue and red rectangles are DSP and BRAMs. Curved arrows indicate routing. We used double and quad wires which route two respectively four columns far. Brackets at the top of the figure indicates if the enclosed macros are placed in static or partial part of the implementation.

3.11 Implementation of static system. Connection macros (yellow squares) placed inside the partial region. Picture taken in GoAhead GUI.

3.12 PR region for custom instructions after routing in Xilinx fpga_editor.
3.13 Implementation of partial module. Connection macros (yellow squares) placed in the static region. Picture taken in GoAhead GUI.

3.14 PR region for custom instructions after routing. Picture taken in Xilinx FPGA Editor.

3.15 Selecting a CI slot is done by decoding the funct-field of the custom instruction as an input to a table, which stores the select signal for the slot multiplexer and consequently, for a placed custom instruction.

3.16 Two connection macros are placed in the static region to define the streaming interface using double wires. This figure shows a simplified view of how the double wires are used to create a streaming interface across the PR region. Only the top and the bottom row of the streaming interface is shown. The two set of double wires are coloured blue and red. They are connected to connection macros in the same colour.

3.17 Screenshot from the FPGA Editor showing the connection macros and the connected double wires within the PR region.

3.18 Screenshot from the FPGA Editor showing a routed partial module.

3.19 Picture of video input/output with video overlay modules running on the board. In this picture we us three different modules, one skin color detection module, two Pacman modules and a module for displaying hex-values.

4.1 Overview of the creation of multiple bitstreams for different placements of the same module netlist, using GoAhead and bitgen.

4.2 Picture of two packman modules running at two different slots within the PR region. Both modules were configured into the system using the configuration controller with one bitstream and manipulation of the Frame Address Register within the bitstream.
List of Tables

2.1 Acheivable configuration speeds with ICAP[12] ........... 6
2.2 Comparison of different PR flows for Xilinx devices, based on table presented in [6] .................. 20

4.1 Configuration controller, resource requirements. ........ 63
4.2 Custom instruction, resource requirements. ............... 64
4.3 Custom instruction, bitstream size. ....................... 64
4.4 Reconfiguration results. Bitstream size is the number of bytes that has to be sent to ICAP. Configuration clock cycles is the number of clock cycles the whole reconfiguration process requires. Software requirement is the number clock cycles a software implementation of the custom instruction requires on the MIPS CPU. .................. 64
4.5 Enhanced configuration controller, resource requirements. 65
4.6 Pacman bitstream size .................. 68
Acknowledgement

I would like to thank my supervisors Dirk Koch and Jim Torresen for great support during my work with this thesis.
Chapter 1

Introduction

Field Programmable Gate Array (FPGA) technology has become popular in industry for allowing designers to create complex digital designs without facing the investment resources required for producing an Application Specific Circuit (ASIC). Modern FPGAs can have over 100000 logic cells, Digital Signal Processing Blocks (DSPs), Block RAM (BRAM), memory controllers and high speed I/O on chip, making them very capable devices. Most modern FPGA devices are also partially reconfigurable, meaning that they can change configuration for parts of the device during runtime. This feature makes FPGA devices interesting for researchers working in the field of Reconfigurable Computing and Adaptive Hardware.

By changing all or parts of the hardware during execution it is possible to gain efficiency over static systems. A typical problem with modern FPGA devices is the significantly lower efficiency in area, power and speed compared to ASICs. In 90 nm technology it has been shown that when only using logic blocks and no hard blocks (DSPs, BRAM, multipliers and so on) the FPGA design required on average 35 times more area than the same design in ASIC technology [17]. When hard blocks were used, the difference was reduced to on average 25 times the area. In modern CMOS technology, transistor leakage current has become a major source of static power consumption for devices. By designing for more efficient use of resources, smaller FPGA devices can be used for the implementation, lowering the static power consumption and also the cost of the device. By using runtime partial reconfiguration (often called PR) it is possible to swap modules in and out of the FPGA fabric during execution to save resources and area. This of course requires the design to have modules that execute mutually exclusive in the time domain.

Some computational problems can gain benefits from a partial reconfiguration approach. For instances, designers of soft core CPUs can use partial reconfiguration to perform instruction set extension to speed-up frequently used functions or code blocks [27]. Another way to make general purpose processors faster on specific problems is to add dedicated hardware acceleration. With the help of PR, it is possible to use far more hardware modules than there is room for on the chip. One example is a system for video based driving assistance where the coprocessors accelerate algorithms for pixel
operations on the video [9]. Since only a subset of the algorithms are used at one time, PR can reduce the resource footprint on the FPGA by allowing runtime swapping of coprocessors.

1.1 Motivation

This thesis is focused on creating a system that can be used as a demonstration and learning platform for PR using a design flow provided by the tool GoAhead [6]. The system will provide a basis for further testing and experimentation with PR and the GoAhead tool flow. One important point with the thesis is to limit the use of restricted IP (intellectual property) cores and try to implement most of the design for portability between different devices.

Moreover, implementing working reconfigurable system takes several additional steps as compared to the implementation of a static only system. This includes:

1. Floorplanning where we have to decide which regions of the FPGA will be reconfigured, and where we define module bounding boxes.

2. Communication infrastructure generation in order to constrain routing resources for the communication with reconfigurable modules.

3. Adding a configuration controller which executes reconfiguration requests by sending partial configuration bitstreams to the FPGA.

4. Modification to the physical implementation all the way to the final configuration bitstream.

This thesis will provide a generic and portable PR system allowing an easy entry in designing reconfigurable systems. Nevertheless, the system is still providing some very sophisticated features including dynamically reconfigurable custom instructions set extension of a softcore CPU and a large area able to host multiple relocatable modules at the same time. For the reconfiguration management, a controller has been implemented supporting different Xilinx FPGA families. This controller can send a bitstream from a serial COM-port or an on-board flash memory (which is available on most popular FPGA development boards used in academia). By using hardware accelerated bitstream decompression, it is possible to configure at close to 100 MB/s.

The system will be built around a baseline MIPS CPU, with some on-chip memory and a instruction ROM. In the static part of the system there will also be modules for general purpose I/O, video encoders and decoders, as well as connectors to an AC97 audio codec and general purpose I/O.
1.2 Chapter overview

Chapter 2: Background
This chapter provides an introduction to reconfigurable hardware and FPGA architecture. Then, it continues on to concepts regarding partial reconfiguration, focusing on module placement, design flows and details regarding reconfiguration of FPGA devices. The last part of this chapter introduces the MIPS architecture and reconfigurable instruction set extension.

Chapter 3: System implementation
In this chapter the implementation of the system is presented. It covers design decisions and implementation of all the components that makes the final system.

Chapter 4: Results
This chapter is about the measured performance of the system, focusing on reconfiguration speed and custom instruction performance. The chapter contains two test cases, with the first focusing on reconfiguration of custom instructions, and the second on reconfiguration of accelerator modules using a concept called module relocation.

Chapter 5: Conclusion and further work
This chapter summarizes the thesis and presents some thoughts on further improvements and use of the implemented system.

Appendix
Appendix A contains a short description on how to set up your own MIPS cross compiler and how to use custom instructions with inline assembly. Appendix B contains the VHDL-code for the MIPS CPU and the SPI flash reader. In Appendix C, two scripts are appended, one to show the tool flow with ISE and GoAhead, and another to show how system creation in GoAhead can be automated.
Chapter 2

Background

2.1 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGA) are logic devices that have a large quantity of programmable cells that can be programmed to perform any logical function. FPGA technology lies between the simpler and smaller Programmable Logic Devices (PLDs) and the more expensive and development heavy Application Specific Integrated Circuits (ASICs). PLDs are a collection of many different types of devices that implement logic functions through programmable changes to the internal interconnect. The internal interconnect in simple PLDs (SPLDs) is often some kind of EEPROM or ROM based array of programmable links that define connections between inputs, logic gates and outputs. SPLDs are designed for smaller functions and can not implement large functions because of limitations in inputs and logic outputs. Complex Programmable Logic Devices (CPLDs) connects many SPLDs together with use of programmable multiplexers and interconnect between them. FPGAs are a step up from the CPLDs, with the capability to implement larger functions and systems. As mentioned FPGAs contain a large amounts of programmable logic blocks. Modern FPGAs may also have CPU-cores, Digital Signal Processors (DSPs) and Block RAMs (BRAMs) embedded in the fabric together with the programmable logic.

2.1.1 Technology

All FPGAs are based on memory elements to hold the configuration of the device. There are three common technologies used to implement memory for the configuration bits in an FPGA: FLASH (EEPROM), antifuse and SRAM (Static Random Access Memory). FLASH is non-volatile memory and retains configuration data after power is removed from the device, while SRAM is volatile and needs to be programmed from an external memory each time power is applied to the device. FLASH is based upon EEPROM (Electrical Erasable Programmable Memory) technology and is added as an extra process step upon CMOS production process. Configuration cells using SRAM memory are designed using
CMOS transistors to form a latch. In antifuse the configuration of the device is defined by creating permanent connections in the configuration cells. FPGAs based on antifuse are one time programmable, and after programming the configuration process cannot be redone. As with FLASH, antifuse requires extra process steps upon the standard CMOS process [19].

The only devices on the market that supports fast configuration in circuit is SRAM based devices. Some FLASH devices can also perform in-circuit configuration, but not as fast and as many times as SRAM devices. Partial reconfiguration in the form described in this thesis can only be performed on some SRAM-based devices from Xilinx.

2.1.2 Configuration details

Configuration of FPGAs are done through the writing of a bitstream to one of the configuration ports of a device. There exists both external and internal configuration ports with different interfaces to accommodate specific protocols and connections. The bitstream consists mostly of data for the SRAM cells holding the configuration of the device. Xilinx FPGAs support reconfiguration of regions on the device during runtime. The smallest region that is reconfigurable is called a configuration frame and varies in size depending on device.

Internal Configuration Access Port (ICAP)

To carry out reconfiguration during runtime, the system needs to write configuration data into the configuration cells. On Xilinx devices, this means writing data to the Internal Configuration Access Port (ICAP). ICAP is the internal version of SelectMap port; one of the external configuration ports on Spartan-6. Table 2.1 shows achievable configuration speeds with ICAP.

Table 2.1: Achievable configuration speeds with ICAP[12]

<table>
<thead>
<tr>
<th>Bit width</th>
<th>Frequency</th>
<th>Configuration speed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MHz</td>
<td>Mb/s / MB/s</td>
</tr>
<tr>
<td>8 bit</td>
<td>100</td>
<td>800 / 100</td>
</tr>
<tr>
<td>16 bit</td>
<td>100</td>
<td>1600 / 200</td>
</tr>
</tbody>
</table>

The ICAP primitive has an input dataport (I) which accepts 8- or 16-bit words of configuration data on Spartan-6 devices. The output port (O) is used for read-back of configuration data already present on the device. The primitive is controlled by setting the write enable (WRITE) and clock enable (CE) signals. Data is read or written by the primitive on the rising edge of the clock (CLK).

2.1.3 Architecture

The fabric of the FPGA can be seen as pattern of logic blocks with interconnect going horizontal and vertical between them. The basic
element in FPGA architecture is the Look Up Table (LUT), which have 3-6 inputs depending on device and manufacturer and can implement any n-bit boolean function, where n is the number of LUT inputs. A LUT is basically a multiplexer with inputs taken from configuration memory and the output selected by the LUT input signals. In hardware a LUT is often designed using storage elements (SRAM, FLASH, antifuse) and transmission gates (See Figure 2.1).

In the majority of FPGAs SRAM cells are used to hold the configuration that defines the boolean function. During reconfiguration, these SRAM cells get overwritten with new functions. When a LUT is combined with configurable registers and multiplexers for implementing the routing, we have a logic cell. This is the main building block for the FPGA fabric, all logic that is not mapped to special blocks like DSPs, CPUs or BRAMs is implemented in logic cells. In recent Xilinx FPGAs, four logic cells are combined to form a slice, and two slices are often combined to form a Configurable Logic Block (CLB) \(^1\). Slices can contain more logic than just basic logic cells to be able to implement fast carry chains, shift registers and distributed RAM. This is mostly done by adding dedicated wires and logic between slices in the same column to propagate signals through many slices without needing to route through the interconnect.

In Xilinx Spartan-6 FPGAs there are 3 types of slices:

\(^1\)At least in current generation of Xilinx devices. In Altera devices CLBs are called Logic Array Blocks (LABs)
Figure 2.2: The logic fabric of the FPGA presented in a simplified way. The interconnect is displayed as black and grey buses connecting to the switch matrices. The logic is presented as CLBs on the right side of each switch matrix, and each CLB contains two slices.

- SLICEX
- SLICEL
- SLICEM

The basic slice is called SLICEX and do not contain any special routing or logic. Every CLB contains one SLICEX and one slice that’s either a SLICEL or SLICEM. SLICELs have extra carry logic and can implement wide multiplexers, SLICEMs have everything from the SLICELs with added support for shift registers and distributed RAM. All slices are located in a XY-grid were each slice is given a X-value for which column of slices it is in and a Y-value for which row of CLBs. So, the first CLB in the starting corner of a FPGA will contain the slices X0Y0 and X1Y0, the next CLB in the same row will contain X2Y0 and X3Y0, and so on. CLBs columns are often distinguished by which type special slice they contain. In this thesis we will use the name CLEXL for CLBs containing SLICELs and CLEXM for CLBs with SLICEMs. A simplified view of the FPGA architecture with slices and interconnect is shown in Figure 2.2.
To connect all columns and rows of CLBs, DSPs and BRAMs together there is a "sea" of programmable interconnect that goes horizontal and vertical between them. A CLB is connected to this interconnect through a switch matrix, a collection of programmable wires that connects to wires in the interconnect. Every wire that connect to a switch matrix can be routed to any input on the two slices in the CLB. In Xilinx Spartan-6 the wires in the interconnect are defined by which CLBs they connect together; a fast wire connects the outputs back to the inputs of a CLB, the single wire connects the outputs of a CLB to the adjacent CLB in horizontal or vertical direction. The double wire connects every other CLB in the row or column and the quad wire connects every fourth CLB. Double and quad wires can also connect tiles diagonally.

Besides columns containing CLBs, Xilinx FPGAs contain (as already mentioned) coarse-grained components like DSPs and BRAMs in their own columns. The DSP blocks on Spartan-6 are called DSP48A1 and each block can take two 18-bit values. Within each DSP there is a multiplier and accumulator, and also pipeline registers and dedicated routing for connecting to neighbouring DSP48-blocks [28]. The BRAMs are memory blocks, where each block has the capacity of 18 K bits and can be accessed by 2-ports. The direction of the ports can be configured, but are limited by which mode the BRAM is configured to. The port width is adjustable between 1-36 bits. A 18 K block can be split into two 9 K blocks [30].

2.2 Reconfigurable hardware

Processors for computing can be divided into 3 groups: general purpose, domain-specific and application-specific[8]. The general purpose processor (GPP) uses memory, data path and control path to perform any computation without changing the underlying hardware. This gives the GPP high flexibility, but requires that the algorithm behind the computation is written as a sequential set of instructions. If the underlying algorithm is inherently parallel the sequential execution on a GPP will not yield best performance. If the processor is only going to be used in one specific field of computation the domain-specific processor may be a better choice then a GPP. Domain-specific processors have data paths with operations that are optimized for a set of algorithms, reducing flexibility, but increasing performance for the target domains. For best performs (and no flexibility) the application-specific processor (ASIP) is the way to go. ASIPs implement the algorithm directly in hardware and doesn’t use instructions, this means it is not limited by the requirement for sequential execution as the general purpose- and domain-specific processor are.

Modern FPGA (section 2.1) technology makes it possible to in many ways take the best features of the GPP and combine them with the power of ASIPs. By changing all or parts of the hardware structure during execution we get hardware that adapts to new applications on-the-fly. Runtime reconfiguration can (when used correctly) create flexible hardware that can remove downsides with FPGA technology. One problem today is the
high static power consumption of modern FPGA devices, which have a large number of transistors, a problem that could be made less significant by increasing device utilization through reconfiguration of the device [16]. By switching hardware modules during execution and only have relevant modules for current execution in circuit, a smaller and consequently less power-hungry FPGA might be used.

In a perfect world we would like a processor with the performance of an ASIP and the flexibility of the GPP, one solution is hardware that can adapt to different problems. This is called reconfigurable hardware and is done by changing the structure of the hardware for all or parts of the device [8]. One problem today is the high static power consumption of modern FPGA devices which have a large number of transistors, a problem that could be made less significant by increasing device utilization through partial reconfiguration [16]. By exchanging modules on demand and not having all modules lying in circuit creates possibilities for both performance- and power gains for many problems. This will bring the FPGA closer to ASIP in performance due to more efficient use of resources, but it will not work for all problems. To be able to divide a problem in a way that fits partial reconfiguration the reconfigurable modules used needs to be mutually exclusive in time and space [16] to make it possible to switch between them without interference to the wanted operation of the circuit. This makes partial reconfiguration especially suited for problems that have 2 or more clearly separated computational tasks.

2.3 Partial reconfiguration

To be able to do partial runtime reconfiguration, the devices need to support this in hardware. Reconfiguration in one part of the device should not halt operation in other parts of the device. There are many ways of doing PR, from small netlist changes to routing and LUT-functions, to replacement of large modules. This section will focus on:

- **Multi-cycle reconfiguration**: Reconfiguration takes more than one cycle of the system clock since reconfiguration data is written from memory into configuration cells. **Single-cycle** is when reconfiguration is done by switching between logic already on the device within one cycle of the system clock. This could be done in devices with time-multiplexing between two or more sets of configuration cells [24].

- **Run-to-completion modules**: context-switching will not be used. The internal state of modules that is removed from hardware will not be stored. Modules will be replaced when needed by the system and not by time-slots given by a scheduler. However, when the state of a module can be accessed by for example a CPU that reads or writes all memory elements of a module, this CPU can implement a form of context switching. This would not be context switching in the hardware sense of doing a read-back of the configuration and storing it, but would serve many of the same purposes.
2.3.1 Styles for module placement

There are different ways to design for partial reconfiguration, like how the region dedicated for PR is used (see also [16] for a classification of PR into configuration styles). One way of doing PR is replacing larger chunks of logic called modules for each reconfiguration, this called module-based reconfiguration [16] and will be the focus of this thesis. Module-based PR can be implemented in a couple of different ways depending on what is wanted of the system. The PR region were the PR modules reside can be one module exclusive or tiled in one or two dimensions for placement of multiple modules in one region.

Island style

This is the style supported by the Xilinx PR flow. If only one island is used it’s often called ”single island style”, if there is more than one island ”multi island style” is used. In island style PR, only one module can occupy the PR region at one time. The static only system would be (for comparison) to have all modules in the static system and switching between them with multiplexers and demultiplexers. Since the PR region should be able to accommodate all modules needed by the system, the region needs to contain all resources required for the different modules. If one PR module requires BRAMs this means that all other modules that don’t use them will still occupy the resource as overhead, since in ”island style” PR module placement there is no flexibility within one island. This is called internal fragmentation and can be observed in Figure 2.3.

If a PR module is to be used in a design with multiple islands, the
islands need to have the same resources. The most straightforward way of handling multiple islands is to generate a new partial bitstream for each placement of the PR module, so if a module is to be placed in \( N \) different islands, the designer has to generate \( N \) different partial bitstreams for that module. This is how Xilinx handles placement of the same module in different islands/PR regions.

**Slot style**

If more flexibility is wanted in the regard to module placement and efficient use of resources, then "slot style" or "grid style" solutions may give higher flexibility. In "slot style" PR, the PR region is divided into slots of fixed size. Modules are allowed to use multiple adjacent slots to best fit their resource requirement, and each PR region is not limited to one module as in "island style" PR. Some aspects of the PR design will get more complicated when slots are used. Slots are often designed with a common interface between them and to the rest of the world. This interface will represent an overhead for each slot/tile, but such communication interfaces are necessary for module relocation between slots. The size of the slots is very important, since this determines how efficient resources within a PR region can be distributed between modules. With small slots the granularity for module placement gets better and internal fragmentation is reduced, but with smaller slots the communication overhead grows. This is possible to calculate with methods given in [16]. For all modules \( m_i \in M \), where \(|m_i|\) is the size of a module given in LUTs and the communication cost is \( c \) LUTs. If one slot provides \( \sigma \) LUTs, then \( \frac{\sigma}{2} \) LUTs will wasted due to internal fragmentation on average.

\[
\bar{O} = \frac{1}{|M|} \sum_{i=1}^{|M|} \left( \left\lceil \frac{|m_i|}{\sigma - c} \right\rceil \cdot \sigma - |m_i| \right) 
\] (2.1)

With equation 2.1 it is then possible to calculate average module overhead \( \bar{O} \) and get good estimates for optimal slot size when considering communication overhead and fragmentation. Figure 2.4 shows calculations on overhead done with equation 2.1 for different module sizes and communication costs. The figure shows that an optimal slot size is between 350 and 450 LUTs, smaller slots will be affected by communication cost and larger slots will lose resources to fragmentation. Replacing modules will not be as straightforward as "island style" PR where the only decision is in which island to do the reconfiguration. Varying slot requirements for different modules can lead to fragmentation challenges inside the PR region. This is shown in figure 2.5 were module \( M3 \) can’t be placed because of external fragmentation created by not optimal placement of other modules. Some of the problems with external fragmentation can be avoid by allowing module reallocation [16].
Figure 2.4: Graph representation of equation 2.1 taken from "Partial Reconfiguration on FPGAs"[16]. The graph displays how communication cost and slot size affect average overhead for a random set of modules being 300-10000 LUTs in size.

Figure 2.5: External fragmentation. M3 can’t be placed because of poor placement of M2, not because of too little resources.
Relocating modules between different islands/slots on the device requires the designer to think about the resources needed for the module and also the underlying FPGA architecture and how resources are distributed on the device[6]. A PR module has a resource footprint and this footprint has to match the underlying FPGA resource footprint, so if a module is relocated to a new set of slots, these slots need to exactly match the footprint of the module. In section 2.1, the FPGA footprint of a Spartan-6 device was presented, with different resources in columns and wires connecting these together. If a module would require features only present in SLICEMs (i.e., LUTs providing distributed memory), then placement is constrained to only slots or islands containing that resource. The FPGA footprint for the placement position has to be the same as the module footprint. The concept is illustrated in Figure 2.6.

One way of looking at the resource footprint of a module is a string of characters were each character represents a resource tile containing a CLB (can be further divided in CLEXM and CLEXL), BRAM or DSP. CLBs could be represented by C, BRAMs by B and DSPs by D. A module that is 3 columns wide and 1 row high could have the following resource string: C C B. The whole FPGA could also be represented by one long string and valid placement positions could be found by matching the module footprint.
resource string against the FPGA resource string. One row of such a string could look as follows: C C D C C B C C C C C B ...... D C C. Special characters could be used to represent don’t care situations. Most columns contain the same resources from top to bottom of the device, but there are regions where this is not true. If the PR region is placed in a region of the device were all the tiles within one column are the same, then only one row is needed to represent the resource footprint of the PR region. A placer can find suitable positions for placement by taking the resource string of a module and slide it over the FPGA resource string until a match occurs, this indicates a valid placement for the module. In many cases there is also necessary to separate between different types of CLBs, if the module uses e.g. distributed memory. So, the string matching would also contain information about the type of CLB (CLEXM versus CLEXL).

It is not only resources that have different footprints, the available wires can also change. Situations can arise where signals from the static system needs to be routed through the PR region. If a module is designed to use these wires then the module can’t be placed in positions were the static is routing through the PR region [6], since the wire footprint won’t match.

Another challenge when allowing module relocation is the change in timing for signals, adding a timing footprint. Timing can change depending on where the module is relocated to. Some areas of the FPGA may have longer routing delays because of hidden features, like the configuration logic.

2.3.3 Spartan-6 configuration

Configuration frame

The smallest addressable blocks of logic on the Spartan-6 are the configuration frames. One configuration frame is one column of logic spanning the height of a clock region. The configuration frames for Spartan-6 devices can be divided into three types containing specific data for different segments of the device [31]:

- Type 0: CLB, DSP, input/output interconnect (IOI), clocking.
- Type 1: Block RAM (BRAM).
- Type 2: IOB (I/O blocks).

Configuration is done with three types of operations provided by the configuration logic:

- "00": NOP
- "01": READ
- "02": WRITE

A configuration command is executed when a configuration register is written with data. All configuration registers are listed in the "Spartan-6
configuration user guide" [31]. Configuration data is arranged into two types of packets. Type 1 contain short blocks (length defined by word count) of 16-bit data sections, while type 2 packets can have long blocks (length defined by the two 16-bit word counts) of multiple 16-bit wide data sections. All packets start with a 16-bit header section containing:

<table>
<thead>
<tr>
<th>Header</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>15-13</td>
<td>12-11</td>
<td>10-5</td>
</tr>
<tr>
<td>Type</td>
<td>Operation</td>
<td>Register address</td>
<td>Word count/Not used</td>
</tr>
</tbody>
</table>

- Type (3-bits): "001" for type 1, "010" for type 2.
- Operation (2-bits): NOP, READ, WRITE.
- Register address (6-bits)
- Word count/not used (5-bits): In type 1 this field contains number of 16-bit data words after header. In type 2 this field is not used.

In type 1 packets, the data sections follow the header, while in type 2, the header is followed by two 16-bit words defining the word count and then data sections:

<table>
<thead>
<tr>
<th>Type 2 word count</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>15-0</td>
</tr>
<tr>
<td>Word count 1</td>
<td>word count 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data section[0]</th>
<th>...</th>
<th>Data section[N]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>15-0</td>
<td>15-0</td>
</tr>
<tr>
<td>Data</td>
<td>...</td>
<td>Data</td>
</tr>
</tbody>
</table>

**Spartan-6 bitstream**

To configure Xilinx devices, the user applies a bitstream to one of the configuration interfaces. The bitstream is an encapsulation for the configuration data packets. Spartan-6 bitstreams have the following format [31]:

- Dummy words: To prepare the pipeline of the configuration interface for data.
- Synchronization words: Two 16-bits words used for synchronization (0xAA99 and 0x5566).
- Header.
- Configuration body.
- Header2.
- De-synchronization word: One word (16-bit) signalling the end of the bitstream (0x000D).
The header is used to set up configuration registers required for the reconfiguration. In the configuration body, data is written to the configuration frames of the device. Header2 is also for setting different configuration registers.

2.3.4 Relocation of partial module bitstreams

Module relocation is when the system can move modules between different slots, instead of locking a module to a specific slot within the PR region. The advantages of module relocation is the gained flexibility in module placement. Problems like external fragmentation becomes easier to handle since modules can be moved between different slots. Flexibility also makes the job of figuring out placement and scheduling of modules much easier, since each module fits into more than one slot.

There are different ways to implement module relocation. One way is to create a new bitstream for each slot/island you want to place your module in. If the system provides $n$ modules and $k$ different positions, we would then have to generate and store $n \cdot k$ different bitstreams. For each module only small differences would exist from location to location. For slots with the same underlying FPGA footprint only the header would contain differences [7].

One solution to minimize storage in a system that supports module relocation would be to store position independent bitstream data separate from position dependent. In this way, only the position dependent data would need to be stored for each position. During system reconfiguration, the new module being configured into slot $P$ would take the position dependent configuration data for slot $P$ and combine it with the position independent data to form a full configuration bitstream.

In [14] a filter is designed in hardware to manipulate bitstreams, enabling module relocation without the already mentioned overhead of storing many bitstreams for each module. This done by adding a filter to the configuration process, allowing for the manipulation of addresses in the bitstream during reconfiguration. The addresses are used to define which configuration frames the configuration data should be written to. There are two addresses that need to be changed in order to move a module, the major address for which column of resources (e.g., CLB column or BRAM column) and the minor address for which frame within that column. By changing the major address in the bitstream, a module can be moved horizontally on the device. Equations for address calculations are presented in [14], but many parameters regarding these types calculations are device specific, like the placement of DSP and BRAM columns on the device.

2.3.5 Algorithm for bitstream relocation

The previous section presented module relocation and how it is possible to partition bitstreams into position dependent and independent configuration data. If a module is moved to a new position providing the same
FPGA footprint as the original position, the only change to the bitstream would be the fields that define the module placement position. We can use this information to derive blocks that are 1) position independent and 2) position dependent as sketched in the following algorithm:

1 : Input: \{module \( M_0 \), placement_positions \( P_0 \),
\( \{M_1, P_1\}, \ldots, \{M_k, P_k\} \)
2 : Output: Bitstream_lists \( B_0, B_1, \ldots, B_k \),
3 : \( m = \text{PlaceModuleToCurrent}(P_0) \)
4 : \( b_0 = \text{GeneratePartialBitstream}(m) \)
5 : \( B_0 = b_0 \)
6 : \( \forall P_i, i > 0 \) do
7 : \( m = \text{PlaceModuleToCurrent}(P_i) \)
8 : \( b = \text{GeneratePartialBitstream}(m) \)
9 : \( \text{offset} = 0 \)
10 : \( \text{length} = 0 \)
11 : while \((\text{offset} + \text{length} \leq \text{size}(b))\) do
12 : \( \text{offset} = \text{offset} + \text{length} \)
13 : \( \text{length} = \text{CorrelateBitstream}(b_0, b, \text{offset}) \)
14 : if \( \text{length} > 1 \) // position independent bitstream
15 : \( B_i = B_i \& \text{AddReference}(b_0, \text{offset}, \text{length}) \)
16 : else // position dependent bitstream
17 : \( B_i = B_i \& \text{AddWord}(b, \text{offset}) \)
18 : \}
19 : \}

The algorithm is called with a list of all modules with all corresponding possible placement positions. For the first placement position of each module, we generate a complete partial bitstream \( b_0 \) and store the result in our bitstream repository (lines 3-5). For all other placement positions, we create a partial bitstream in the same way. We then correlate this new bitstream \( b \) with the first bitstream \( b_0 \) starting at the position \( \text{offset} = 0 \) (line 13). The result is the length of the longest match where the two bitstreams are identical. If the matching length is larger than one, we append a reference referring to the original bitstream to the configuration for the current placement position; otherwise, we will add the current word from the bitstream \( b \). This process is repeated until the end of the bitstream. Note that we use the native word size of the configuration state machine which is 32 bit for all Virtex FPGAs and 16 bit Spartan FPGAs from Xilinx.

The result of this algorithm is a full (not relocated) bitstream for each module and a sequence of references to this full bitstream for all other placement positions for each module. Only the position information will not be generated by a reference, but be directly included into the bitstream (line 17).
2.3.6 Bitstream compression for reconfiguration

The amount of logic on FPGA devices is growing for each new generation of devices, and larger devices requires more configuration data making bitstreams larger. Consequently, more data is stored in non-volatile external memory.

Compressing bitstreams before they are loaded into the FPGA configuration memory can both save storage space and improve configuration speeds. Reading bitstreams from non-volatile memory can be slow. Typical external flash memories found on many development boards will not produce enough throughput to directly saturate the configuration port of Spartan and Virtex devices. Compressed bitstreams require less read operations from memory for a given bitstream, making it possible to achieve higher configuration speeds, if fast decompression modules are placed between the memory controller and the configuration port.

There has been a couple of interesting papers published on bitstream compression [10][18]. In most research, a set of algorithms have been tested and modified, mostly variations of Huffman coding, run-length encoding and LZ (Lempel Ziev) encoding. A configuration controller should use as little resources as possible to minimize the logic used on the part of the system which is not contributing to computations. Consequently, for most cases, the decompression module (as a part of the configuration controller) should not implement a decompression algorithm that requires a large amount of resources.

Another important point to focus on when working with a slow memory is to improve worst case compression ratio. In worst case situations, the memory throughput will limit the system. In best case situations the memory will idle and the throughput of the decompression module will be limited by the configuration interface. When selecting and modifying the compression algorithm, focus should be put on having good worst case behaviour, even when this impacts best case results [15]. In other words, for a slow memory it is important that the compression is evenly distributed across the bitstream, minimizing stalling of the output of the decompression module because of limited memory throughput.

In [15] variations of run-length, Huffman and LZSS (based on LZ) coding are implemented and tested. Huffman coding provides excellent compression ratios, but the presented decompression module is slow and requires a lot of logic. Both the modified run-length encoding and LZSS encoding provided small and fast decompression modules, suitable for accelerating reconfiguration. In this thesis, the LZSS decompression module from [15] was used, which is a good compromise on configuration speed, implementation and cost.

2.4 Design flows for partial reconfiguration

There are several different design flows for PR. Both Altera and Xilinx provide devices that support PR [29][3] and also flows for designing
Table 2.2: Comparison of different PR flows for Xilinx devices, based on table presented in [6]

<table>
<thead>
<tr>
<th>feature</th>
<th>Xilinx PlanAhead</th>
<th>OpenPR</th>
<th>GoAhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>supported devices</td>
<td>V4, V5, V6</td>
<td>V4, V5</td>
<td>V4, V5, V6, V7, S6</td>
</tr>
<tr>
<td>floorplanning GUI</td>
<td>yes</td>
<td>uses PlanAhead</td>
<td>yes</td>
</tr>
<tr>
<td>script interface</td>
<td>TCL</td>
<td>yes, GOA</td>
<td>yes</td>
</tr>
<tr>
<td>module relocation</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>static/partial decoupling</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>partial region crossing</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>hierarchical PR</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>component-based design</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>communication method</td>
<td>proxy logic</td>
<td>bus macro</td>
<td>proxy logic, bus macro, direct wire</td>
</tr>
<tr>
<td>reconfiguration style</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>single island</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>multi island</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>slot-based</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>grid-based</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

systems using PR. There are also academic tools that are developed to help designers and researchers to implement PR systems. Since Xilinx has supported PR longer than Altera, most academic tools are targeted for Xilinx devices. In this thesis, the system will be created using GoAhead, an academic tool for PR designs. In this section we will compare GoAhead against the Xilinx PlanAhead PR flow [29] and OpenPR [22]. Most of the academic tools for PR on Xilinx devices utilize the Xilinx Design Language (XDL) [5], which gives designers a human readable format of FPGA resources and netlists. Both GoAhead and OpenPR use XDL for low level netlist operations and human readable versions of the NCD design files created by the Xilinx tools. OpenPR is based on Torq, an Application Programming Interface (API) written in C++ for low level design control over netlist, routing and bitstreams, basically hiding XDL syntax behind an API [23].

### 2.4.1 Xilinx PlanAhead PR flow

The current Xilinx PR flow is based upon “island style” reconfiguration with no module relocation between islands. Modules are implemented as increments of the static design through the use of proxy logic [6]. Proxy logic is placed as anchors in the PR islands to connect signals from the static region to the partial region (see figure 2.7(a)). The routing to the proxy logic is not constrained to one specific wire. This means that the routing to the proxy logic anchors will differ in all islands. Static routing through the PR region will also change between islands. This means that a module have to go through a full routing process for each island it will be placed in. Also, since all modules are created as increments of the static system, the modules are affected by changes to the static part of the system. Each time the static system is changed, the modules will need to be reimplemented. PlanAhead only supports one module for each PR region (“island style”).

20
Figure 2.7: 3 ways of implementing the interface between static- and partial system. Red colour represents the the routing and logic generated during implementation of the partial modules, light blue represents everything added during implementation of the static system. The dark blue wires are the signals creating the interface between static- and partial system.
2.4.2 OpenPR

OpenPR is built to extend the Xilinx PR flow with more advanced features like module relocation. To achieve this, OpenPR uses bus macros and a combination of different types of constraints [22]. For placement, OpenPR uses PROHIBIT and AREA_GROUP constraints for the static and partial regions, to prevent static resources to be placed in the partial region and vice versa. Blocking nets are used to consume all wires to force the router not to use routing resources inside the partial region when routing the static system. To connect signals between the static system and partial region, OpenPR uses Xilinx Bus Macros that are instantiated in HDL code and placed automatically by the tool along the border of the PR region. All this means that OpenPR can provide modules that are independent of the static system, allowing for module relocation, something the Xilinx Flow doesn’t support. One downside with OpenPR is that routing is done by the FPGA Editor from Xilinx instead of the better performing PAR place and route tool [22]. Furthermore, static routing cannot cross the reconfigurable region which might result in weak performance or even unroutable situations.

2.4.3 GoAhead

GoAhead is a tool developed as part of the COSRECOS (Context Switching Reconfigurable Hardware for Communication Systems) [1] project at the University of Oslo. The project aim at making partial reconfiguration of FPGAs more accessible. GoAhead is designed to help designers creating PR systems by hiding low level details of the FPGA architecture. The tool allows user to focus on floor-planning and design decisions, with functions providing control over some low level details. It provides a GUI, shown in Figure 2.8, with a tile view of the FPGA device for floor-planning and macro placement. GoAhead provides a framework for doing more advanced PR than Xilinx PlanAhead allows. It supports module relocation, “grid style” placement and hierarchical PR (i.e. reconfigurable module inside a reconfigurable module).

The way GoAhead implements the interface between static system and partial region is in some ways similar to how it is done in OpenPR. Instead of bus macros, GoAhead uses Direct Wire Binding with connection macros. The connection macros are specially designed pre-wired hard macros which forces signals to connect to specific pins on a SLICE. A blocker is used to force the signals onto specific wires, creating a wire binding of signal $x$ to wire $y$. The blocker is a hard macro generated for a defined region in GoAhead and blocks all signals starting or ending within the defined region. The wires that are needed for the signals within the blocked region must be removed from the blocker by using exclude commands in GoAhead on their start and end points. PROHIBIT constraints are generated together with the blocker to keep logic from being placed within the blocked region by the Xilinx tools. By using connection macros and blockers, GoAhead can achieve PR without the LUT overhead associated with bus macros and proxy logic. A big advantage for complex
communication architectures and small slot sizes.

One drawback with the direct wire approach is that connection macros must be redesigned for each family of FPGAs because of differences in routing and architecture (the same applies for bus macros). In addition, templates for the blocker macros have to be provided for each device family. Another problem with academic tools is that they rely on XDL as a backend. Unfortunately, XDL might not be continued in the future (it is unclear if this is going to happen [2]).

When designing systems, the GoAhead flow is responsible for the hard macros, constraints generation and some XDL operations. The synthesis, mapping, place and route are all done by the Xilinx tools. To simplify implementation, GoAhead can execute any command or script on the command line, allowing for design automation by using command line scripts (batch files in Windows). By running the Xilinx tools also from the command line, the whole implementation process can be automated. This is especially good for reproducibility of the design process when many steps are involved.

2.4.4 GoAhead compared to PlanAhead and OpenPR

In Table 2.2 GoAhead is compared against OpenPR and PlanAhead on what feature they implement in regards to PR. The following list denotes more details on the different features given in the table:

- Hierarchical PR: GoAhead allows for designs using PR modules within PR modules. Using GoAhead, a downscaled MIPS processor with partial reconfigurable custom instructions could be placed as a PR module within a bigger system. It could even be possible to have a common library with custom instructions shared between the processor in the static region and processors configured as PR modules. This could be done by small changes to the bitstream.

- Partial region crossing: GoAhead allows signals from the static system to cross the partial region. This is (as already mentioned) not possible using OpenPR since the partial region is totally blocked. In GoAhead this is accomplished by excluding some wires from the blocker for the partial region when designing the static system, and then blocking the same wires in the partial module. PlanAhead also allows for partial region crossing at the cost of not allowing relocation of PR modules.

- Component-based design: Since GoAhead can read, change and write XDL netlists, it can cut out a module or part of a design and store it in a special data structure. Later these modules can be placed into new designs, fusing the nets between the module and the rest of the design. This can be used to build designs using components from a library of finished modules or to perform timing verification on partial modules by fusing them into the static design [7]. These features are also provided in Torc, which is the base for OpenPR.
Figure 2.8: GoAhead GUI, with the FPGA resources represented as colored tiles.

Figure 2.9: Tool flow for GoAhead and ISE. Figure taken from [6]
2.5 Runtime reconfigurable modules

The previous section presented tool flows for implementing runtime reconfigurable systems on Xilinx FPGA devices. In this section, we will focus on how to use the tools to create more efficient FPGA designs with runtime reconfigurable modules (also called PR modules).

2.5.1 Benefits of using runtime reconfigurable modules

FPGAs are known as reconfigurable devices, but are in many cases used only as ASIC replacements. All modules of the system are placed on the device on start-up and remain until power down, not utilizing the possibilities provided by using the configuration logic to replace modules during execution. Advantages of creating systems by using reconfigurable modules is the possibility of more efficient use of resources, leading to lower resource requirements for the system. This could allow the designer to fit the system onto a smaller device with lower cost and power requirements. Another approach could be to allow each module more area to increase the speed of the computation. This is in particular beneficial for some computations like, for example, large matrix multiplications scale superlinear to the area used for computation [16]. Figure 2.10 illustrates how reconfiguration can save area and/or accelerate execution for modules mutually exclusive in time and space.

One reason why many developers are using FPGAs instead of ASICs is the possibility to make changes to the hardware in-system. This can allow the designer to add features to the system without having to replace any hardware components. This kind of updates are mainly done be replacing the old configuration bitstream with a new one. In a runtime reconfigurable system, features could be added even easier with new reconfigurable modules. With GoAhead, each runtime reconfigurable module is designed separately from the static system. The only thing they share is an communication interface (see Section 2.4.3 on page 22). This makes it possible to create modules independent of the static system and store them as hard wired macros/bitstreams [16]. The modules then form a library that can be used to accelerate different computations. New modules can be designed for the interface and made available to the system through updates without any changes to the static part of the system or to other modules. Changes can also be made to the static system to allow new features as long as the communication interface and floorplanning is kept unchanged. By using this approach we can build systems that can easily be updated to handle new computations without the need to upload a new bitstream for the whole system. Furthermore, since all modules are placed and routed separately from the static system, it is possible to generate all modules in parallel, hence saving time.
Figure 2.10: A) Time and area consumption for a static implementation with module x, y and z. B) Implementation using reconfiguration, only one module is placed on the device at one time. This allows for a smaller device, since area requirements are lower. Modules are changed using reconfiguration (indicated with red in the graph). C) An implementation that utilizes the full size the device used for the static implementation, but only for one module at a time. This allows more resources to spent per module, allowing for speed-up of execution for some problems.
2.6 MIPS architecture

2.6.1 RISC

The MIPS is a Reduced Instruction Set Computing (RISC) instruction set architecture (ISA) and is used in high number of different hardware devices around the world. The opposite of RISC is Complex Instruction Set Computing (CISC). RISC CPUs are designed to have instruction sets consisting of small and simple instructions. The idea behind this is that a simple instruction set will make the whole architecture (and in particular the instruction decoding) faster. A goal in RISC design is to design all instructions so that they can execute within one cycle. To achieve this all instructions in a RISC CPUs have the same size (often four bytes in a 32-bits architecture). This simplifies instruction decoding and ALU operations.

While CISC CPUs can do memory access from many different instructions, RISC CPUs only allow access to memory with two dedicated instructions; load and store. This design choice is tightly coupled with fixed instruction size and simple architecture. Since most instructions in a RISC CPUs are register to register, most RISC CPUs are designed with a larger number of registers than for example the original x86-architecture. This allows for less use of the memory during operations like, for example, passing arguments to a procedure [11].

2.6.2 MIPS overview

A typical MIPS consists of the following functional blocks (the MIPS CPU implemented for this project is shown in Figure 3.3 on page 37):

- Instruction decoder: This block decodes the instructions. The MIPS has a very simple instruction decoder since all instructions are the same length and only three different formats are used (see section 2.6.3).

- Program counter (PC): The program counter outputs the address to the next instruction that should be executed. During ordinary program flow, the program counter increments the instruction address with 4. If a branch or jump occur, one more instruction is performed (this is called delayed branch), then the value provided by the branch or jump instruction is added to the instruction address.

- ALU: This is "the brain" of the CPU. It performs arithmetic and logic operations on the data from the registers and instructions.

- Registers: The MIPS has 31 general purpose registers, were register 0 is used to hold a constant zero. How the registers are used by the compiler is detailed in the "MIPS32® Instruction Set Quick Reference" [20].

- Memory: The MIPS accesses memory through load and store instructions.
To allow the MIPS to run at high clock speeds pipeline registers are often placed between the functional blocks to minimize the delay. The MIPS is designed to use pipelining to improve throughput, many MIPS designs use a 5-stage pipeline. The stages are called Instruction Fetch, Instruction Decode, Execute, Memory access, Register write back.

2.6.3 MIPS instruction set

The MIPS I has three types of instructions:

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>opcode</td>
</tr>
<tr>
<td>I</td>
<td>opcode</td>
</tr>
<tr>
<td>J</td>
<td>opcode</td>
</tr>
</tbody>
</table>

All instructions have an opcode, this is the code used by the instruction decoder to determine how the instruction should be decoded. rs, rt and rd are addresses to the registers in the register file. Immediate is a 16-bit constant that is used for operations with a constant value. Address is used in jump-instructions to move execution to another part of the code.

The different instructions are used for:

- R-type instructions are register to register operations were two operands are taken from the register file pointed to by rs and rt and the result of operation is returned to the register with the address rd. Many R-type instructions share opcode and the operation they perform is determined by the funct-code.

- I-type instructions are used when an operation is done with a constant value. The constant is coded as the immediate field of the instruction, the other operand is from the register pointed to by rs. The result is put back into register pointed to by rt. I-type instructions are also used for branches, then the immediate is added to the current PC value to perform a PC-relative branch.

- J-type instructions are used for providing a new address to the program counter, moving execution to a new code block. Since the address-field is 26-bit long, J-type instructions allow for direct-addressing by changing the lower 28 of the bits in PC. It is 28-bits and not 26-bits since the provided address is addressing words and not bytes. Jumps are not PC-relative like branches.

2.7 Reconfigurable instruction set extension

General purpose processors (GPPs) are designed with an ISA to handle as many different applications as possible. Most applications only use a small subset of all the available instructions in the GPP, and the fastest way to run an application is dedicated hardware. Sometimes small changes to hardware can give huge improvements in execution time. If a calculation
is done many times in a program and it requires many instructions, it could be possible to increase performance by adding dedicated hardware for the calculation. For example, if a compression algorithm has to count the number of ‘1’-bits in a vector, a dedicated instruction can substantially speed up this algorithm. One way of doing this could be to extend the instruction set of a CPU to allow for hardware acceleration of small parts of a program. This is already possible on the softcore CPUs from Xilinx and Altera, the Microblaze and the Nios[16]. With runtime reconfiguration it is possible to allow custom instructions and still have a CPU that is designed for true general purpose use. In other words, it can be used for combining the beauty of a fast RISC machine with the ability of executing more complex instructions.

The idea of extending the instruction set of a processor is old, but FPGA technology has made it interesting in new ways. One interesting architecture is the Dynamic Instruction Set Computer (DISC) [26], a processor based fully on a dynamic instruction set were all instructions were runtime reconfigurable modules. The design was of course limited by the size of FPGA devices of the time, but the paper discusses many interesting points regarding custom instructions that will be discussed in this section.

2.7.1 Custom instructions in hardware

Runtime reconfigurable custom instructions are typically smaller in size than large reconfigurable accelerator modules placed outside the CPU on the system bus. The custom instructions are placed close to the CPU, with small slots/islands. This means that a high number of signals have to enter a small area, making routing very congested. Design flows using bus macros or proxy logic with overhead for each signal crossing between static and partial will not give good results with the high number of signals and small islands/slots. If bus macros were used to connect two 32-bit operands and one 32-bit result, the communication overhead would be somewhere around 200 LUTs [16]. For proxy logic, the overhead would be around 100 LUTs. For comparison, the custom instruction modules presented in Section 4.1.4 on page 63 need in the range of 32 to 34 LUTs. With the direct wire approach of the GoAhead flow, custom instructions can be implemented very efficient in small islands/slots.

Another good reason to use GoAhead for implementation of custom instructions is that the modules can be relocatable. As mentioned in [26], it is very important to allow custom instructions to be placeable in more than one slot, since this allows for a flexible use of the slots and minimizes external fragmentation. It also removes unnecessary reconfiguration calls, because of a higher probability that instructions can stay in the circuit. When allowing custom instructions to be placed and relocated between different slots, the processor needs a look up table to store at which slot a certain instruction is located. This is needed for instruction decoder to know from which custom instruction slot the result should be routed.
2.7.2 Custom instructions in software

Custom instructions require some changes to software and in some cases also the compiler. Custom instructions are easiest to use when they are transparent to the programmer and compiler. This is how it is done in the eMIPS [21], with profiling of the compiled binaries for sections of code that are repeated (ending in a branch instruction). These sections are called basic blocks and are ranked based on how many times they are used. High ranking blocks can be implemented in hardware and used as reconfigurable extensions to the ISA of the eMIPS. The interesting part is how the extensions are added to the software binary. The call to the extension is placed in front of the block of code it is replacing, if the extension is in hardware the call to the extension will be executed and the trailing block of code skipped. If the extension is not in hardware, the instruction to call the extension is treated as a NOP instruction and the trailing block of code is executed. The use of software tools on the binary after compilation allows the programmer to create software the same way as for an ordinary processor. Furthermore, old code could also be accelerated by replacement of basic blocks with extensions.

A more low level approach is to use custom instructions directly in software by assigning them to unused opcodes in the instruction decoder. These opcodes will then correspond to instructions in the binary code. This requires that the compiler is changed to make the instructions usable in Assembly- and/or C-code.

2.7.3 Reconfiguration of custom instructions

For a custom instruction in the binary to be usable, the corresponding hardware module must be present in a custom instruction slot (CI slot). Since reconfiguration time is the biggest overhead when using custom instructions, it is important to minimize the impact of reconfiguration on the system. In the DISC implementation the whole system was stalled when the program reached an instruction that was not in hardware, and waited the time it took to configure the instruction in to hardware [26]. The eMIPS uses (as already mentioned) the software function if the hardware implementation is not configured on the device [21].

One way of minimizing the configuration time is to pre-fetch custom instruction into hardware before they are needed by the processor. This was proposed by [13], where special pre-fetch instructions are used to start a configuration process before the instruction is called. The time it takes to configure a custom instruction depends on the speed of the configuration controller and the size of the bitstream, but even with fast memory and an overclocked ICAP we are in the range of 1000 to 10000 clock cycles (reconfiguration time for our system is presented in Section 4.1.4). This means that an custom instruction must be pre-fetched more than a 1000 instructions before it is used in most cases. If configuration is not finished before the custom instruction is needed the processor must be stalled.

Another approach is to do something similar to the eMIPS, shown
custom_instruction(OP_A, OP_B, RES);

Instruction decode

Available

Execute hardware implementation

Not available

Execute trap handler for reconfiguration

Reconfiguration in progress?

Yes

Execute software implementation

No

Decide on slot

Send commands to configuration controller

Figure 2.11: Flow chart of a possible solution for reconfiguration and handling of custom instructions by using trap handlers and software emulation of CIs.
in Figure 2.11. If the processor wants to execute a custom instruction that is not in hardware, an exception trap will be triggered and the processor will run an trap handler that initialises configuration of the needed custom instruction. The handler will also run a software version of the instruction to allow the execution of the program to continue. As long as the configuration is not finished, the interrupt routine will continue to run the software version of the instruction. This approach removes a lot of the overhead by not stalling the CPU while configuration is in progress. The solution will make the binary code larger since it has to contain software functions for each CI and the trap handler. This approach can of course also be used in combination with pre-fetching too further minimize configuration overhead.
Chapter 3

System implementation

3.1 System overview

The full system implemented in this thesis is shown in Figure 3.1. The system is divided into a static part and a reconfigurable region. The static part contains a MIPS processor and modules for peripherals, memory and configuration. All modules are connected to the processor by a simple bus. In the static region, there is also modules for encoding and decoding of DVI video that is streamed through the reconfigurable region.

The system is divided between different clocks. Most of the system, including the MIPS and the peripherals that are connected to the bus, use a 50 MHz clock derived from the system clock using a PLL primitive, that is also responsible for the 2x and 10x clocks which are needed for the DVI video output. On video input side a pixel clock is generated from the DVI signal. The configuration controller for module relocation (presented in Section 3.6) is designed to use two clocks, the 50 MHz for the part of the controller that is connected to the bus and the 100 MHz system clock for the part that handles configuration.
Figure 3.1: System
3.2 Hardware platform

Our system is implemented on the Atlys development board from Digilent (shown in Figure 3.2). The board is built around a Spartan-6 LX45 FPGA in a 324-pin package, with a 100 MHz system clock. This FPGA provides 6,822 slices (which is 27,288 6-input LUTs) for logic, 116 BRAMs (which is 2,088 Kb of on-chip memory) and 58 DSP blocks (which each can compute 18-bit multiply-accumulate (MAC) operations). For memory, the board provides 1 Gbit of DDR2 memory and 128 Mbit of flash memory. Video is provided through 4 HDMI connectors, two inputs and two outputs. Communication can be done with the COM-port on the computer through a USB-UART bridge on the board. An on-board Ethernet MAC chip can be used for network connection. There is also three 3.5 mm input jacks for line out, line in and mic out coming from an AC97 sound chip.
3.3 System bus

The bus is used to connect all modules and uses a baseline protocol which can be easily connected to many other bus implementations. The bus consists of the following signals:

- Chip select (CS): Active high input signal to module.
- Write enable (WR_en): Active high input signal to module. Instructing the module to read data from the bus.
- Address: Input vector containing the address the master wants read/write to.
- Writedata: 32-bit input from the master.
- Readdata: 32-bit output to the master.

There is no support for bursts or DMA on the system bus, and the only master is the MIPS.

3.3.1 Module testing

All modules connected to the system bus were tested with a command-driven testbench based on a bus functional module. The testbench is written in TCL and VHDL, and use command files to simulate bus transactions and verify that correct data is written and read. This approach allows us to test modules without having to simulate the full system.
3.4 Baseline MIPS

To make the system easy to modify and extend, we decided to build our own CPU core based on the MIPS I instruction set. In the system the MIPS is needed to control all the different modules, and also configure in new modules if needed. The MIPS is also used in this system as a platform to demonstrate instruction set extension using reconfigurable instructions. An overview of the MIPS is shown in Figure 3.3.

The implementation of the MIPS is kept simple to make it easily understandable for any person that is interested in using the system. The problem with soft cores like the MicroBlaze is that they often come with many design files and complex tools. Our MIPS is basically only one HDL-file with 500 lines of VHDL code, and can easily be understood and modified by users, without the need to explore large projects. However, despite the simple implementation style, our baseline MIPS runs still at 50 MHz, delivering 50 M instructions per second (peak), which is more than many micro-controllers. Another advantage of the small size is that it can be combined with the configuration controller presented in Section 3.5 to form a very compact, but smart configuration manager. If the MIPS is still too large for the application, it is easy to reduce memory size and also remove instructions that is not needed. Actually, ISA subsetting (the process of removing unused instructions) can allow for both higher execution speed and smaller size. The only downside is that the MIPS becomes application-specific, but this is not a problem as long as the MIPS
is only going to be used as a advanced state machine for the configuration controller.

### 3.4.1 Implementation

As mentioned in Section 2.6, most RISC CPUs are pipelined designs, often with 5 pipeline stages that compute one instruction per cycle. Pipelining requires that we can handle hazards in different stages of the pipeline, by adding corresponding control logic. Our MIPS is not pipelined, but it can still execute one instruction per cycle. This basically reduces the achievable clock frequency as compared to other pipelined soft CPU implementations. However, pipelined CPUs are typically restricted on FPGAs and register forwarding cannot be used in the same way as in an ASIC implementation. This is due to the extra multiplexer that implemented the forwarding [32]. Not using pipelining requires that instruction fetch, decoding, execution (ALU) and memory write-back has to happen within one clock cycle. Consequently, the MIPS has long combinatorial paths between flip-flops, reducing the maximum clock frequency we can achieve with the design.

One of the tricks that was applied to allow the MIPS to execute one instruction per cycle was sending the instruction address to the instruction memory just before the setup time for the next clock edge, making the instruction word available at the beginning the following clock cycle. This is done to avoid having to wait one clock cycle for the instruction memory to output the instruction word. The "trick" is illustrated in Figure 3.4, with nextPC being passed on to the instruction memory instead of PC. This was used because reading the internal instruction memory cannot be performed asynchronous and it takes at least one clock cycle delay to read data from a BRAM primitive. Some may think that this impacts timing since the address nextPC has to meet setup requirements on the input of the instruction memory after a long combinatorial path, but timing analysis in ISE show that other paths impact the timing more.

The MIPS has a delayed branch to avoid hazards in pipelined implementations. In our case, the delayed branch is pure overhead, since extra logic and flip-flops are required to handle the storing of the branch address while the delay slot is executing.

The critical path that impacts the timing and worst case performance the most is the signed and unsigned multiplication instructions. Actually, the slowest instruction would probably be the division instruction, but since implementations of division are resource expensive and seldom used we have omitted the \texttt{div} instruction from the MIPS. If division is needed, it could be added as a software function or multi-cycle instruction. Note that the compiler supports code-generation with or without \texttt{div} instruction support. The multiplication instruction is implemented in DSP-blocks if this option is enable in synthesis tool, but even with the use of DSPs the multiplication is too slow in relation to other instructions considering single cycle execution. The solution is shown in Figure 3.5. Both the multiplication and division instructions operate on their own storage registers called \texttt{HI} and \texttt{LO}. By constraining the combinatorial path between
Figure 3.4: An overview of the Program Counter. The address nextPC is passed on to the instruction memory instead of PC to allow for execution of one instruction per clock cycle. The register for delayed branch is pure overhead in our case, because of the lack of pipelining in the design.
the output of the instruction memory and the inputs to HI and LO with multi-cycle constraints in ISE, the path is allowed two or more cycles to execute. To allow the path multi-cycle operation in hardware, a stall signal is used to stall the MIPS while multi-cycle execution is happening. In other words, the mul instruction will now take two clock cycles, but the clock frequency can be higher resulting in a higher average performance.

The stall signal can also be triggered outside the MIPS. This is used during loads to make sure that the loaded value is ready to be read from the register file by the next instruction before execution continues. This was added to the design to make the MIPS simulate cycle accurate with a MIPS simulator, but it may not be needed since closer inspection of the binary produced by the compiler shows that No-Operation instructions (NOPs) are added after each load.

The rest of the MIPS is a straight forward implementation of the different instructions and their formats (see Section 2.6.3).

### 3.4.2 Software and testing

Code is compiled for the MIPS using a GCC cross-compiler. The resulting binary is processed with a script to produce a HDL-file that represents the MIPS instruction memory. To verify correct operation of the MIPS the same binary can be simulated in GXemul (an emulator for different computer architectures) to produce a reference file with contents of all registers for each cycle. This file is used in a slightly modified version of the testbench used in the course INF5430 to test MIPS implementations. The testbench is implemented in TCL and VHDL, and compares the simulation in Modelsim against the file produced by GXemul. If one or more registers do not match, the simulation will halt and the difference is shown in the terminal.
3.5 Configuration controller

The configuration controller is a key component of the system since it handles the runtime reconfiguration of modules. To have fast reconfiguration is critical in many PR applications, because it affects the how much speed the system can gain by using reconfigurable custom instructions or dedicated accelerators. Slow reconfiguration speeds could eliminate the gains of hardware acceleration. This is one of the bigger challenges within PR.

This means that the controller should perform Direct Memory Access (DMA) between the memory holding the module bitstreams and the Internal Configuration Access Port (ICAP) primitive. This limits the load on the CPU and the system bus while providing high configuration throughput at the same time. Another important consideration is storage of the module bitstreams. To allow high configuration data throughput the memory holding module bitstreams should be fast, most optimally providing one configuration word (16-bit on Spartan-6) each clock cycle, which is the maximum word size of the used Spartan-6 FPGA. The memory should also provide enough space to hold all module bitstreams for the system. One solution proposed by researchers is to store bitstreams in BRAM [4]. This solution provides high data rates and can provide one word each clock cycle, but uses BRAMs that can be a scarce resource in many systems. Another solution is to put all bitstreams in DDR-memory at start up and use DMA between the DDR memory and ICAP [25]. This provides high speed memory access and large amount of storage, but requires both a DMA-engine and a DDR memory-controller. Furthermore, it uses some of the DDR memory bandwith may be needed by other parts of the system (e.g. frame buffering, processor memory). Consequently, the configuration process might impact the system by adding extra latency to DMA transfers.

Our configuration controller is designed to provide the needed reconfiguration data as fast as possible from the flash memory to the ICAP primitive. To provide fast configuration, the controller is connected to the MIPS CPU over the system bus. Reconfiguration is started by sending a DMA commando. Then, the reconfiguration process is done directly between flash memory and ICAP, without further involvement of the CPU and the system bus. This DMA approach is important because, as mentioned, too much CPU involvement during reconfiguration would lower the throughput of configuration data and consume both the system bus and the CPU. And by using the typically unused bandwidth of the flash (often only used at start-up to load configurations or data into the FPGA), interference on the DDR memory is removed.

This approach was chosen to make the configuration controller easily portable to other devices and boards. Many board designs provide SPI flash memory and the configuration controller should work on other boards with only small changes to the flash reader module. For example, the Spartan-6 Atlys board from Digilent, the Spartan-6 FPGA LX9 Board from Avnet, the Kintex-7 KC705, and the Zynq ZedBoard, include all a Numonyx N25Q12 or compatible QIO-SPI flash memory. In addition to the
wide availability, flash does not need to be preloaded at system start and board vendors commonly provide solutions for writing memory images to the device.

The block view of the configuration controller is shown in Figure 3.6. It consists of the following interconnected components:

- Flash controller.
- SPI flash reader.
- Decompression module.
- UART receiver.
- 8-bit to 16-bit buffer.
- ICAP primitive.

### 3.5.1 Flash controller

This module controls reading configuration data from the flash memory. It has 3 registers accessible on the system bus:

- (base + 4), Address register (24-bit): The read address sent to the flash memory.
- (base + 8), Size register (24-bit): Number bytes to read from flash memory.
- (base + 0), Control register (3-bit): Bit 0 is set high to start configuration from flash memory. Bit 3 is read-only and signals that configuration is finished. Bit 2 is reserved for later use.

The operation of the controller is shown in Figure 3.7.

### 3.5.2 SPI flash reader

The SPI flash reader module is designed to read data from the 128Mbit Numonyx N25Q12 flash memory located on the Atlys board. The N25Q12 transfers data over SPI at maximum clock frequency of 108 MHz. It has the option to use a quad-SPI mode called QIO-SPI. In this mode the read speed from the memory is 50 MB/s with a memory clock frequency of 100 MHz (the system clock on the Atlys board). This is the fastest transfer mode on the N25Q12. We are interested in saturating the configuration port with configuration data to allow the fastest possible reconfiguration time, since this is one of the biggest challenges within reconfiguration. Because of this the flash reader is designed to read from the flash at full rate using QIO-SPI at speeds up to 108 MHz.

At start-up (or reset) of the system, the flash reader module goes into configuration mode. To use QIO-SPI mode on the N25Q12 two configuration registers needs to be written:
Figure 3.6: Block view of configuration controller.
Figure 3.7: Flash controller ASM diagram. The signal names correspond to the ones in Figure 3.6.
• Volatile Enhanced Configuration Register (VECR). Written to enable QIO-SPI.

• Volatile Configuration Register (VCR). Written to set dummy cycles.

When the configuration mode is finished, a ready flag is set to signal that the module is ready for QIO-SPI read instructions. In read mode, the flash reader waits for the request signal to be pulled high. When this happens the flash reader sends a fast read instruction and address to the flash and continues to receive data until request goes low. The flash memory produces 4-bit data each clock cycle until the fast read instruction is terminated by the flash reader. To temporary halt the reading from the flash memory without ending transfer, a input signal for stalling (STALL) is provided. When a stall signal is received the flash reader holds the clock signal to the flash memory low to halt transfer. For safely stalling the flash clock, an I/O primitive (ODDR2), originally intended for double-data transfers was used to prevent glitches while achieving a low skew on the clock pin. All signals from and to the FPGA and the external flash were constrained with flip-flops in the input/output blocks (IOBs), meaning that signals are routed through special flops located close to the output and input pads of the device. This hides any timing issues from long signal paths within the FPGA.

3.5.3 Decompression module

At 100 MHz system clock, the flash memory can provide 50 MB/s. Adding compression on the bitstream requires less data to be read from the memory, but requires a module for decompression before data is sent to the ICAP. The configuration bitstreams are compressed with Lempel–Ziv–Storer–Szymanski (LZSS) lossless compression before they are written to the flash memory. A decompression module is placed in the configuration controller between the memory and ICAP to decompress the configuration bitstreams. In this thesis a decompression module described in [15] was integrated into the configuration controller.

Under ideal conditions, the decompression module outputs 8-bit every clock cycle, pushing configuration throughput with 100 MHz system clock from 50 MB/s without compression close to 100 MB/s with compression (see results). Note that the 100 MB/s is the maximum specified configuration speed of Spartan-6.

3.5.4 UART receiver

The UART receiver is added to the configuration controller as an option to allow reconfiguration with bitstreams not stored in the flash memory. It operates at 115200 baud and is not connected to the decompression module. Alternatively, 1 Mbit is supported for systems providing a corresponding USB to RS232 bridge. When a bitstream is sent over the UART, this module takes control over the ICAP and starts writing the received configuration data to the ICAP data port. This process will be
much slower than configuration from flash memory and is mostly added for portability to systems without flash memory and also for testing of new modules without the need to upload bitstreams to the flash memory. The UART receiver was provided as an IP-core and integrated into the configuration controller.

3.5.5 ICAP

An ICAP primitive is instantiated inside the configuration controller to allow us to write configuration data to the device. The ICAP is connected to the decompression module and also the UART receiver. There are 2 ways to control the reading and writing of data to and from the ICAP. Either clock is left toggling and clock enable is used to control throughput, or clock enable is kept high and the clock signal is controlled to achieve wanted throughput. In our design we use the last option.

3.6 Configuration controller for module relocation

The configuration controller presented in section 3.5 lacked some features that are needed for experimentation with bitstreams and reconfiguration. There was no way for the MIPS to write any data to the ICAP port. This is essentially if the system should support module relocation by changing parts of the module bitstream during reconfiguration. At the same time the configuration should still be free running from the rest of the system, not stalling the MIPS.

3.6.1 Implementation

To allow for bitstreams comprised of position dependent and independent data, our configuration controller has been modified by extending it with FIFOs for the address and size registers. This extension makes it possible for the configuration controller to chain together multiple reads from different parts of the flash memory. The CPU is in charge to control reconfiguration and to fill the size and address FIFOs with data. In other words, the configuration process is then a sequence of different DMA transfers. In addition to DMA transfers, the value of ‘0’ for the size FIFO was used to send a data word (stored in the start address FIFO) directly to ICAP, without initializing a DMA transfer with the flash memory. With this command, the CPU can for example compute the address values defining the module position and include the result into the configuration data without further interaction with the configuration controller or the need to store all different positions in flash memory. To give feedback from the configuration process to the rest of the system, a register can be set with a user chosen value by writing only ‘1’s to the size FIFO and the value to the address FIFO. By reading this user register, the CPU can for example determine if the configuration process of a instruction has finished or how far the process has come. Figure 3.8 shows how the flash controller inside the configuration controller is changed to allow for module relocation.
Figure 3.8: This figure show how the flash controller is changed to allow for module relocation. It replaces the flash controller in Figure 3.6 to form a complete configuration controller capable of module relocation.
Another design change is that the configuration controller is designed to operate on a higher clock speed than the rest of the system. This is done by separating the flash memory controller, decompression module and ICAP from the system bus interface and registers. This allows for fast reconfiguration even when the rest of the system has to meet clock timing for slower parts of the system. There are only four signals crossing between the system clock and the reconfiguration clock: Address and size from the FIFOs and handshaking signals for the initialization of a new command. There are no longer a control register to start reconfiguration as there were in the first iteration of the controller. Reconfiguration will start if there exists a tuple of size and address in the FIFOs and the previous command has finished. The controller operates on 16-bit words. If words are written directly from the MIPS using the size 0 command, each 16-bit word requires a new command to be issued. In addition to the user register there is also a status register that contains information about the FIFOs, if they are empty or almost full.

3.7 Custom instructions

The MIPS was extended with custom instructions as shown in Figure 3.9. The custom instructions were added as extensions to the MIPS ISA and acts the same way as ordinary instructions taking one or two 32-bits input operands and computing one 32-bit output. The custom instructions were added to the system to demonstrate how moving small (but frequently used) functions from software into hardware can substantially speed up the execution time of an application. The custom instructions are also tightly coupled to the CPU, acting exactly in the same way as the rest of the baseline instructions. Larger, more decoupled modules should be placed as runtime reconfigurable accelerator modules in a larger PR region with a different type of communication architecture.

The runtime reconfigurable custom instructions are implemented in GoAhead using one-dimensional slots and direct wire communication. GoAhead is chosen for the implementation of the custom instructions since it provides zero overhead communication and good support for slot style implementations with more than one module inside one PR region.

3.7.1 Hardware implementation

The slots are placed on the west side of the MIPS CPU with operands (OP_A and OP_B) going in west direction on double wires and the result (RES_X) returning east on quad wires. A conceptual drawing of the hardware implementation with two slots is shown in Figure 3.10. The figure only shows one CLB row of the full implementation, each colored line represents 4-bits of the full signal. To implement the full structure needed to support 32-bit operands, 8 identical rows of CLBs are required. All macros are specially designed connection macros that are placed as hard macros in GoAhead.
Figure 3.9: The MIPS ALU extended with custom instructions (CIs). The two slots for CIs acts as an extension of the ALU, allowing for two more instructions that can be reconfigured at any time.

Figure 3.10: Drawing of hard macro placement and routing for two slots west of CPU. White boxes are connection macros placed in CLB columns. Blue and red rectangles are DSP and BRAMs. Curved arrows indicate routing. We used double and quad wires which route two respectively four columns far. Brackets at the top of the figure indicates if the enclosed macros are placed in static or partial part of the implementation.
Figure 3.11: Implementation of static system. Connection macros (yellow squares) placed inside the partial region. Picture taken in GoAhead GUI.

To allow for multiple one-dimensional slots the routing for signals needs to be regular, allowing for the same interface in all slots. With this, modules can connect to identical interfaces even if placed at different positions. This is achieved by using wires in an alternating fashion (also called nested wires), with double wires in westwards direction connecting the operands and quad wires in east direction for the result vectors. The use of alternating wires enhances the wire density for connecting many wires into a small region.

There are certain consideration that needs to be taken into account when implementing PR regions in the design. The PR region for the custom instructions do not cover a full configuration frame in height. This means that logic above and below will be affected by reconfiguration. One problem can arise if the CLBs contain SLICEMs that implements distributed memory/RAM, then the reconfiguration could change the contents stored in memory. In our case this means that the register file of the MIPS should not be placed in the same configuration frames as the PR region for the custom instructions. This means that the register file has to be constrained left or right beside the custom instructions, but never above or below.

**Static implementation**

The implementation flow for the static system with Xilinx tools and GoAhead is as follows:

1. Create PR region in GoAhead. For two slots with 32-bit operands, the area must be 4 CLB columns wide and at least 8 rows high. In order to accommodate all interface signals when using double wires.

2. Use the macro placer in GoAhead to place connection macros inside the PR region (see figure 3.11 and 3.10). Note that the connection macros basically substitute the partial modules.
3. Block all routing inside the PR region, except for all routing used for operands and result vectors. Then export the blocker to XDL.

4. Instantiate the connection macros in the system HDL-code and connect both operands and result vector. The tool provides code for this. Update the constraint file for the design (UCF-file) with placement constraints for the PR region generated by GoAhead.

5. Run synthesis and map in Xilinx ISE

6. Append blocker to mapped design using GoAhead.

7. Run Xilinx place and route.

8. Remove blocker net from routed design.

9. Run bitgen to create bitstream file of static design.

Figure 3.12 shows the static implementation of the PR region for the custom instructions.

Partial implementation

The partial implementation follows many of the same steps as the static system:

1. In GoAhead: Define same PR region as in static.

2. Use the macro placer in GoAhead to place connection macros in the static region (see figure 3.10 and 3.13). Note that in this case the connection macros are substituting the static system.

3. Block all routing around the PR region by creating a fence with a thickness of at least 5 rows/columns in GoAhead, which is the longest path a single wire on Spartan-6 can route. We exclude all
Figure 3.13: Implementation of partial module. Connection macros (yellow squares) placed in the static region. Picture taken in GoAhead GUI.

1. Instantiate the connection macros in the system HDL-code and connect operands and result vector. Update the constraint file for the design (UCF-file) with placement constraints for the static region generated by Goahead.

2. Run synthesis and map in Xilinx ISE

3. Append blocker to mapped design using GoAhead.

4. Run Xilinx place and route.

5. Remove blocker net from routed design.

6. Cut out the module from the design using GoAhead.

7. Place and fuse the module into the static design to create a full static design with the PR module in one of the slots.

8. Run bitgen with -g ActiveReconfig:Yes and the option -r to create a differential bitstream from the differences between the static design with the PR module and the static design without it. This will produce a partial bitstream for the PR module/custom instruction.

One important thing with PR module implementation is to remember to constrain the design properly. The blocker fence will generate prohibit statements for the area covered by the fence, but not the rest of the FPGA. To fully limit the placement of partial module logic to certain slots within the PR region, AREA_GROUP constraints must be used to confine a module within an area.

Figure 3.14 shows the implementation of a custom instruction in the FPGA Editor.
3.7.2 Software implementation

Changes to the compiler

To be able to use the reconfigurable custom instructions in software, we have to make some small changes to the GCC cross compiler for the MIPS. The custom instructions should not use the instruction encoding of any of the instructions already defined in the MIPS I instruction set. Inside the file, defining the MIPS instructions for the compiler (`../binutils-2.23/opcodes/mips-opc.c`), we found a range of instructions called UDIs (user defined instructions), with instruction word encoding from 0x70000010 to 0x7000001f. The UDIs are SPECIAL2-type (very similar to R-TYPE instructions in format) instructions sharing opcode, but with different funct-codes (see Section 2.6.3). These instructions were modified to fit our usage by changing the format to the R-TYPE format. This was done by replacing the format of the UDI with a the format of the R-TYPE XOR instruction:

```
{"xor", "d,v,t",0x00000026, 0xfc0007ff, WR_d|RD_s|RD_t,0,I1 },
```

Modified UDI instruction renamed to CUST:

```
//}}"udi0", "s,t,d,+1", 0x70000010, 0xfc00003f, WR_d|RD_s|RD_t, 0, I33 },
{"cust", "d,v,t", 0x70000010, 0xfc0007ff, WR_d|RD_s|RD_t, 0, I1 },
```

With the changes to (.../binutils-2.23/opcodes/mips-opc.c) stored, we recompiled our cross-compiler with the new changes. The UDI range of instructions is quite large and therefore designers can add many instructions to their system. In total, there are 16 individual user instructions possible. Note that multiple UDIs might be mapped to the same physical CI module. This could, for example, be used to control operation of a custom instruction (e.g. addition versus subtraction).

Using custom instructions in software

Inline assembly is used to call a custom instruction in C-code. A call to our implemented CUST instruction would look like:
Figure 3.15: Selecting a CI slot is done by decoding the funct-field of the custom instruction as an input to a table, which stores the select signal for the slot multiplexer and consequently, for a placed custom instruction.

```asm
__asm__ ("nop\n	" cust %0, %1, %2\n	":="r" (z) :
":="r" (x), "r" (y));
```

Here x and y are the input operands and z is the result.

### 3.7.3 Reconfigurable decoder for custom instructions

To make placement of custom instructions fully flexible, by allowing to place any custom instruction in any CI slot, we need a reconfigurable instruction decoder within the MIPS CPU. Figure 3.15 shows how this is implemented for this project. Instead of using the opcode and funct-code of the custom instruction to directly decide on the result slot, as we would for a ordinary R-TYPE instruction, we instead use the funct-code as a look-up in a table where the correct select signal is stored for each CI. This select signal is then used instead of the funct-code to select the correct slot for the
instruction. The entries in the table are changed by the MIPS itself to allow custom instructions to be put into any slot.

3.8 Partial reconfigurable region

The top part of the FPGA is dedicated to a PR region meant for larger modules placed in a slot and/or grid fashion. This is shown as the PR region in Figure 3.1 on page 34. A streaming interface will cross the region to provide the possibility of feeding modules placed in the region with video, sound or any other data that fits the interface.

3.8.1 Implementation

GoAhead is used to implement the large PR region. For most of the implementation scripts have been written for simplifying the adjustment of the size of the region, by simplifying access to variables which define placement of macros and the size of the blocker for the static implementation.

The streaming interface was defined such that each slot is connected with a 32-bit vector. Double wires were used in an alternating fashion to route the streaming interface across the region. This means that a module has to span two columns (meaning two switch matrices) to connect all 32 wires needed for the streaming interface, hence, making slots two CLBs wide in horizontal direction. In vertical direction, the slots span the height of one clock region, the same as one configuration frame.

The implementation in GoAhead and ISE follows mostly the same steps as the implementation of custom instructions in Section 3.7.

Static implementation

During floorplanning, an area in the upper part of the device was dedicated to the PR region. 8 connection macros were placed inside PR region as shown in Figure 3.16. Each connection macro connects 4 input signals and 4 output signals from the surrounding static system. To make it a streaming interface, the input and output signals are actually connected to the same equivalent wire. The whole region was blocked using GoAhead, except for the nested double wires which had all their start and end points excluded from blocking. Figure 3.17 shows the placed connection macros in Xilinx FPGA Editor.

Partial implementation

The modules for the PR region are implemented by defining a region with the same resource and wire footprint as the one used in the static implementation, then placing 8 connection macros on both sides of the region to define the streaming interface. It is important to understand that the location of the region for implementation of partial modules doesn’t have to be the same as in the static implementation, but the interface
Figure 3.16: Two connection macros are placed in the static region to define the streaming interface using double wires. This figure shows a simplified view of how the double wires are used to create a streaming interface across the PR region. Only the top and the bottom row of the streaming interface is shown. The two set of double wires are coloured blue and red. They are connected to connection macros in the same colour.
Figure 3.17: Screenshot from the FPGA Editor showing the connection macros and the connected double wires within the PR region.

defined by the connection macros and allocation of wires must be exactly the same. Implemented partial modules can later easily be cut out and moved using GoAhead to produce bitstreams for the module at new positions. Figure 3.18 shows a routed partial module in the FPGA Editor.
3.9 Video input and output

Video is streamed through the PR region as shown in Figure 3.1. This allows us to create modules that can do video processing, providing an easy way to illustrate how PR modules are reconfigured in and out of the system, and also their processing capabilities. Figure 3.19 shows a picture taken of video input/output with video overlay modules running on the board.

Video sources and monitors are connected to the HDMI-connectors on the board. The DVI video is decoded and encoded using IP-cores from Xilinx. The Xilinx DVI decoder outputs 24-bit of RGB data per pixel, and also synchronization signals in the form of VSYNC, HSYNC and ACTIVE. Data from the decoder arrives with the pixel clock, which is derived from the DVI input signal. The pixel clock changes depending on the resolution and refresh frequency of the video signal. We use the pixel clock from the DVI decoder also to clock the DVI encoder, in this way we achieve the same resolution and refresh rate on the output as on the input.

3.9.1 Video buffering

Since the pixel clock most likely is not operating at the same frequency as the system clock, some kind of buffering of pixel data is needed to separate the two clock domains. The most common approach is frame buffering in external memory, but this requires memory controllers and often also some extra buffering to hide memory latency. Another approach is line buffering, where only one line of the video is buffered and consumed at a time. This can be combined with frame buffering, but can also be used alone to separate the pixel clock from the system clock. With only line buffering, the system must be able to empty the line buffer as fast as it is filled. This means that the system clock can not be to much slower than the input and output pixel clock. In this project we use an IP-core that allows us to have a slightly slower system clock compared to the pixel clock. Each
Figure 3.19: Picture of video input/output with video overlay modules running on the board. In this picture we use three different modules, one skin color detection module, two Pacman modules and a module for displaying hex-values.

A line of video has a period of active pixel data, which contains the RGB data for each pixel, but it also has two periods per line which doesn’t contain any pixel data, called front- and back porch. For example, the video resolution 800x600 with refresh rate of 40 Hz has 800 active pixels per line, but the complete line is 1056 pixels. With the already mentioned IP-core, we can use the porches to allow us more time to empty the line buffer between lines. In front of the encoder we use a new instantiation of the same IP-core to allow us longer time to fill the output line buffer. This means that we use the porches of the video signal to allow us more time to process data in the system clock domain. This allows us to use line buffering even when the system clock is slower than the pixel clock. Of course, how much slower depends on how long the porches are, which varies between video modes.

3.10 Challenges during implementation

3.10.1 Custom instructions

In the implemented system the slots for custom instructions are located to the west (left) of the MIPS CPU. During implementation we firstly tried to place the CI slots to the east of the MIPS. This caused problems, we experienced situations where the router couldn’t route the quad wires in the design. By examining the switch matrices, we found that the wire allocation we wanted was not possible for quad wires in that routing direction. For the most part, the routing resources are homogeneous and easy to use for PR, but for quad wires in this particular direction the routing was not as predicted. The solution was to move the slots to the other side of the MIPS, since quad wires going in the opposite direction could be routed as expected.
3.10.2 DDC2 module for plug-and-play video

When a monitor is connected to a graphics adapter, a transfer will be initiated by the graphics adapter to read plug-and-play information regarding supported display modes from the monitor. This communication is performed over the Display Data Channel (DDC), which is a collection of protocols for communication between monitor and graphics adapter. The most common protocol is DDC2B, which uses I2C to read a 128-byte ROM inside the monitor. On the I2C-bus, the graphics adapter is the master and the monitor is the slave.

During implementation of the system we designed a module that could emulate the EDID-ROM inside a monitor. This allows us to connect a video source to the development board without having to connect a monitor. The module was designed as an I2C-slave containing 128 byte of display information taken from a real monitor. We tested the module by connecting a Panasonic Lumix compact camera to the board, and the camera started to send video over DVI as if it was connected to a monitor. However, when connecting the board to the DVI output of a computer, the computer does not recognize the board as a monitor. Using an oscilloscope we examined the communication on the I2C-bus between the board and the graphics adapter, but the communication looked very much the same as what we had observed using a real monitor. Because of limited time, we were not able to find the design error.
Chapter 4

Test cases and results

In this chapter we present two test cases of the system. In the first case we test reconfiguration of custom instructions and in the second case we focus on reconfiguration and relocation of reconfigurable modules in the PR region.

4.1 Custom instructions

4.1.1 Introduction

In this test, we will use the configuration controller to carry out the reconfiguration of custom instructions on our MIPS CPU. Two custom instructions have been implemented for this test.

4.1.2 Setup

To test the configuration controller presented in section 3.5, we have implemented custom instructions using the methods described in section 3.7. For the MIPS CPU a small program is written to allow us to read the switches on the Atlys board and to control the reconfiguration of a custom instruction from flash memory. The test code is listed in listing 4.1 on page 61.

Listing 4.1: Test code for configuration of custom instructions.

```c
while (1) {
    sw = *gpio;  // read from switches
    i = 1000;
    if (sw & 0x00000001) {
        OP_A = 0x00000080;
        OP_B = 0x00000111;
    } else if (sw & 0x00000002) {
        OP_A = 0x00000610;
        OP_B = 0x00000111;
    } else {
```

61
The testing and reconfiguration is controlled by reading DIP-switches on the Atlys board using a GPIO-module (line 2 in the test code). Reconfiguration is started by setting switch 8 on the board to '1' (line 22). In order to start the configuration controller, size, address and start command is written to the controller (line 24, 25, 26). To wait for the reconfiguration to finish we use a blocking while-loop that waits for the reconfiguration finished flag (line 27) from the configuration controller.

Configuration time was measured in clock cycles with a counter inside the UART module. It is started and stopped by a control register that is written by the test program (line 39 and 44).

Both custom instructions were compared against a software implementations of the functions running on the standard ISA of the MIPS CPU. The software functions used can be seen in Listing 4.2. To avoid that the functions use to many load/store instructions, all the loop variables have been prefixed with register to make the compiler use the registers as much as possible. Because of poor register usage by the compiler, the software implementation of count ones used double the amount of clock cycles without the register keyword.

Listing 4.2: Software functions.

```
unsigned int CountOnes(unsigned int A) {
    register int i;
    register unsigned count = 0;
    ...}
```
Table 4.1: Configuration controller, resource requirements.

<table>
<thead>
<tr>
<th>Module</th>
<th>Nr. of LUTs</th>
<th>Nr. of slices</th>
<th>Nr. of LUTRAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI flash reader</td>
<td>55</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>UART receiver</td>
<td>255</td>
<td>90</td>
<td>0</td>
</tr>
<tr>
<td>Decompression</td>
<td>63</td>
<td>30</td>
<td>16</td>
</tr>
<tr>
<td>Flash controller</td>
<td>145</td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td><strong>Configuration controller</strong></td>
<td><strong>518</strong></td>
<td><strong>189</strong></td>
<td><strong>16</strong></td>
</tr>
</tbody>
</table>

```c
5  for (i=0; i<32; i++)
6    if ((A & (0x00000001 << i)) > 0)
7        count ++;
8  return count;
9 }
10 unsigned int CCITT_CRC(unsigned int CRCreg, unsigned int input)
11 {
12    register int i;
13    register unsigned int tmp;
14    register unsigned int LocalCRCreg;
15    LocalCRCreg = CRCreg;
16    for(i=0; i<16; i++)
17    {
18        tmp = (LocalCRCreg & 0x80008000) >> 15;
19        tmp = tmp | ((LocalCRCreg & 0x80008000) >> 10);
20        tmp = tmp | ((LocalCRCreg & 0x80008000) >> 3);
21        LocalCRCreg = ((LocalCRCreg & 0x7FF7FF) << 1);
22        LocalCRCreg = LocalCRCreg | ((input & (0x00010001 << i)) >> i);
23        LocalCRCreg = LocalCRCreg ^ tmp;
24    }
25    return LocalCRCreg;
26 }
```

4.1.3 Configuration controller

The implementation cost of the configuration controller is shown in Table 4.1. The clock speed of this test is limited by the MIPS CPU to 25 MHz, but the configuration controller can run at a 100 MHz. This will not affect the test since all measurements are based on number of clock cycles.

4.1.4 Custom instructions

Two custom instructions were implemented for this test:

- 32-bit CRC: Takes two 32-bit operands and computes a CRC.
- Count ones: Counts the number of ones in a 32-bit vector.

The CRC instruction is created by taking the function written in C-code and run it through a high-level synthesis tool (Catapult C from Mentor
Table 4.2: Custom instruction, resource requirements.

<table>
<thead>
<tr>
<th>Module</th>
<th>Nr. of LUTs</th>
<th>Nr. of slices</th>
<th>Slot usage (LUTs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC</td>
<td>34</td>
<td>11</td>
<td>27 %</td>
</tr>
<tr>
<td>Count ones</td>
<td>32</td>
<td>12</td>
<td>25 %</td>
</tr>
</tbody>
</table>

Table 4.3: Custom instruction, bitstream size.

<table>
<thead>
<tr>
<th>Module</th>
<th>Uncompressed size</th>
<th>Compressed size</th>
<th>Compression ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC</td>
<td>7,966 bytes</td>
<td>1,530 bytes</td>
<td>5.2</td>
</tr>
<tr>
<td>Count ones</td>
<td>7,820 bytes</td>
<td>1,548 bytes</td>
<td>5</td>
</tr>
</tbody>
</table>

Graphics) to produce a VHDL-description. This description is directly used as custom instruction in our system. The "count ones" instruction was provided as an IP core written in VHDL to fit the underlying FPGA architecture efficiently, without using multiple clock cycles for execution. Table 4.2 shows the implementation costs for both custom instructions. The slot size was two columns of 8 CLBs, with a total of 128 LUTs and 32 slices. Since the "count ones" instruction only needs one input operand and logic requirements are below 64 LUTs, the instruction could be implemented in only one column of 8 CLBs.

Both instructions had their bitstreams compressed using LZSS so that they could be used with our decompression module. The results of the compression is listed in Table 4.3.

4.1.5 Reconfiguration results

The results from the reconfiguration test is presented in Table 4.4. All clock cycle measurements are done with the counter in the UART-module and reflects the number of clock cycles from the counter is started until it is stopped.

If we look at the number of cycles required to run a full reconfiguration of one custom instruction compared to the bitstream size, we get:

\[
\text{For count ones: } \frac{7,828 \text{ bytes}}{8,077 \text{ cycles}} \approx 0.97 \text{ bytes/cycle}
\]

Table 4.4: Reconfiguration results. Bitstream size is the number of bytes that has to be sent to ICAP. Configuration clock cycles is the number of clock cycles the whole reconfiguration process requires. Software requirement is the number clock cycles a software implementation of the custom instruction requires on the MIPS CPU.

<table>
<thead>
<tr>
<th>Module</th>
<th>Bitstream size</th>
<th>Config. clock cycles</th>
<th>Software requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC</td>
<td>7,976 bytes</td>
<td>8,239 cycles</td>
<td>568 cycles</td>
</tr>
<tr>
<td>Count ones</td>
<td>7,828 bytes</td>
<td>8,077 cycles</td>
<td>461 cycles</td>
</tr>
</tbody>
</table>
For CRC: $\frac{7,976 \text{ bytes}}{8,239 \text{ cycles}} \approx 0.97 \text{ bytes/cycle}$

With the maximum clock speed of the Atlys board at 100 MHz, we can get configuration speeds of approximately 97 MB/s with the tested custom instructions. Without the decompression module the configuration speed would have been limited to the throughput of the flash memory, which can deliver $0.5 \text{ bytes/cycle}$ continuously after transfer has been started.

Multi-cycle configuration of custom instructions always comes with an configuration overhead. If this overhead is larger than what is saved by using custom instructions, it is not worth the effort. By comparing the number of cycles used by the configuration controller to configure a custom instruction compared to the cost of running it in software, we get a number of how many times an instruction must be called before making the system faster:

For count ones: \[
\frac{\text{Configuration overhead}}{\text{Software execution}} = \frac{8,077 \text{ cycles}}{461 \text{ cycles}} \approx 18
\]

For CRC: \[
\frac{\text{Configuration overhead}}{\text{Software execution}} = \frac{8,239 \text{ cycles}}{568 \text{ cycles}} \approx 15
\]

For the CRC instruction, configuration pays off if the code uses the custom instruction more than 15 times before it is replaced by some other instruction. This test case is a worst case scenario where blocking of execution is started right after the reconfiguration process. In many cases the impact of the configuration process can be far less since the configuration controller works independently of the rest of the system. And by using pre-fetching, configuration overhead might be completely hidden. It is also important to note that the software side of the overhead can get larger if more advanced algorithms are used for placement and scheduling. This software overhead can not be hidden with pre-fetching since it requires the CPU.

### 4.2 Placing and relocating PR modules

#### 4.2.1 Introduction

In this test, we explored the placement of modules using our enhanced configuration controller (see section 3.6) with support for module relocation through bitstream changes. The implementation cost of the enhanced configuration controller is listed in Table 4.5.

| Table 4.5: Enhanced configuration controller, resource requirements. |
|---|---|---|---|---|
| Module | LUTs | Slices | Slice reg. | LUTRAM |
| Enhanced configuration controller | 658 | 275 | 631 | 40 |

65
4.2.2 Setup

For the test case, we used our full system with the MIPS, configuration controller and a PR region for modules. The PR region was implemented as described in section 3.8. In this test case, the streaming interface of the PR region was connected to a video stream from a background generation module as we are mostly interested in observing module relocation. As in the previous test case, a test program was written to read inputs and control the reconfiguration process. The code has many similarities with the biggest change being the command sequence that needs to be written to the configuration controller (sequence is shown in 4.3). Configuration time is measured in clock cycles from the start of the configuration process until the end. To block operation during reconfiguration, we used the user register in the configuration controller to give us a reference point for when the controller has executed all commands in the FIFOs. We used switches on the board to start configuration and chose to which position the module should be written.

During the test, the MIPS and connected peripherals running at 50 MHz clock speed. However, the configuration controller used internally a 100 MHz clock speed.

Listing 4.3: Command sequence in the c-code for configuring one of the two pacmans.

```c
1  *(flash) = 0; //read start pos
2  *(flash+1) = size_top; //size
3
4  *(flash) = 0x00000509; //16-bit to icap
5  *(flash+1) = 0x00000000; //Write ICAP
6
7  *(flash) = 0x00000038; //read start pos
8  *(flash+1) = size_mid; //size
9
10 *(flash) = 0x00000509; //16-bit to icap
11 *(flash+1) = 0x00000000; //Write ICAP
12
13 *(flash) = 0x00000730; //read start pos
14 *(flash+1) = size_bot; //size
15 *(flash) = user_code; //user reg
16 *(flash+1) = 0xFFFFFFFF; //Write user_reg
```

4.2.3 Test module

For this test case, we used a video overlay module that draws a bitmap representing a Packman on the screen. To fit our streaming architecture, the module only requires a 32-bit input and output signal and a clock to operate. By incrementing internal registers using the video sync signals, the Pacman bitmap is moved across the screen. The full module fits within
two columns of CLBs spanning the height of a clock region and requires 212 LUTs for implementation.

4.2.4 Placement and bitstream generation

The theory behind the placement and bitstream generation performed in this section is presented in Section 2.3.5. The Packman module presented in the previous section has a simple module footprint, it requires two columns of CLBs. The module has no special requirements when it comes to types of CLB columns, allowing for many potential placements with only one netlist/bitstream (see Figure 4.1).

To investigate differences in the bitstreams for different placements of a module on the device, we placed and routed a Pacman module within our PR region. Then we used a feature in GoAhead that allows use to cut out a part of a netlist and store it as a relocatable netlist. Using GoAhead, we could now place the module at any position on the device were it would fit (according to it’s resource footprint). By placing our extracted pacman at different horizontal positions on the device and generating differential bitstreams against an empty design, we produced a bitstream library of the same Packman module relocated to different horizontal positions.

In the next step, a script was used to compare the different bitstreams to find the position dependent parts. A small part of the output can be seen in listing 4.4. It is easy to see that there are two byte values that have to be changed in order to relocate the pacman module. It is these values that are written to the ICAP directly on line 4 and 10 in listing 4.3. Both position dependent values found with the diff tool are prefixed by the hex value 0x0322 in the bitstream. This is the command for writing two words (each word is 16-bit) to the Frame Address Register (FAR), meaning that the only change required to relocate the pacman module is to change two values in the bitstream.

Listing 4.4: Output from script comparing bitstreams generated from different placements of the pacman module.

```
1 Comparing files clexm_x13y95.bin and CLEXM_X15Y95.BIN>>>>>
2 Difference: 0000002F: 0E 10, Distance: 2
3 || Difference: 00000C73: 0E 10, Distance: 2
4
5 Comparing files clexm_x13y95.bin and CLEXM_X19Y95.BIN>>>>>
6 Difference: 0000002F: 0E 14, Distance: 6
7 || Difference: 00000C73: 0E 14, Distance: 6
8
9 Comparing files clexm_x13y95.bin and CLEXM_X21Y95.BIN>>>>>
10 Difference: 0000002F: 0E 16, Distance: 8
11 || Difference: 00000C73: 0E 16, Distance: 8
```
Figure 4.1: Overview of the creation of multiple bitstreams for different placements of the same module netlist, using GoAhead and bitgen

Table 4.6: Pacman bitstream size

<table>
<thead>
<tr>
<th>Module</th>
<th>Bitstream size</th>
<th>Compressed bitstream size</th>
<th>Compression ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pacman</td>
<td>8,194 bytes</td>
<td>4,860 bytes</td>
<td>1.68</td>
</tr>
</tbody>
</table>
4.2.5 Relocation results

The reconfiguration used 5645 cycles of the system clock from the first command written to configuration controller until the whole reconfiguration process was finished. The uncompressed and compressed size of the pac-man bitstream is shown in Table 4.6. The configuration clock is twice as fast as the system clock, so we can compute the effective configuration throughput for the whole configuration process:

Number clock cycles required: \(5645 \text{ cycles} \times 2 = 11290 \text{ cycles}\)

Number of bytes per cycle: \(\frac{8194 \text{ bytes}}{11290 \text{ cycles}} \approx 0.73 \text{ bytes/cycle}\)

Throughput estimate: \(0.73 \text{ bytes/cycle} \times 100 \text{ MHz} \approx 73 \text{ MB/s}\)

The configuration data throughput for the whole process is 73 MB/s. Since relocation in this case requires the bitstream to be split into three separate read operations from the flash and two direct writes from the MIPS to ICAP, the reconfiguration speed is slower than if everything was done in one read operation from the flash memory. The upside is that it allows us to only store one bitstream for each module and still be able to do module relocation.
Chapter 5

Conclusion and further work

The goal for this thesis was to build a PR system for demonstration and for education of partial runtime reconfiguration. In previous chapters, the necessary components and tools as well as the implementation of a fully functional PR system were described. The system can place and relocate runtime reconfigurable modules within a dedicated PR region and also perform dynamic instruction set extension of a MIPS CPU.

The most important parts of the implemented system are:

1. MIPS CPU with relocatable custom instructions.
2. Configuration controller capable of runtime reconfiguration with module relocation at a throughput of close to 100 MB/s.
3. Partial reconfigurable region with streaming interface supporting relocatable modules.

5.1 System performance

The system can run at a system clock of 50 MHz (with the configuration controller running at 100 MHz). For the system clock, the limiting factor is the MIPS CPU, for the configuration controller it is the external flash memory with peak clock frequency of 108 MHz.

For a PR system one important benchmark is reconfiguration time/speed. In Section 4.1.5 the number of cycles required to perform reconfiguration of two custom instructions were measured. It took the system 8077 cycles to reconfigure a custom instruction for counting ones into the system. A custom instruction for CRC computations took 8239 cycles. Taking the size of the bitstreams into consideration, the configuration speed at 100 MHz was approximately 97 MB/s for custom instructions. The configuration speed achieved for relocatable modules placed in the PR region was 73 MB/s (see Section 4.2.5 on page 69). Without the decompression module, the maximum achievable configuration speed would have been only 50 MB/s. Thus, for our testcases, bitstream compression allows us to configure modules 1.46 to 1.94 times faster than without.
Another important benchmark is the impact of using custom instructions to accelerate small software functions. For the two custom instructions tested in Section 4.1.5, a saving of 461 (count ones) and 568 (CRC) cycles compared to the software implementation was achieved. To break even with reconfiguration overhead the instructions would have to be called only 15 times for CRC and only 18 times for count ones. Design overhead is added when designers needed to implement hardware instructions. With the CRC function, we demonstrated that it is possible to generate custom instructions using C to HDL compilers like Catapult C from Mentor Graphics to save time when going from software functions to custom instructions. Further tests are needed to see if this approach works for a wider set of software functions.

5.2 Future work

Most of the system works as we expected, and especially the configuration controller achieves it’s target throughput. However, there are some improvements that can be done:

- In its current state, the MIPS lack proper runtime debugging. This could be improved by adding support for sending characters from the MIPS to a terminal running on a PC. This would require a software function (a scaled down version of printf) and a more advanced module for communication over COM-port. In addition GDB debug support should be added.

- The MIPS use BRAM as both program memory and data memory. A much more elegant solution would be to implement a simple cache controller connected to DDR-memory. This would allow us to write and execute larger programs and also store and manipulate large data structures, like, for example, frame buffers.

- The configuration controller presented in this thesis could be extended to support read-back of configuration data from ICAP. With this capability the system could be used in research were it is necessary to store the internal state of module. Cases were this could be relevant would be reconfiguration with module preemption and the possibility to roll-back a module to an earlier state [16].

Hopefully, with, the here developed system, we hope to stimulate future research and projects using partial configuration. This work is currently extended by another thesis integrating state capturing and restoring by reading back configuration data and extracting state information. Moreover, results of this thesis have resulted in a publication with the title "EasyPR – an Easy Usable Open-Source PR System", which has been submitted to the ICFPT 2013 conference.
Appendix A

MIPS custom instructions with inline assembly

A.1 Introduction

This is documentation on how to do custom instructions with GCC and BINUTILS on a simple MIPS softcore processor. More specific: adding support for your custom instructions on the MIPS with the use inline assembly in the code.

A.2 Modifying binutils source

The first step is to edit and compile binutils and gcc. So go to the folder where you have put the source files for gcc and binutils, then find ../binutils-2.23/opcode/mips-opc.c, this is the source file that defines all instructions for the MIPS architecture. Inside you find a struct that defines all legal instructions with name and format:

```c
const struct mips_opcode mips_builtin_opcodes[] =
{
/* These instructions appear first so that the disassembler will find them first. The assembler uses a hash table based on the instruction name anyhow. */
/* name, args,match, mask,pinfo, pinfo2,membership,[exclusions] */
{"pref", "k,o(b)", 0xcc000000, 0xfc000000, RD_b, 0,I4_32|G3},
{"pref", "k,A(b)",0, (int) M_PREF_AB,INSN_MACRO,0,I4_32|G3},
{"prefx", "h,t(b)",0x4c00000f, 0xfc0007ff, RD_b|RD_t|FP_S,0,I4_33},
{"nop", "", 0x00000000, 0xffffffff, 0, INSN2_ALIAS, I1}, /* sll */
...
...
```
Scrolling down through the struct there is a set of instructions named UDI (User Defined Instructions). So by changing the format (and name if you fell like it) of one or more of the UDI-instructions you can make your own assembly instructions. The UDIs are SPECIAL2-type instructions with the R-type format, so this means they all have the same opcode, but different funct-codes. You don't want to change opcodes or funct-code, only arguments and membership. The easiest way is to copy what you need from a known basic R-type instruction like xor:

```
{"xor", "d,v,t", 0x00000026, 0xfc0007ff, WR_d|RD_s|RD_t, 0,I1 },
```

Then use this to modify a UDI-instruction:

```
//{"udi0", "s,t,d,+1",0x70000010, 0xfc00003f, WR_d|RD_s|RD_t, 0, I33 },
{"cust", "d,v,t", 0x70000010, 0xfc0007ff, WR_d|RD_s|RD_t, 0, I1 },
```

The new instruction "cust" can now be used in your code with inline assembly:

```
__asm__ ("nop\n	" 
"cust %0, %1, %2\n\t" 
:"=r" (z) 
:"r" (x), "r" (y));
```

### A.3 Building crosscompiler

To compile your changed binutils source and gcc for the mips do the following: First you need to set some environment variables:

```
% export TARGET=mips-unknown-elf
% export PREFIX=/usr/local/$TARGET
% export PATH=$PATH:$PREFIX/bin
```

Then, setting your working directory to the path were you have your binutils-2.23 folder, do the following:

```
% mkdir build-binutils
% cd build-binutils
% ../binutils-2.23/configure --target=$TARGET --prefix=$PREFIX
% make all
% make install
```

You also need to build GCC:

```
% tar xjfv gcc-4.6.1.tar.bz2
% mkdir build-gcc
% cd build-gcc
% ../gcc-4.6.1/configure --target=$TARGET --prefix=$PREFIX --without-headers --with-newlib --with-gnu-as --with-gnu-ld \
% make all-gcc
% make install-gcc
```
Appendix B

VHDL code

In this appendix we present the VHDL-code for the MIPS CPU and the SPI flash reader module. The full system contains thousands lines of VHDL code, too much to append in this thesis.

B.1 MIPS CPU

This is the full source code of the MIPS CPU used in the system. The register file is contained in it's own entity for placement purposes.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity MIPS_CPU is
  port(
    clk : in std_logic;
    reset : in std_logic;
    WaitRequest : in std_logic;
    D_write_en : out std_logic;
    D_read_en : out std_logic;
    I_ADR : out std_logic_vector(31 downto 0);
    I_DATA : in std_logic_vector(31 downto 0);
    D_ADR : out std_logic_vector(31 downto 0);
    D_W_DATA : out std_logic_vector(31 downto 0);
    D_R_DATA : in std_logic_vector(31 downto 0);
    RES_0 : in std_logic_vector(31 downto 0);
    OP_A : out std_logic_vector(31 downto 0);
    OP_B : out std_logic_vector(31 downto 0))
  end MIPS_CPU;

architecture a_MIPS_CPU of MIPS_CPU is
  component MIPS_regfile is
    port(clk : in std_logic;
         DATA_A : out std_logic_vector(31 downto 0);
         DATA_B : out std_logic_vector(31 downto 0);
         WADR : in std_logic_vector(4 downto 0));
  end component;
end;
RegFile_en : in std_logic;
D_R_DATA : in std_logic_vector(31 downto 0);
ALU_out : in std_logic_vector(31 downto 0);
WaitRequest : in std_logic;
load : in std_logic;
rs : in std_logic_vector(4 downto 0);
rt : in std_logic_vector(4 downto 0);
);
end component;
signal regfile_load : std_logic;
type Instruction_type_type is (R_type, ADDI, ADDIU, SLTI, SLTIU, ANDI, ORI, XORI, LUI, j_type, BNE, BEQ, load, store, JAL, BLTZ, BGTZ, BLEZ, SPECIAL2);
type PC_type is (vanlig, branches);
signal cycle_en : std_logic;
signal wait_en : std_logic;
signal Instruction_type : Instruction_type_type;
signal PCstate : PC_type;
signal local_D_write_en : std_logic;
signal rs, rt, rd, sa : std_logic_vector(4 downto 0);
signal W_ADR : std_logic_vector(4 downto 0);
signal R_DATA_A, R_DATA_B : std_logic_vector(31 downto 0);
signal ALU_out : std_logic_vector(31 downto 0);
signal ALU_out64 : std_logic_vector(63 downto 0);
signal PC, PC4, nextPC, branchPC : std_logic_vector(31 downto 0);
signal RegFile_en : std_logic;
signal instr : std_logic_vector(5 downto 0);
signal funct : std_logic_vector(5 downto 0);
signal immediate, SL2immediate : std_logic_vector(31 downto 0);
signal immediateU : std_logic_vector(31 downto 0);
signal immediateJ : std_logic_vector(27 downto 0);
signal BranchTaken, branching : std_logic;
signal JumpTaken, JumpTakenJR : std_logic;
signal idata : std_logic_vector(31 downto 0);
signal HI, LO : std_logic_vector(31 downto 0);
signal WaitRequest_i : std_logic;
signal WaitRequest_comb : std_logic;
signal mul_wait : std_logic;
signal MTHI, MILO : std_logic;
signal mul_taken : std_logic;

begin
  --- Register File
  InstRegfile : MIPS_regfile
  port map(clk => clk,
          DATA_A => R_DATA_A,
          DATA_B => R_DATA_B,
          W_ADR => W_ADR,
          RegFile_en => RegFile_en,
          D_R_DATA => D_R_DATA,
          ALU_out => ALU_out,
          WaitRequest => WaitRequest_comb,
          load => regfile_load,
          rs => rs,
          rt => rt);
  regfile_load <= '1' when Instruction_type = load else '0';

  OP_A <= R_DATA_A;
  OP_B <= R_DATA_B;
  
  --- Setting idata to correct signals:
  idata <= I_DATA;
  funct <= idata(5 downto 0);
  instr <= idata(31 downto 26);
  rs <= idata(25 downto 21);
  rd <= idata(15 downto 11);
  rt <= idata(20 downto 16);
  sa <= idata(10 downto 6);
  
  --- Immediate sign extended:
  immediate(31 downto 16) <= (others => idata(15));
  immediate(15 downto 0) <= idata(15 downto 0);
  
  --- Immediate unsigned:
  immediateU <= x"0000" & idata(15 downto 0);
  
  --- Jump offset:
  immediateJ <= idata(25 downto 0) & "00";
  
  --- Immediate sign extended and leftshift 2:
  SL2immediate <= immediate(29 downto 0) & "00";
  
  --- Decoding instructions:
  p_INS_DECORDER : process (instr)
  begin
    case instr is
      when "000000" => Instruction_type <= R_type;
      when "000001" => Instruction_type <= BLTZ;
      when "010000" => Instruction_type <= ADDI;
      when "010001" => Instruction_type <= ADDIU;
      when "010100" => Instruction_type <= SLTI;
      when "010101" => Instruction_type <= SLTIU;
      when "011000" => Instruction_type <= ANDI;
      
      when others => Instruction_type <= None;
when "001101" => Instruction_type <= ORI;
when "001110" => Instruction_type <= XORI;
when "001111" => Instruction_type <= LUI;
when "000010" => Instruction_type <= J_type;
when "000100" => Instruction_type <= BEQ;
when "000101" => Instruction_type <= BNE;
when "000111" => Instruction_type <= BGTZ;
when "000110" => Instruction_type <= BLEZ;
when "100011" | "100000" => Instruction_type <= load;
when "101011" | "101000" => Instruction_type <= store;
when "011100" => Instruction_type <= SPECIAL2;
when "000011" => Instruction_type <= JAL;
when others =>
    Instruction_type <= R_type;
    report " +++ unimplemented instruction type !! ";
end case;
end process;

WaitRequest_i <= '0' when (mul_taken = '1' and mul_wait = '0') else '1';
WaitRequest_comb <= WaitRequest and WaitRequest_i;
InstMulreg : process(clk)
begin
if rising_edge(clk) then
    if MTHI = '1' then
        HI <= R_DATA_A;
    elsif mul_wait = '1' then
        HI <= ALU_out64(63 downto 32);
    end if;
    if MILO = '1' then
        LO <= R_DATA_A;
    elsif mul_wait <= '1' then
        LO <= ALU_out64(31 downto 0);
    end if;
    if mul_taken = '1' and mul_wait = '0' then
        mul_wait <= '1';
    else
        mul_wait <= '0';
    end if;
end if;
end process;

--ALU process:
D_write_en <= local_D_write_en;
DADR <= ALU_out;
DWDATA <= R_DATA_B;
p_ALU: process (PC, hi, lo, WaitRequest_comb, RES_0,
    Instruction_type, funct, instr, rt, rd, rs, sa,
    immediate, immediateU, SL2immediate, R_DATA_A,
    R_DATA_B, ALU_out, ALU_out64, WADDR)
begin
    -- if WaitRequest /= '1' then
--- default values:
ALU_out <= (others => '0');
ALU_out64 <= (others => '0');
JumpTaken <= '0';
BranchTaken <= '0';
W_ADR <= (others => '0');
RegFile_en <= '0';
D_read_en <= '0';
local_D_write_en <= '0';
JumpTakenJR <= '0';
MTHI <= '0';
MILO <= '0';
mul_taken <= '0';
case Instruction_type is
  when R_type =>
    RegFile_en <= '1';
    W_ADR <= rd;
  case funct is
    when "000000" => --I:SLL
      ALU_out <= std_logic_vector(unsigned(R_DATA_B) SLL to_integer(unsigned(sa))); --I:SLL
    when "000010" => --I:SRL
      ALU_out <= std_logic_vector(unsigned(R_DATA_B) SRL to_integer(unsigned(sa))); --I:SRL
    when "000110" => --I:SRLV
      ALU_out <= std_logic_vector(unsigned(R_DATA_B) SRL to_integer(unsigned(R_DATA_A))); --I:SRLV
    when "000100" => --I:SLLV
      ALU_out <= std_logic_vector(unsigned(R_DATA_B) SLL to_integer(unsigned(R_DATA_A))); --I:SLLV
    when "000011" => --I:SRA
      ALU_out <= std_logic_vector(unsigned(R_DATA_B) SRL to_integer(unsigned(sa))); --I:SRA
    when "000111" => --I:SRAV
      ALU_out <= std_logic_vector(unsigned(R_DATA_B) SRL to_integer(unsigned(R_DATA_A))); --I:SRAV
    when "101010" => --I:SLT
      if signed(R_DATA_A) < signed(R_DATA_B) then
        ALU_out <= x"00000001"; --I:SLT
      else
        ALU_out <= (others => '0'); --I:SLT
      end if; --I:SLT
    when "101011" => --I:SLTU
      if unsigned(R_DATA_A) < unsigned(R_DATA_B) then
        ALU_out <= x"00000001"; --I:SLTU
      else
        ALU_out <= (others => '0'); --I:SLTU
      end if; --I:SLTU
    when "100001" => --I:ADDU
ALU_out <= std_logic_vector(unsigned(R_DATA_A) + unsigned(R_DATA_B)); -- I:ADD
when "100000" => -- I:ADD
ALU_out <= std_logic_vector(signed(R_DATA_A) + signed(R_DATA_B)); -- I:ADD
when "100010" => -- I:ADD
ALU_out <= std_logic_vector(signed(R_DATA_A) - signed(R_DATA_B)); -- I:SUB
when "100011" => -- I:SUB
ALU_out <= std_logic_vector(unsigned(R_DATA_A) - unsigned(R_DATA_B)); -- I:SUBU
when "100100" => -- I:AND
ALU_out <= R_DATA_A and R_DATA_B; -- I:AND
when "100101" => -- I:OR
ALU_out <= R_DATA_A or R_DATA_B; -- I:OR
when "100110" => -- I:XOR
ALU_out <= R_DATA_A xor R_DATA_B; -- I:XOR
when "100111" => -- I:NOR
ALU_out <= R_DATA_A nor R_DATA_B; -- I:NOR
when "010000" => -- I:MFHI
ALU_out <= HI; -- I:MFHI
when "010010" => -- I:MFLO
ALU_out <= LO; -- I:MFLO
when "010001" => -- I:MTHI
MTHI <= '1'; -- I:MTHI
when "010011" => -- I:MTLO
MTLO <= '1'; -- I:MTLO
when "001000" => -- I:JR
JumpTakenJR <= '1'; -- I:JR
RegFile_en <= '0'; -- I:JR
when "001001" => -- I:JALR
ALU_out <= std_logic_vector(unsigned(PC) + 8); -- I:JALR
JumpTakenJR <= '1'; -- I:JALR
when "001011" => -- I:MOV
ALU_out <= R_DATA_A; -- I:MOV
if R_DATA_B = x"00000000" then -- I:MOV
RegFile_en <= '0'; -- I:MOV
end if; -- I:MOV
when "011010" => -- I:MOVZ
ALU_out <= R_DATA_A; -- I:MOVZ
if R_DATA_B /= x"00000000" then -- I:MOVZ
RegFile_en <= '0'; -- I:MOVZ
end if; -- I:MOVZ
when "011000" =>
mul_taken <= '1';
ALU_out64 <= std_logic_vector(signed(R_DATA_A) * signed(R_DATA_B));
when "011001" =>
mul_taken <= '1';
ALU_out64 <= std_logic_vector(unsigned(R_DATA_A) * unsigned(R_DATA_B));
when others =>
report "+++ unimplemented instruction type !! ";
end case;
when ADDIU => --I:ADDIU
  RegFile_en <= '1'; --I:ADDIU
  WADR <= rt; --I:ADDIU
  ALU_out <= std_logic_vector(unsigned(R_DATA_A) + 
              unsigned(immediate)); --I:ADDIU
when ADDI => --I:ADDI
  RegFile_en <= '1'; --I:ADDI
  WADR <= rt; --I:ADDI
  ALU_out <= std_logic_vector(signed(R_DATA_A) + 
              signed(immediate)); --I:ADDI
when SLTIU => --I:SLTIU
  RegFile_en <= '1'; --I:SLTIU
  WADR <= rt; --I:SLTIU
  if unsigned(R_DATA_A) < unsigned(immediateU) then
    ALU_out <= (0 => '1', others => '0'); --I:SLTIU
  else
    ALU_out <= (others => '0'); --I:SLTIU
  end if; --I:SLTIU
when SLTI => --I:SLTI
  RegFile_en <= '1'; --I:SLTI
  WADR <= rt; --I:SLTI
  if signed(R_DATA_A) < signed(immediate) then --I:
    ALU_out <= (0 => '1', others => '0'); --I:SLTI
  else
    ALU_out <= (others => '0'); --I:SLTI
  end if; --I:SLTI
when ANDI => --I:ANDI
  RegFile_en <= '1'; --I:ANDI
  WADR <= rt; --I:ANDI
  ALU_out <= R_DATA_A and immediateU; --I:ANDI
when ORI => --I:ORI
  RegFile_en <= '1'; --I:ORI
  WADR <= rt; --I:ORI
  ALU_out <= R_DATA_A or immediateU; --I:ORI
when XORI => --I:XORI
  RegFile_en <= '1'; --I:XORI
  WADR <= rt; --I:XORI
  ALU_out <= R_DATA_A xor immediateU; --I:XORI
when LUI => --I:LUI
  RegFile_en <= '1'; --I:LUI
  WADR <= rt; --I:LUI
  ALU_out <= immediate(15 downto 0) & X"0000"; --I:
  LUI
when BNE => --I:BNE
  if R_DATA_A /= R_DATA_B then --I:BNE
    BranchTaken <= '1'; --I:BNE
  end if; --I:BNE
when BEQ => --I:BEQ
  if R_DATA_A = R_DATA_B then --I:BEQ
309 BranchTaken <= '1'; --I:BEQ
310 end if; --I:BEQ
311 when BGTZ => --I:BGTZ
312 if signed(R_DATA_A) > x"00000000" then --I:BGTZ
313 BranchTaken <= '1'; --I:BGTZ
314 end if; --I:BGTZ
315 when BLEZ => --I:BLEZ
316 if signed(R_DATA_A) <= x"00000000" then --I:BLEZ
317 BranchTaken <= '1'; --I:BLEZ
318 end if; --I:BLEZ
319 when load => --I:LW
320 RegFile_en <= '1'; --I:LW
321 D_read_en <= '1'; --I:LW
322 W_ADR <= rt; --I:LW
323 ALU_out <= std_logic_vector(signed(immediate) + signed(R_DATA_A)); --I:LW
324 when store => --I:SW
325 local_D_write_en <= '1'; --I:SW
326 ALU_out <= std_logic_vector(signed(immediate) + signed(R_DATA_A)); --I:SW
327 when J_type => --I:J
328 JumpTaken <= '1'; --I:J
329 when JAL => --I:JAL
330 RegFile_en <= '1'; --I:JAL
331 W_ADR <= "11111"; --I:JAL
332 ALU_out <= std_logic_vector(unsigned(PC) + 8); --I:JAL
333 JumpTaken <= '1'; --I:JAL
334 when SPECIAL2 =>
335 RegFile_en <= '1'; --I:?  
336 W_ADR <= rd; --I:?  
337 if funct = "010000" then --I:CUST
338 ALU_out <= RES_0; --I:CUST
339 else
340 report " +++ unimplemented instruction type !! ";
341 end if;
342 when BLTZ =>
343 if rt = "00000" then
344 if signed(R_DATA_A) < x"00000000" then --I:BLTZ
345 BranchTaken <= '1'; --I:BLTZ
346 end if;
347 elsif rt = "00001" then --I:BGEZ
348 if signed(R_DATA_A) >= x"00000000" then --I:BGEZ
349 BranchTaken <= '1'; --I:BGEZ
350 end if; --I:BGEZ
351 elsif rt = "10001" then --I:BGEZAL
352 W_ADR <= "11111"; --I:BGEZAL
353 ALU_out <= std_logic_vector(unsigned(PC) + 8); --I:BGEZAL
354 if signed(R_DATA_A) >= x"00000000" then --I:BGEZAL
355 BranchTaken <= '1'; --I:BGEZAL
356 end if; --I:BGEZAL
357 end if; --I:BGEZAL
end if; —I:BGEZAL

elsif rt = "10000" then —I:BLTZAL
  W adr <= "11111"; —I:BLTZAL
  ALU_out <= std_logic_vector(unsigned(PC) + 8);
  —I:BLTZAL
  if signed(R_DATA_A) <= x"00000000" then —I:
    BranchTaken <= '1'; —I:BLTZAL
  end if; —I:BLTZAL
end if;

when others =>
  report " +++ unimplemented instruction type !! ";
end case;
end if;
end process;

with PCstate select
  nextPC <= PC4 when vanlig,
  branchPC when branchs,
  PC4 when others;

I_ADR <= nextPC when WaitRequest_comb = '1' else PC;
PC4 <= std_logic_vector(unsigned(PC) + 4);
process (clk)
begin
  if clk'event and clk = '1' then
    if WaitRequest_comb = '1' then
      if reset = '1' then
        PCstate <= vanlig;
        PC <= X"BFC00000";
        branchPC <= X"BFC00000";
      else
        PC <= nextPC;
      end if;
    end if;
    case PCstate is
      "Normal" state of PC:
      when vanlig =>
        —If a branch is taken:
        if BranchTaken = '1' then
          PCstate <= branchs;
          branchPC <= std_logic_vector(signed(PC4) +
                                      signed(SL2immediate));
        —If a jump is taken:
        elsif JumpTaken = '1' then
          PCstate <= branchs;
          branchPC <= PC4(31 downto 28) & immediateJ;
        — If a jump from register is taken:
        elsif JumpTakenJR = '1' then
          PCstate <= branchs;
          branchPC <= R_DATA_A;
        else
          PCstate <= vanlig;
        end if;
      — branch state of PC:
      when branchs =>
        —...
when branchs =>
  PCstate <= vanlig;
when others =>
  PCstate <= vanlig;
end case;
end if;
end if;
end if;
end process;
end;

The register file:

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity MIPS_regfile is
  port ( clk : in std_logic;
         DATA_A : out std_logic_vector(31 downto 0);
         DATA_B : out std_logic_vector(31 downto 0);
         W_ADR : in std_logic_vector( 4 downto 0 );
         RegFile_en : in std_logic;
         D_R_DATA : in std_logic_vector(31 downto 0);
         ALU_out : in std_logic_vector(31 downto 0);
         WaitRequest : in std_logic;
         load : in std_logic;
         rs : in std_logic_vector( 4 downto 0 );
         rt : in std_logic_vector( 4 downto 0 )
    );
end entity;

architecture rtl of MIPS_regfile is

type memtype is array (31 downto 0) of std_logic_vector
     (31 downto 0);
signal RegFile : memtype := ( others => ( others => '0' ) );

begin
  p_write : process ( clk )
  begin
    if clk'event and clk = '1' then
      —if Wait_en = '1' then
        if WaitRequest = '1' then
          if RegFile_en = '1' and (W_ADR /= (W_ADR' range => '0')) then
            — we initialize reg with 0 and prohibit to overwrite it in the future
            if (load = '1') then
              RegFile(to_integer(unsigned(W_ADR))) <= D_R_DATA;
            else
RegFile(to_integer(unsigned(W_ADR))) <= ALU_out;
end if;
end if; -- RegFileEnable
end if; -- Waitrequest
end if; -- clk
end process;
DATA_A <= RegFile(to_integer(unsigned(rs)));
DATA_B <= RegFile(to_integer(unsigned(rt)));
end architecture;

B.2 SPI flash reader

Presented in this section is the SPI flash reader from the configuration controller (as seen in Figure 3.6 on page 43):

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity my_Flash_Read is
port (
  clk100 : in std_logic;
  reset : in std_logic;
  ADR : in std_logic_vector(23 downto 0);
  request : in std_logic; -- Reads from flash while '1'
  stall : in std_logic; -- Stalls flash reading while high
  data_ready : out std_logic;
  quad_ready : out std_logic;
  DATA : out std_logic_vector(7 downto 0);
  FlashCLK : out std_logic;
  FlashCS : out std_logic;
  FlashMemDq : inout std_logic_vector(3 downto 0)
);
end my_Flash_Read;

architecture beh of my_Flash_Read is

-- Configuration register signals:
type instructionType_type is (wren, wrecr, wait1, wait2, idle, cmd1, cmd2, addr23_20, addr19_16, addr15_12, addr11_8, addr7_4, addr3_0, dummy, recvstart, recvlow, recvhig, waitst);
signal instructionType : instructionType_type;
signal config_select : std_logic := '0';
signal bitcounter : integer range -1 to 25 := 25;
begin
  data <= data_i;
  FlashCS <= FlashCS_config;
  FlashCLK_en <= Fclk_en_config;
  data_ready <= data_ready_i;
  quad_ready <= quad_ready_i;
write_config_reg : process (clk100)
begin
  if rising_edge(clk100) then
    if reset = '1' then
      config_select <= '0';
      FlashCS_config <= '0';
      bitcounter <= 9;
      instructionType <= wren; —Setup flash for QIO
      DQ_dir <= '1';
    else
      case instructionType is
        —Setup flash
        when wren =>
          bitcounter <= bitcounter - 1;
if bitcounter = 9 then
    FlashCS_config <= '0';
elsif bitcounter <= 8 and bitcounter >= 1 then
    DQ_dir <= '1';
    Fclk_en_config <= '1';
    DQ_send(0) <= wren_ins(bitcounter-1);
    DQ_send(3 downto 1) <= "11Z";
elsif bitcounter = 0 then
    FlashCS_config <= '1';
    Fclk_en_config <= '0';
    bitcounter <= 10;
    instructionType <= wait1;
end if;
when wait1 =>
    FlashCS_config <= '1';
    Fclk_en_config <= '0';
    bitcounter <= bitcounter - 1;
if bitcounter = 0 then
    bitcounter <= 17;
    instructionType <= wrecr;
end if;
when wrecr =>
    bitcounter <= bitcounter - 1;
if bitcounter = 17 then
    FlashCS_config <= '0';
elsif bitcounter <= 16 and bitcounter >= 9 then
    DQ_dir <= '1';
    Fclk_en_config <= '1';
    if config_select = '1' then
        DQ_send(0) <= vconfig_ins(bitcounter-9);
        DQ_send(3 downto 1) <= "11Z";
    else
        DQ_send(0) <= vconfig_ins(bitcounter-9);
        DQ_send(3 downto 1) <= "11Z";
    end if;
elsif bitcounter <= 8 and bitcounter >= 1 then
    if config_select = '1' then
        DQ_send(0) <= vconfig_reg(bitcounter-1);
        DQ_send(3 downto 1) <= "11Z";
    else
        DQ_send(0) <= vconfig_reg(bitcounter-1);
        DQ_send(3 downto 1) <= "11Z";
    end if;
elsif bitcounter = 0 then
    Fclk_en_config <= '0';
    FlashCS_config <= '1';
    if config_select = '1' then
        quad_ready_i <= '1';
        instructionType <= idle;
    else
        quad_ready_i <= '0';
        bitcounter <= 10; -- endret fra 8
        instructionType <= wait2;
        config_select <= '1';
end if;
end if;
end if;

when wait2 =>
  FlashCS_config <= '1';
  Fclk_en_config <= '0';
  bitcounter <= bitcounter - 1;
  if bitcounter = 0 then
    bitcounter <= 9;
    instructionType <= wren;
  end if;

---End Setup flash---

when idle => --Wait for request
  FlashCS_config <= '1';
  Fclk_en_config <= '0';
  DQ_dir <= '1';
  bitcounter <= 9;
  if request = '1' then
    instructionType <= cmd1;
  else
    instructionType <= idle;
  end if;
when cmd1 => --Send fast read command
  Fclk_en_config <= '1';
  FlashCS_config <= '0';
  DQ_send <= x"6";
  instructionType <= cmd2;
when cmd2 =>
  DQ_send <= x"B";
  instructionType <= addr23_20;
when addr23_20 => --Send start address to flash
  DQ_send <= ADR(23 downto 20);
  instructionType <= addr19_16;
when addr19_16 =>
  DQ_send <= ADR(19 downto 16);
  instructionType <= addr15_12;
when addr15_12 =>
  DQ_send <= ADR(15 downto 12);
  instructionType <= addr11_8;
when addr11_8 =>
  DQ_send <= ADR(11 downto 8);
  instructionType <= addr7_4;
when addr7_4 =>
  DQ_send <= ADR(7 downto 4);
  instructionType <= addr3_0;
when addr3_0 =>
  DQ_send <= ADR(3 downto 0);
  instructionType <= dummy;
when dummy => --Wait 10 dummy cycles
  if bitcounter = 0 then
    instructionType <= recvstart;
  else
    bitcounter <= bitcounter - 1;
    instructionType <= dummy;
  end if;
when recvstart => —Receive data from flash:
  DQ_dir <= '0';
  instructionType <= recvhigh;
when recvhigh =>
  instructionType <= recvlow;
when recvlow =>
  instructionType <= recvhigh;
  if request = '0' then
    instructionType <= idle;
  elsif stall = '1' then
    instructionType <= waitst;
    Fclk_en_config <= '0';
  end if;
when waitst => —Stalled
  Fclk_en_config <= '0';
  instructionType <= waitst;
  if stall = '0' then
    instructionType <= recvhigh;
    Fclk_en_config <= '1';
  end if;
when others =>
  instructionType <= wren;
  quad_ready_i <= '0';
end case;
end if;
end if;
end process;

—Buffer for data form flash:
p_data_buffer_in : process(clk100)
begin
  if falling_edge(clk100) then
    data_in <= DQ_in;
  end if;
end process;

—Output buffer for data from flash:
p_data_buffer : process(clk100)
begin
  if rising_edge(clk100) then
    data_in <= DQ_in;
  else
    if instructionType = recvhigh then
      data_high <= '1';
    else
      data_high <= '0';
    end if;
    if instructionType = recvlow then
      data_low <= '1';
    else
      data_low <= '0';
    end if;
    if data_high = '1' then
      data_tmp <= data_in;
    data_ready_i <= '0';
  end if;
end process;
elsif data_low = '1' then
    data_i <= data_tmp & data_in;
    data_ready_i <= '1';
else
    data_ready_i <= '0';
end if;
end if;
end process;

-- Tristate driver on DQ inout:
P_tristate : process (FlashMemDq, DQ_dir, DQ_send)
begin
    DQ_in <= FlashMemDq;
    if DQ_dir = '1' then
        FlashMemDq <= DQ_send;
    else
        FlashMemDq <= "ZZZZ";
    end if;
end process;

-- Creating flash clock:
not_clk <= not clk100;
ODDR2_inst : ODDR2
generic map(DDR_ALIGNMENT => "NONE",
            INIT => '0',
            SRTYPE => "SYNC")
port map(Q => FlashCLK,
            C0 => clk100,
            C1 => not_clk,
            CE => '1',
            D0 => '0',
            D1 => FlashCLK_en,
            R => '0',
            S => '0');
end architecture;
Appendix C

Scripts

Most of the work done with ISE and GoAhead in this thesis has been automated using scripts.

C.1 Batch-script

Batch scripts are used to control the tool flow, much like it is shown in Figure 2.9 on page 24. Below is the batch script used to create the system. It runs GoAhead with GOA scripts, and also all the ISE tools needed to create the bitstream:

```batch
@Set STATIC_DIRECTORY=C:\Master\Full_system_video
@if "%1"="" GOTO CALL
@GOJO %1

:CALL
@Echo " Call with: run_partial.bat operation
    place_module_pos place_module_name Inst_tile"
@GOJO eof

:BeforeGOA

:OnlyGOA
GoAhead_pre82 -exec "%STATIC_DIRECTORY%\scripts\big_area.goa"
goahead_pre82 -exec "%STATIC_DIRECTORY%\scripts\C1_two_slots_west.goa"

cp top.ucf top.ucf.bak
cat top.ucf | grep -v generated_by_GoAhead | grep -v "could not find" > tmp.ucf
cat goahead.ucf >> tmp.ucf
cat GoAhead_constraints.ucf >> tmp.ucf
mv tmp.ucf top.ucf

@if "%1"="OnlyGOA" GOJO eof

GOJO AfterMap
```

91
BeforeMap

```
xst -intstyle ise -ifn "./top.xst" -ofn "./top.syr"
ngdbuild -intstyle ise -dd Ngo -nt timestamp -i -p xc6slx45-csg324-2 top.nc top.ngd
map -intstyle ise -p xc6slx45-csg324-2 -w -logic_opt off -ol high -t 1 -xt 0 -register_duplication off -r 4 -global_opt off -mt off -ir off -pr off -lc off -power off -o top_map.ncd top.ngd toppcf
```

AfterMap

```
REM this is a workaround for a Xilinx tool bug
@ECHO #################################
@ECHO ### Unbind macros in placed design (otherwise macro routing gets lost during XDL conversion)
@echo setattr main edit-mode read-write > unbind_macros.scr
@echo open design top_map.ncd toppcf >> unbind_macros.scr
@echo select hm * >> unbind_macros.scr
@echo unbind >> unbind_macros.scr
@echo save -w design "top_unbind.ncd" >> unbind_macros.scr
@echo exit >> unbind_macros.scr
yes | fpga_edline -p unbind_macros.scr
```

```
GoAhead_pre82 -command FixS6XDLBug XDLInFile=top_map.xdl
XDLOutFile=top_map.xdl;
```

```
GoAhead_pre82 -command MergeBlockerAndConnectClock
XDLInFile=top_map.xdl XDLOutFile=top_blocked1.xdl
BUFGInstanceName=NULL XDLBlockerFiles=big_blocker.xdl;
```

```
GoAhead_pre82 -command MergeBlockerAndConnectClock
XDLInFile=top_blocked1.xdl XDLOutFile=top_blocked2.xdl
BUFGInstanceName=NULL XDLBlockerFiles=C1_blocker.xdl;
```

```
xl -ncd2xdl top_unbind.ncd top_map.xdl
```

```
GoAhead_pre82 -command FixS6XDLBug XDLInFile=top_map.xdl
XDLOutFile=top_map.xdl;
```

```
GoAhead_pre82 -command FixS6XDLBug XDLInFile=top_routed.xdl
```

```
cat top_routed.xdl | sed 's/;//g' | tr "\n" "$" | tr "!" "\n" | grep -v net...RBB_Blocker | grep -v net...
```

```
xdl -ncd2xdl top_routed.ncd top_routed.xdl
bitgen -d -w Binary:Yes -g CRC:Disable top_routed.ncd top.bit
```

EOF
C.2 GOA-script

Below is an example of a GOA-script. GOA-scripts contain commands for GoAhead, and is way of automating the design process in the tool. The presented script is for creating two CI slots to the west of the MIPS in the static system:

```
1 OpenBinFPGA FileName="%GOAHEAD_HOME%\Devices\xc6slx45csg324-3.binFPGA";
2 #SetDefine Pattern=STATIC_DIRECTORY Value=C:\Master\Full_system;
3 #PrintToFile Text="" FileName="%STATIC_DIRECTORY%\hdl\GoAhead.vhd" Append=False CreateBackupFile=True;
4 #PrintToFile Text="" FileName="%STATIC_DIRECTORY%\hdl\GoAhead_constraints.ucf" Append=False CreateBackupFile=True;

5 # place connection macros – Slot 0 – A
6 ClearSelection;
7 AddToSelectionXY UpperLeftX=25 UpperLeftY=122 LowerRightX=26 LowerRightY=129;
8 ExpandSelection;
9 StoreCurrentSelectionAs UserSelectionType=Slot_0A; # allows us to select this slot symbolically
10 AddInstantiationInSelectedTiles
11 Mode=row-wise Horizontal=left-to-right Vertical=bottom-up
12 SliceNumber=1 InstanceName=Slot_0_A
13 LibraryElementName=Connect4_S6_double_reg
14 AddToNetlistContainer=True
15 NetlistContainerName=default_netlist_container;

18 # place connection macros – Slot 0 – B
19 ClearSelection;
20 AddToSelectionXY UpperLeftX=23 UpperLeftY=122 LowerRightX=24 LowerRightY=129;
21 ExpandSelection;
22 StoreCurrentSelectionAs UserSelectionType=Slot_0B; # allows us to select this slot symbolically
23 AddInstantiationInSelectedTiles
24 Mode=row-wise Horizontal=left-to-right
25 Vertical=bottom-up SliceNumber=1
26 InstanceName=Slot_0_B
27 LibraryElementName=Connect4_S6_double_reg
28 AddToNetlistContainer=True
29 NetlistContainerName=default_netlist_container;

31 # place connection macros – Slot 1 – A
32 ClearSelection;
33 AddToSelectionXY UpperLeftX=21 UpperLeftY=122 LowerRightX=22 LowerRightY=129;
34 ExpandSelection;
35 StoreCurrentSelectionAs UserSelectionType=Slot_1A; # allows us to select this slot symbolically
```
AddInstantiationInSelectedTiles
  Mode=row-wise Horizontal=left-to-right
  Vertical=bottom-up SliceNumber=1
  InstanceName=Slot_1_A
  LibraryElementName=Connect4_S6_double_reg
  AddToNetlistContainer=True
  NetlistContainerName=default_netlist_container;
#
  place connection macros – Slot 1 – B
ClearSelection;
AddToSelectionXY UpperLeftX=19 UpperLeftY=122 LowerRightX=20 LowerRightY=129;
ExpandSelection;
StoreCurrentSelectionAs UserSelectionType=Slot_1B;  # allows us to select this slot symbolically
AddInstantiationInSelectedTiles
  Mode=row-wise Horizontal=left-to-right
  Vertical=bottom-up SliceNumber=1
  InstanceName=Slot_1_B
  LibraryElementName=Connect4_S6_double_reg
  AddToNetlistContainer=True
  NetlistContainerName=default_netlist_container;
ClearSignalAnnotations InstantiationFilter=(^ Slot_0_A_0$) (! (^ Slot_0_A_1$) (! (^ Slot_0_A_2$) (! (^ Slot_0_A_3$) (! (^ Slot_0_A_4$) (! (^ Slot_0_A_5$) (! (^ Slot_0_A_6$) (! (^ Slot_0_A_7$)
AnnotateSignalNames InstantiationFilter=(^ Slot_0_A_0$) (! (^ Slot_0_A_1$) (! (^ Slot_0_A_2$) (! (^ Slot_0_A_3$) (! (^ Slot_0_A_4$) (! (^ Slot_0_A_5$) (! (^ Slot_0_A_6$) (! (^ Slot_0_A_7$)
PortMapping=I: OP_A: external, O: not_used0 : internal, H: 1: internal, CLK: clk: no_vector;
ClearSignalAnnotations InstantiationFilter=(^ Slot_0_B_0$) (! (^ Slot_0_B_1$) (! (^ Slot_0_B_2$) (! (^ Slot_0_B_3$) (! (^ Slot_0_B_4$) (! (^ Slot_0_B_5$) (! (^ Slot_0_B_6$) (! (^ Slot_0_B_7$)
AnnotateSignalNames InstantiationFilter=(^ Slot_0_B_0$) (! (^ Slot_0_B_1$) (! (^ Slot_0_B_2$) (! (^ Slot_0_B_3$) (! (^ Slot_0_B_4$) (! (^ Slot_0_B_5$) (! (^ Slot_0_B_6$) (! (^ Slot_0_B_7$)
ClearSignalAnnotations InstantiationFilter=(^ Slot_1_A_0$) (! (^ Slot_1_A_1$) (! (^ Slot_1_A_2$) (! (^ Slot_1_A_3$) (! (^ Slot_1_A_4$) (! (^ Slot_1_A_5$) (! (^ Slot_1_A_6$) (! (^ Slot_1_A_7$)
AnnotateSignalNames InstantiationFilter=(^ Slot_1_A_0$) (! (^ Slot_1_A_1$) (! (^ Slot_1_A_2$) (! (^ Slot_1_A_3$) (! (^ Slot_1_A_4$) (! (^ Slot_1_A_5$) (! (^ Slot_1_A_6$) (! (^ Slot_1_A_7$)
ClearSignalAnnotations
InstantiationFilter=(^Slot_1_B_0$)
|(^Slot_1_B_1$) |(^Slot_1_B_2$) |(^Slot_1_B_3$) |(^Slot_1_B_4$) |(^Slot_1_B_5$) |(^Slot_1_B_6$) |(^Slot_1_B_7$) ;

AnnotateSignalNames
InstantiationFilter=(^Slot_1_B_0$) |(^Slot_1_B_1$) |(^Slot_1_B_2$) |(^Slot_1_B_3$) |(^Slot_1_B_4$) |(^Slot_1_B_5$) |(^Slot_1_B_6$) |(^Slot_1_B_7$)

# generate the VHDL declarations
PrintVHDLWrapper
InstantiationFilter=(^Slot_0_A_.*$) |(^Slot_0_B_.*$)
EntityName="CI_wrapper_slot0"
FileName="%STATIC_DIRECTORY%\hdl\CI_wrapper_slot0.vhd"#
look in GUI VHDL window
Append=False
CreateBackupFile=True;

PrintVHDLWrapper
InstantiationFilter=(^Slot_1_A_.*$) |(^Slot_1_B_.*$)
EntityName="CI_wrapper_slot1"
FileName="%STATIC_DIRECTORY%\hdl\CI_wrapper_slot1.vhd"#
look in GUI VHDL window
Append=False
CreateBackupFile=True;

# create UCF placement constraints for connection macros
PrintLocationConstraints InstanceNamePrefix=
HierarchyPrefix="*
FileName="%STATIC_DIRECTORY%\GoAhead_constraints.ucf"
Append=False CreateBackupFile=True;

# define tunnels before blocking
# First we create the tunnels for the A slots that do not drive outputs
ClearSelection;
SelectUserSelection UserSelectionType=Slot_0A;
SelectUserSelection UserSelectionType=Slot_1A;
ExcludePortsFromBlockingInSelection PortName=WW2E0
CheckForExistence=False IncludeAllPorts=False;
ExcludePortsFromBlockingInSelection PortName=WW2E1
CheckForExistence=False IncludeAllPorts=False;
ExcludePortsFrom BlockingInSelection PortName=WW2E2
CheckForExistence=False IncludeAllPorts=False;
ExcludePortsFrom BlockingInSelection PortName=WW2E3
CheckForExistence=False IncludeAllPorts=False;

# Then we create the tunnels for the B slots that do drive outputs
Finally, we allow the double line extension in Slot_0

# create blocker
AddNetlistContainer NetlistContainerName=CI_blocker;
ClearSelection;
SelectUserSelection UserSelectionType=Slot_0A;
SelectUserSelection UserSelectionType=Slot_0B;
ExcludePortsFromBlockingInSelection PortName=WW2E0
  CheckForExistence=False IncludeAllPorts=False;
ExcludePortsFromBlockingInSelection PortName=WW2E1
  CheckForExistence=False IncludeAllPorts=False;
ExcludePortsFromBlockingInSelection PortName=WW2E2
  CheckForExistence=False IncludeAllPorts=False;
ExcludePortsFromBlockingInSelection PortName=WW2E3
  CheckForExistence=False IncludeAllPorts=False;
ExcludePortsFromBlockingInSelection PortName=EE4B0
  CheckForExistence=False IncludeAllPorts=False;
ExcludePortsFromBlockingInSelection PortName=EE4B1
  CheckForExistence=False IncludeAllPorts=False;
ExcludePortsFromBlockingInSelection PortName=EE4B2
  CheckForExistence=False IncludeAllPorts=False;
ExcludePortsFromBlockingInSelection PortName=EE4B3
  CheckForExistence=False IncludeAllPorts=False;

# finally, we allow the double line extension in Slot_0
ClearSelection;
SelectUserSelection UserSelectionType=Slot_0A;
SelectUserSelection UserSelectionType=Slot_0B;
ExcludePortsFromBlockingInSelection PortName=WW2B0
  CheckForExistence=False IncludeAllPorts=False;
ExcludePortsFromBlockingInSelection PortName=WW2B1
  CheckForExistence=False IncludeAllPorts=False;
ExcludePortsFromBlockingInSelection PortName=WW2B2
  CheckForExistence=False IncludeAllPorts=False;
ExcludePortsFromBlockingInSelection PortName=WW2B3
  CheckForExistence=False IncludeAllPorts=False;

# create blocker
AddNetlistContainer NetlistContainerName=CI_blocker;
ClearSelection;
SelectUserSelection UserSelectionType=Slot_0A;
SelectUserSelection UserSelectionType=Slot_0B;
SelectUserSelection UserSelectionType=Slot_1A;
SelectUserSelection UserSelectionType=Slot_1B;
AddToSelectionXY UpperLeftX=19 UpperLeftY=121 LowerRightX=25 LowerRightY=121;
ExpandSelection;
BlockSelection PrintUnblockedPorts=False Prefix=RBB_Blocker
  BlockWithEndPips=True SliceNumber=0
  NetlistContainerName=CI_blocker;
GenerateXDL FileName="C:\WORK\Students\Anders\CustomInstructions\start\CI_blocker.xdl"
GenerateXDL FileName="%STATIC_DIRECTORY%\CI_blocker.xdl"
NetlistContainerNames=CI_blocker
IncludePorts=False
IncludeDummyNets=True
IncludeDesignStatement=False
IncludeModuleHeader=False IncludeModuleFooter=False
DesignName=_XILINX_NMC_MACRO
SortInstancesBySliceName=False;

# generate UCF prohibit constraints for the partial region
# we continue with the last selection
PrintProhibitStatementsForSelection ExcludeUsedSlices=True
FileName="%STATIC_DIRECTORY%\GoAhead_constraints.ucf"
Append=True CreateBackupFile=False;
Bibliography


Field programmable gate arrays, FPGA ’11, pages 41–44, New York, NY, USA, 2011. ACM.


