NTNU
Norwegian University of
Science and Technology

Prototyping a Reconfigurable Architecture suitable for Multitasking

Kjetil Wathne Oftedal

Master of Science in Computer Science
Submission date: July 2010
Supervisor: Gunnar Tufte, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Reconfigurable computers offers a large speed-up compared to traditional processor based computers. The speed-up in such machines is a result of including customized hardware for time consuming computation. Although reconfigurable computers offer a considerable speed-up when the reconfigurable parts are configured to a specific computation, the concept requires that the reconfigurable parts of the machine change when the problem changes. In most of today's computers a multitasking approach is taken to utilize the computation power. To include reconfigurable hardware in a multitasking machine the reconfigurable parts must be able to change according to the requirement of each task. Such change of configuration requires effective switching between configurations.

In "Multitasking on a reconfigurable computing system" an architecture for a system capable of integrating reconfigurable computation with a multitasking system was proposed. To further investigate the proposed concept it is required to take the abstract description to a detailed level that can produce a working prototype. To be able to move to a detailed level that can be modeled and implemented in existing FPGA technology the computational reconfigurable building blocks must be defined together with logic that can meet the requirement for reconfiguration in a multitasking environment.

The goal of this thesis is to propose computational building blocks together with a reconfiguration system that can support the architecture proposed in "Multitasking on a reconfigurable computing system" at a level that can be prototyped and modeled.

Assignment given: 18. January 2010
Supervisor: Gunnar Tufte, IDI

Abstract

Integrating reconfigurable computing hardware into general purpose computers offers promise of performance improvement. General purpose computers allows for a large amount of multitasking, as such, the reconfigurable hardware integrated into such a system should also support multitasking. This requires a low overhead reconfiguration method that supports preemption of tasks running on reconfigurable hardware. To investigate methods that can integrate reconfigurable hardware into a multitasking machine an architecture for a reconfigurable device is proposed. In this work the proposed architecture is taken to prototype level. This includes a definition of the computational properties of the basic reconfigurable blocks, a reconfiguration method that can fit within the requirements of multitasking, a configuration format that allows for backwards binary compatibility, and support for rudimentary control flow. The resulting prototype system has been tested and evaluated.

Preface

This thesis was written at the Department of Computer and Information Science, Norwegian university of Science and Technology. I would like to thank my supervisor Dr. Gunnar Tufte for his advice and support. I would also like to thank my friends in the Computer Architecture group at IDI, especially those whom I have shared an office with for the past year.

Contents

List of Figures

List of Tables

List of abbreviations

ALU Arithmetic Logic Unit

ASIC Application-specific integrated circuit

BNF Backus-Naur Form

CAL Configurable Array Logic

CISC Complex Instruction Set Computing

CLB Configurable Logic Block

CPU Central Processing Unit

DMA Direct Memory Access

DSP Digital Signal Processing

FIFO First In First Out

FPGA Field Programmable Gate Array

FLEX Fast Lexical Analyzer

FF Flip Flop

IO Input/Output

LSB Least Significant Bit

LUT LookUp Table

MAC Multiply ACcumulate

MD Multiply Divide

MIPS Microprocessor without Interlocked Pipeline Stages

MSB Most Significant Bit

MUX Multiplexer

PE Processing Element

RAM Random Access Memory

RC Reconfigurable Computing

ROM Read-Only Memory

RPU Reconfigurable Processing Unit

RISC Reduced Instruction Set Computing

SPARC Scalable Processor ARChitecture

SRAM Static Random Access Memory

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuits

WAP Wavefront Array Processor

YACC Yet Another Compiler-Compiler

Chapter 1

Introduction

Reconfigurable hardware allows for the potential speedup of applications, as the hardware can be tailored to increase the execution speed of specific operations the application performs. This is referred to as reconfigurable computing. The speedup is achieved by removing generality in the hardware used to perform the computational intensive parts of the application.

Less general circuitry can employ several techniques to outperform a conventional general purpose processor. It can perform operations that operates on data widths that better suit a specific application. Constants can be integrated into the circuitry to increase the speed of said circuitry, in addition to reducing the required hardware resources. Application specific pipelines of considerable depth can be implemented which would increase the throughput of the hardware.

On the other hand it is hard to implement an entire application in hardware. Hence, the computational intensive parts should be mapped to reconfigurable hardware, and the parts that is dominated by control flow should be run on a conventional general purpose processor. Further, it must be decided how reconfigurable hardware should be allocated to applications.

Adding reconfigurable hardware for each application that can utilize such hardware to a computing system might be an option. However, this is unlikely to be effective in terms of cost and area consumption. It is therefore appealing to reuse the same hardware resources between applications. This requires that the hardware is dynamically reconfigurable, which is the ability to reconfigure hardware after it has been initialized with a default configuration. Another requirement is that the reconfigurable hardware is general enough to support the computations the different applications want to execute.

Allocating the reconfigurable hardware to a single application at the time is a simple solution that allows hardware reuse. Unfortunately, this matches the usage patterns of modern computing systems poorly. When an application is allocated the reconfigurable hardware, and will perform a large amounts of computations with it, the hardware will be monopolized by a single application for extended periods of time. In a multitasking environment, other applications might then be forced to wait until this application terminates. This is unacceptable as this

wait period might eliminate any performance that could be gained by running on reconfigurable hardware.

Performing multitasking in a manner that resembles the general purpose processor approach is appealing. This would require that the state of the reconfigurable hardware is stored in memory, and then reconfiguring the hardware to suit the needs of another application. Unfortunately, this is not feasible. The reason for this is that the time required for reconfiguration the most common reconfigurable hardware, that is the Field Programmable Gate Array (FPGA), is prohibitively high. This is summarized in the following quote:

> Studies estimate that, in some cases, reconfiguration time alone occupies approximately 25 to 98 present of the the total execution time of a reconfigurable computing application.
> - Reconfigurable Computing: The theory and practice of FPGA-based computing [HD07]

There is also no standard method for retrieving and restoring state of such a device. A method for retrieving and restoring the state of a FPGA was attempted by examining the configuration stream of a FPGA in *Preemptive Multitasking on FPGAs* [LMSS00]. The idea was to filter out the state of the readback bitstream from an FPGA, and inserting this state in the configuration bitstream when the task was to be resumed. The problem with this is that the state filtering and configuration splicing would require knowledge of the FPGA configuration format. This format is unlikely to be the same between FPGA families. In addition the filtering and splicing operation would consume computational resources and could increase the already prohibitively long configuration time.

Another possibility is dividing the available reconfigurable hardware into smaller configurable regions, and allocating these regions to applications. The applications would then be able to use the reconfigurable hardware in parallel, and in effect achieve true multitasking. These smaller regions will also impact the configuration time. As a smaller region contains less hardware resources the configuration time would be reduced. The cost of this is not surprisingly that the available area for a single configuration is smaller.

If an application cannot fit all the desired computations within one region, it must use several. Then the problem that this method was trying to avoid might reassert itself, as a single application might require all the available regions. A similar problem might also occur if there are more applications trying to use reconfigurable hardware than there are regions available. Then it would still be required that a the reconfigurable hardware support retrieving and restoring the state of a running configuration if equal access to the hardware resources is to be achieved.

Hence, to achieve multitasking on reconfigurable hardware it is important that a low overhead reconfiguration method is utilized. One such method is presented in *A Time-Multiplexed FPGA*. Several configuration is stored on-chip. These configurations a distributed throughout the reconfigurable hardware in such a manner that the active configuration can be switched instantaneously. However, the extra storage required reduces the amount of hardware that contributes to computation. When there are more applications requiring the use of reconfigurable hardware

than there are room for on-chip, additional challenges occur. The period required for a reconfiguration is still high. However, in terms of hardware utilization this can be masked, as a configuration can be loaded into one an inactive storage area, while another configuration performs computation.

These existing approaches are general in nature, and can be used in any hardware system. However, if the reconfigurable hardware is integrated into a general purpose computer, some additional considerations can be made. General purpose processors perform multitasking by allocating time slots to different tasks. This gives the tasks some amount of equality in terms of computation time available to each task.

However, letting a task occupy the entire time slot that has been allocated to it might reduce processor utilization. The cause of this is usage of virtual memory systems, and Input/Output (IO) operations performed against slower storage units. The processor is much faster than the storage unit. As such, the processor would sit idle whenever a task performs IO operations.

Switching task when an IO operation is performed is likely to increase processor utilization, as another task can use the processor while the IO operation is being performed. The utilization reduction caused by virtual memory system is very similar. This is caused by page misses, which occurs when some task tries to access a memory page that has been moved to secondary storage. In fact, this also causes IO operations to be performed, when this page is retrieved from secondary storage.

This problem is not exclusive to general purpose processors in a conventional computing system. If reconfigurable hardware that operates on memory is integrated into computer, that uses a virtual memory system, this hardware would also be affected. A low overhead reconfiguration technique can therefore increase the utilization of reconfigurable hardware, as other tasks can be allowed to use the hardware, when the current task triggers an IO operation.

Increasing the hardware utilization alone might not be enough. Using the same time slot approach as general purpose processors on reconfigurable hardware will also give more predictable response times for the tasks utilizing such hardware. The effect of this is the harmonization of scheduling on general purpose processors and reconfigurable hardware. As such, the existing scheduling principles that have been developed for general purpose processors could be used with reconfigurable hardware to some extent.

> The problem of multi-tasking in a dynamically reconfigurable RC context is largely unsolved.
> - High-Performance Embedded Architecture and Compilation Roadmap 2007 [BLM+07]

The preceding quote illustrates that this harmonization might not be trivial, as the multitasking part of reconfigurable computing is unresolved. Some approaches to this multitasking has been presented already. These approaches have strengths and weaknesses, and the basis for them is the reconfiguration overhead that exist in reconfigurable hardware.

None of these approaches have taken the characteristics of existing computer architecture into account in the design. This thesis will present a low overhead

reconfiguration technique that relies on overlapping the computation of two tasks. The hope is that this low overhead technique will allow for multitasking on reconfigurable hardware integrated into a mainstream computer architecture.

It has been attempted to implement the support hardware required for this reconfiguration technique, to prove that it is a feasible solution. An example reconfigurable hardware core that utilizes this reconfigurable technique has been implemented. This hardware core operates as semi-autonomous processing unit, that supports a range of computational operation combinations. One of the more interesting ones being a form of limited control flow. That is the ability to perform rudimentary branches.

As the design target is integration into a existing computer system, preserving some of the characteristics of computer programs has been attempted. The most notable is binary compatibility between device revisions.

1.1 Thesis outline

Chapter 2: This chapter contains important background material and a brief presentation of concepts parts of the resulting architecture can be related to or is based upon.

Chapter 3: Here the architecture and considerations around the implementation features is presented.

Chapter 4: The implementation of the device has been performed in two stages, to demonstrate binary compatibility. This chapter presents the first system version.

Chapter 5: Here the second system version is presented. The first version is extended with the ability to perform rudimentary branching.

Chapter 6: Support software is presented in this chapter.

Chapter 7: Here the testing procedures is outlined. In addition the results of synthesizing the hardware are illustrated.

Chapter 8: This chapter contains a discussion of the implemented hardware and concepts. Additionally the conclusion and future work is presented.

Appendix A: This appendix describes the hardware implementation of a combined multiplication and division unit.

Appendix B: This appendix contains detailed data of the configuration bitstream used, and layout of important system interfaces.

Appendix C: Here the grammar for the system assembler is shown.

Appendix D: This appendix contains an overview of the tests performed.

Appendix E: This appendix contains the code required for system reproduction.

Chapter 2

Background

2.1 History

2.1.1 Fixed+Variable Computer

In the late 1950s/early 1960s, Gerald Estrin observed that even though faster computers were being built, the number of discovered problems that could not be solved were increasing. The limiting factor being the amount of time required to solve the problem. This time limitation was caused by several factors; the cost of using a computer for an extended amount of time could be too high compared to the value of the solution. If there was some sort of real-time demand on solving the problem, the solution might not be valid or useful, when the computation was eventually completed. Another problem emerges if the computation requires a larger amount of time than the computer's mean time between component failure [Est60].

To overcome these problems, special-purpose hardware could be built that achieved significantly higher performance than that of a general purpose computer. With this in mind, Gerald Estrin formulated the following goal for the *Fixed+Variable Computer*:

> To permit computations which are beyond the capabilities of present systems by providing an inventory of high speed substructures and rules for interconnecting them such that the entire system may be temporarily distorted into a problem oriented special purpose computer.
> - Gerald Estrin [Est60]

The *Fixed+Variable Computer*, as the name suggest, consists of two parts: a fixed and a variable part. The fixed part would typically be a general purpose computer. This would handle IO in addition to providing a stable programming interface, so computer scientists would not be discouraged from developing complex programs by a continuously changing intruction set.

The variable part would be a collection of various components optimized for certain operations. The interconnection between these components and the fixed

part of the system would be decided by a set of rules. These rules would be problem specific. The resulting structure would increase performance when solving certain problems defined by the currently used interconnection.

An attempt at realization of such a computer was described in *Parallel processing in a restructurable computer system* [EBTB63]. Restructuring based upon electric switching and physical reconnection was considered. Electrical switching did not allow for physical relocation of components, and was limited with regard to the interconnect changes that can be made. Therefore physical reconnection of components was chosen. Physical reconnection was implemented by the use of printed circuits and a module based system.

Although the *Fixed+Variable Computer* could achieve performance improvements the rapid increase in computational power of general purpose computers hampered the development. The idea on the other hand would live on in the research field of reconfigurable computing.

2.1.2 Reconfigurable Computing

Reconfigurable computing gained momentum by the use of technologies derived from FPGAs [CH02]. The FPGA was commercialized by Xilinx in the 1980s. The purpose of the FPGA was to be a electrically programmed device, that had the ability to perform complex computations internally. As the FPGA are generally reconfigured by the use of electronic switching they are a child of the development path originally dismissed as being too limiting by Gerald Estrin [EBTB63].

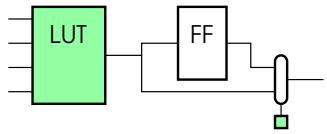

Figure 2.1: Simple logic block

In figure 2.1 a simplified version of a FPGA building block is shown. The block consists of a LookUp Table (LUT), a Flip Flop (FF) and a by-pass Multiplexer (MUX). The LUT is primarily used to emulate logic function. The FF can be used to store the result of this logic emulation. The by-pass MUX allows the result of the logic emulation to pass directly to the output of the logic block. This output can be used as an input in a different logic block and thus be used to implement logic functions that will not fit in a single LUT.

FPGA reconfiguration is often performed by changing internal Static Random Access Memory (SRAM). This internal SRAM is shown in green in figure 2.1. When the SRAM is changed the logic function emulated by the LUT is changed. In addition the interconnection network in the FPGA is changed, as represented by the MUX in figure 2.1.

Even though the early FPGAs became popular with the reconfigurable computing researchers, the FPGAs were severely limited in the amount of logic gate emulation that could be performed in a single FPGA [HD07]. The computational

power of general purpose computers were also increasing at a rapid rate. This caused research to be focused on constructing systems with multiple FPGAs and problems that could not be efficiently solved with word-oriented general purpose processors.

Spatial computing is one of the reconfigurable computing approaches to solving problems faster than general purpose processors. General purpose processors rely on reusing the same execution units. Whenever a new instruction is to be executed the computation performed by a execution unit is changed, to suit this instruction. As the execution units is reused in time, this referred to as temporal computing.

Spatial computing on the other hand has execution units with fixed functionality at fixed location in space. As data flows through these units computations are performed. One of the simplest implementations of this is pipelining of operations. As each stage perform different operations on the data as it pass through the pipeline.

One of the more interesting devices of this era was the Configurable Array Logic (CAL), which was later developed into the Xilinx XC6200. This FPGA consisted of rather simple logic cells, however each logic cell could be dynamically reconfigured. This is the process where an individual cell can be reconfigured while the FPGA is operating. The ability to do this did require that the users of such features get complete access to the device configuration specification, and specification of the internal structure of the device.

Such information was usually kept secret by the FPGA manufacturers, as they feared that cheap clones of the chip could be manufactured. The information available about the XC6200, and the backing provided by Xilinx, made it popular in the reconfigurable computing research community. The available information gave the research community the ability to experiment with new applications and tools for reconfigurable devices.

2.2 Coupling

When a Reconfigurable Computing (RC) unit is to be integrated into an ordinary microprocessor based computing system, the point of connection must be decided. Where in the system the connection is made in relation to the microprocessor is referred to as coupling. In figure 2.2 some possible connection points are shown in relation to the existing memory hierarchy. The RC units are represented with green, and the existing computing system with gray.

When the RC unit is integrated as a part of the processor, or as a co-processor, the coupling in the system is referred to as close. With such a closely coupled system, large amounts of the existing memory system is shared between the microprocessor and the RC unit. The degree of sharing might be so high that even the microprocessor's register file is shared. In the other end of the coupling range is loosely coupled, in which the RC unit is connected to the system over an external bus.

The coupling degree determines several key parameters of the communication between the RC unit and microprocessor, such as communication speed, latency

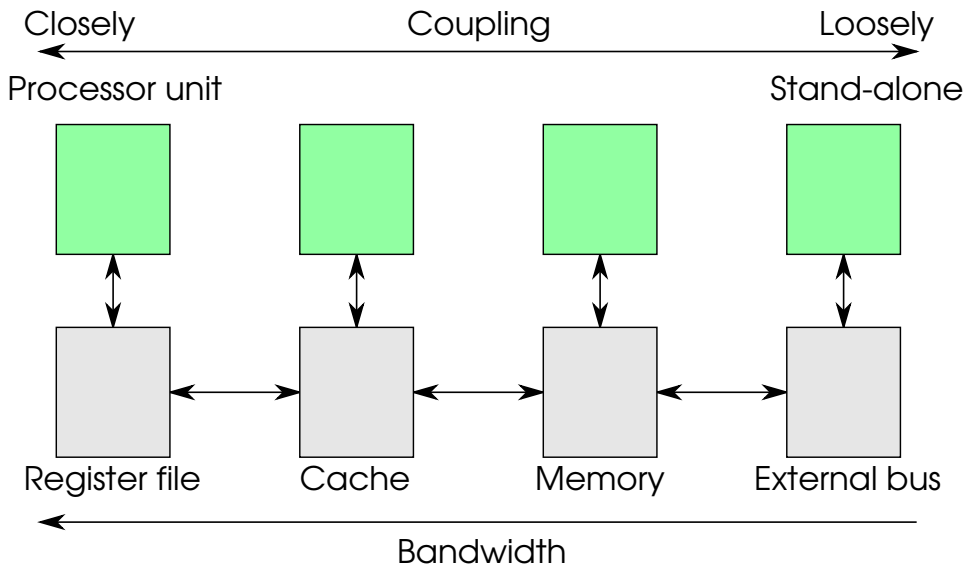

Figure 2.2: Coupling

and control of the RC unit. At the bottom of figure 2.2 it is indicated that the bandwidth decreases when the degree of coupling decreases. It should be noted that this is the bandwidth between the microprocessor and the RC unit, not the RC unit's memory bandwidth as it might include a separate memory system. This might be required if the RC unit needs large amount of bandwidth, and the existing memory system is not optimized for the memory access pattern of the RC unit. However, this kind of coupling would decrease the amount of control the microprocessor can assert upon the RC unit as the information flow between them is limited by both bandwidth and latency.

2.3 Granularity

Granularity is an important property of a reconfigurable device. This refers to the operational width or rather the complexity of the reconfigurable logic blocks. The spectrum of granularity is shown in figure 2.3, with the part of the device that is reprogrammed during reconfiguration highlighted.

Fine granularity devices typically consists of basic logic gates with reconfigurable interconnect. The logic gates are often of NAND type, which can be used to implement all other binary logic functions. The major drawback of fine granularity devices is that the interconnect requirements are high compared to the implemented logic functions [MB07].

The logic blocks of a medium granularity device can perform logic operations with several inputs and outputs. As shown in the middle of figure 2.3, the logic block of a medium granularity device can consist of a LUT with several inputs

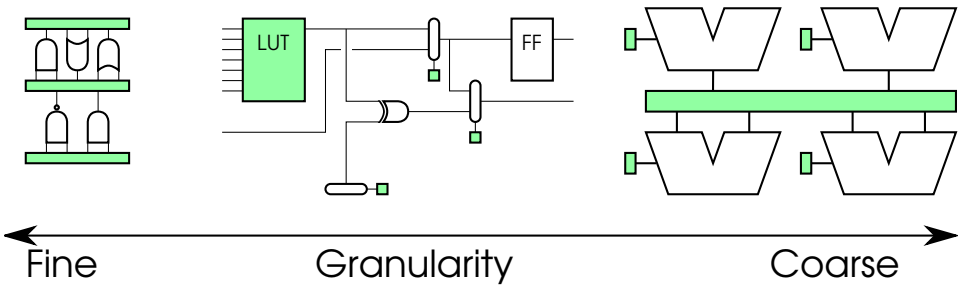

Figure 2.3: Granularity

which is connected through some reconfigurable interconnect to a FF or directly to the block output. Some basic gates are often added to the structure to speed up basic arithmetic operations.

Devices of coarse granularity contain logic blocks that can perform even more complex logical operations. These operations are often arithmetic or logic operations of a somewhat higher level, which are performed on several inputs in parallel. This allows for faster operation as the logic can be optimized for the supported operations. The amount of configuration data is also reduced as the operation that is to be performed only needs to be specified once for all the inputs to a logic block. The drawback of coarse granularity is that generality is lost as the number of implementable logic functions is reduced. Arithmetic Logic Unit (ALU), as shown in figure 2.3, is typical logic block in a coarse grained architecture. Only a small amount of configuration is needed for each ALU. The amount of configuration needed for the interconnect is also reduced. The reason for this is that multiple wires can be grouped together. The size of this group would equal the operational width of a ALU, and the group can then be routed by a small amount of bits.

Such a reduction in the configuration size might be beneficial for multitasking as the amount of configuration that must be loaded is reduced.

2.4 Multitasking

Multitasking on general purpose processors was conceived to allow multiple programs to share system resources and achieve higher system utilization. If a only a single program was allowed in a computer at any given time the Central Processing Unit (CPU) would be idle whenever IO was performed. Because IO is rather slow compared to CPU speeds, the CPU utilization would be low. By allowing several programs to be present in the system simultaneously the operating system could switch to another program when one program starts an IO operation. Hopefully one of the other programs will be ready to use the CPU, and thereby reducing the idle periods of it.

This approach would lead to higher utilization. However, if one program does not perform any IO, this one program would always be active and the other programs present in the system would never be able to use the CPU. This is referred

to as starvation. To mitigate this problem, time sharing is used. All programs that can use the CPU are said to be in the *Ready* state. Programs waiting for IO are in the *Blocked* state. If more than one program is in the *Ready* state the CPU resources must be divided between them. The programs are given time slots on the CPU and a timer generates interrupts that invoke a scheduler that switches to the next program when the current time slot has ended.

Some programs might have higher priority than the rest and must be allocated a larger amount of CPU time. This can lead to situations were it is required that the current running program must yield to a higher priority program, that becomes *Ready* as a result of the completion of a IO operation. This form of multitasking is called preemptive multitasking, and the process that must yield is said to be preempted.

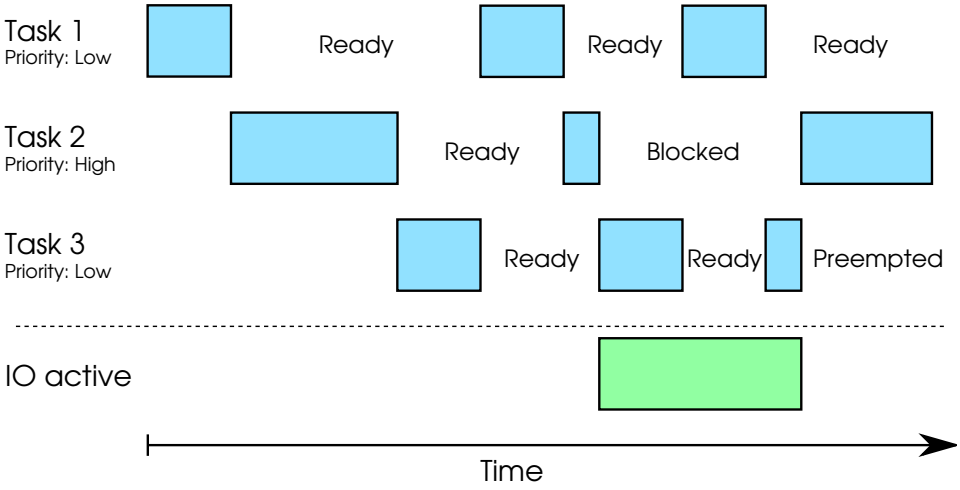

Figure 2.4: Multitasking

Figure 2.4 illustrates the multitasking process. Three tasks are present in the system, with various priorities. The blue blocks above the dotted line represent periods when the corresponding task is using the CPU. The green block below the dotted line represents activity in the IO system. Task 1 is the first task allowed to use the CPU. After a while a timer generates an interrupt and the current running task is switched to Task 2. This task has a higher priority and is allocated a larger amount of CPU time. After a period of time Task 3 is reactivated, and subsequently Task 1 is activated again, and so on.

During the second time slot allocated for Task 2, the running task performs an IO request. This task then enters the *Blocked* state and is deactivated, and the scheduler continues to Task 3. As Task 2 is *Blocked* it is not activated at all, and the running task switches between Task 1 and Task 3. After a while the IO operation started by Task 2 completes, and an interrupt is generated. At that time Task 3 is the current running task. Task 2 is now in the *Ready* state and has higher priority than the current running task. Based upon this information and

the period from the last activation of Task 2 the scheduler deactivates Task 3 and reactivates Task 2.

Achieving multitasking by reusing the same hardware is also possible with reconfigurable hardware. The simplest approach to this is configuring a FPGA with the required configuration, and the running the task to completion. When the task has finished the FPGA is configured with the configuration required by the next task. Although simple, and hardware effective, it drastically increases latency of tasks not currently running, and can cause starvation.

The major problem with this approach is that the overhead of reconfiguration a FPGA is quite large. Loading a configuration onto a modern Xilinx Virtex 6 FPGA requires $\sim$ 85ms[1] [Xil09].

A solution to this problem is partial reconfiguration of the reconfigurable hardware. Dividing the device into smaller regions and using these independently, reduces the amount of configuration that needs to be loaded. Although this reduces reconfiguration overhead it also reduces the size of configurations. If this approach is to be viable the reduction in reconfiguration overhead must be substantial.

To achieve this efficient overlapping computation and configuration has been attempted by several researchers [JTY+99], [QSN06], [RMC05], [RCG+08], [EAGEG09]. The idea is configuring an inactive region of the device, while others perform computation. The problem is deciding which configuration the inactive region should be configured with. In some cases the next required configuration will be decided by the result of a running configuration. The next configuration is not predetermined under these conditions. A solution would then be to guess which configuration will be required next, and speculative load this configuration. When the wrong configuration is loaded the system will have to reconfigure the region to the correct configuration. When this happens the reconfiguration overhead will be just as large as not speculating at the next configuration at all.

A different method of handling configurations can almost eliminate the reconfiguration overhead. This approach was presented in *A Time-Multiplexed FPGA* [TCJW97]. The reconfiguration overhead is reduced by storing several configurations on-chip. These configurations are spread throughout the reconfigurable fabric, in a manner that stores the part of a configuration near the unit that it will control. Instead of connecting the internal units directly to a configuration it is connected to a MUX. This MUX is in turn connected to several different configurations. The reconfiguration procedure is then reduced to simply switching witch configuration is connected to the internal units. This hardware structure is referred to as a multi-context FPGA.

The overhead of this reconfiguration in relation to time is very low. However, it does require storage for several configurations on-chip. The high speed reconfiguration is also limited to the configurations located on-chip. Combining this approach with the overlapping reconfiguration approach will alleviate this. As an inactive configuration can be switched out, when another is connected to the internal units.

The advantages of time sharing and preemptive multitasking should not be ignored in a reconfigurable hardware context. Such support does require that

[1] Using a 16-bits configuration bus running at 30Mhz

the current state of a running configuration can be saved and restored. When reconfigurable hardware is used in a computing system that incorporates some slow secondary storage system, the state storage and reconfiguration overhead must be lower than the mean IO operation time, if the reconfigurable hardware utilization is to be increased by multitasking.

2.5 Related technology and architectures

This section includes some related architectures and principles. Some of these approaches will find new use in the reconfigurable architecture presented in the next chapters.

2.5.1 FPGA

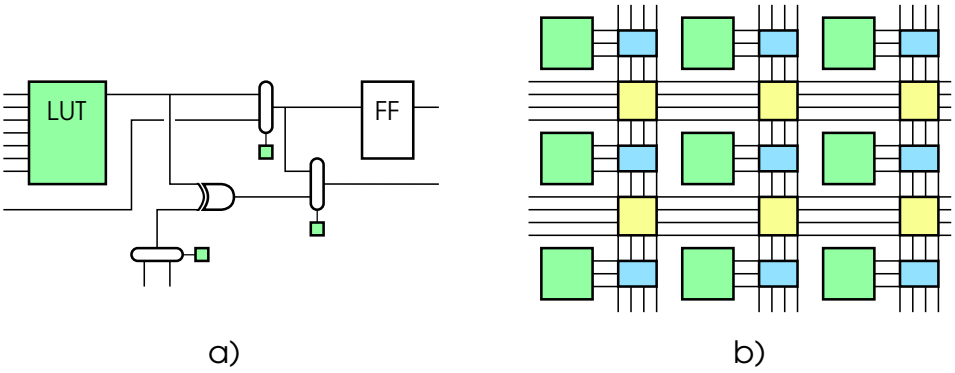

Figure 2.5: Field Programmable Gate Array

FPGA is as mentioned on flavor of reconfigurable hardware. It is often used for logic prototyping, and has even found its way into commercial products due to a fall in chip cost.

In figure 2.5a, a simplified version of the basic reconfigurable block of a FPGA is shown. As previously mentioned this consists of a LUT used for logic emulation. This LUT is often implemented with SRAM in reprogrammable FPGAs. By changing the contents of the LUT the logic function emulated in the LUT is changed.

The LUT is often connected to a FF, which can be used to store the output value across clock ticks. These FFs can often function in a variety of ways. Such, being triggered by an enable signal, having an asynchronous clear and so forth.

In addition some complex gates used for arithmetic operations are often included into the device, to increase the speed of such operations. This is illustrated in figure 2.5a with an XOR-gate. The internal connection between these components is often controlled by MUXes. The setting of these MUXes is controlled by small SRAM cells, and can therefore be reconfigured in the same way as the LUT.

Several such reconfigurable block are grouped together in a Configurable Logic Block (CLB). The internal blocks of a CLB are connected together in various ways. Many CLBs is present within a FPGA. These are illustrated with green, in figure 2.5b. The rest of figure 2.5b illustrate a simplified mesh-based interconnection scheme for a FPGA.

Each CLB is connected to the interconnect via a connection box, shown in blue. This connection box connects the IO ports of the CLB to various interconnection wires. These interconnection wires are connected to switch boxes, shown with yellow in figure 2.5b. In the switch boxes various vertical and horizontal wires are connected together. As such, connections between the connection boxes of different CLBs can be made. Both the connection boxes and the switch boxes are reconfigurable. In a real FPGA the interconnect is richer than the illustrated one. An example of this would be that neighboring CLBs can be connected together directly, without using routing resources. In addition, wire delay between two distant CLBs can be reduced by letting some wires run straight through the switch boxes.

One of the appealing possibilities of FPGA, is to partial reconfigure the device. As the name suggests, this is the possibility to load a configuration into a specified region of the FPGA. This requires that two independent configurations running in the same FPGA does not utilize the routing resources. This would be easy in the simplified illustrated interconnect. However, in a real FPGA the long wires running straight through the switch boxes can cause problems, as this can cause unintended connections between the two configurations.

The independent configurations must also not make use of the fixed hardware structures embedded into the device. These units are added to speed up complex operations, or provide more area efficient structures. Examples of these would be microprocessor cores, and Random Access Memory (RAM). Because of these resources FPGAs can be classified as a mix between coarse grained and medium grained devices. However, they are often referred to as fine granularity devices as they can perform bit wise operations.

2.5.2 Piperench

Piperench is a reconfigurable architecture presented by Seth Copen Goldstein et al. in *Piperench: A Reconfigurable Architecture and Compiler* [GSB+00]. The focus of the Piperench architecture is virtualization of hardware. This can be compared to virtual memory, where the amount of physical memory differs from the amount of virtual memory a program can access. Within reconfigurable hardware it refers to a similar mismatch between the size of the configuration and the amount of configuration the physical device can hold. Piperench uses pipelining to solve this problem. The application is divided into a number of virtual pipeline stages (v) that can be scheduled on a smaller number of physical pipeline stages (p).

One of the advantages of virtualization of reconfigurable hardware, as done in Piperench, is the ability to reuse the same configuration on larger devices as they become available. Traditionally FPGAs require resynthesizing of the design when switching to a larger FPGA. As long as no changes are made to the physical stages

used in a Piperench processor it is able to load a configuration containing any number of virtual pipeline stages.

As devices containing larger amounts of physical pipeline stages become available, throughput will increase as a larger number of the virtual pipeline stages can be active at any time. This increase in throughput related to the number of physical pipeline stages has its limits, and will stop when the number of physical pipeline stages increases beyond the number of virtual pipeline stages.

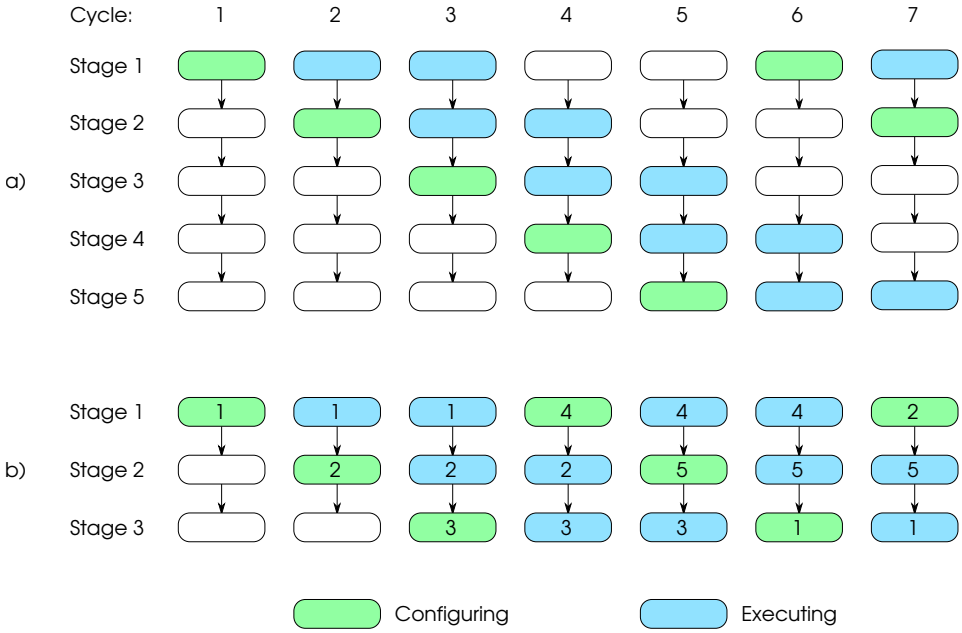

Figure 2.6: Pipelined reconfiguration (adapted from [GSB+00])

Piperench uses a pipelined reconfiguration technique described in *Pipelined reconfigurable FPGAs* [SCMG00]. This process is illustrated in figure 2.6. Figure 2.6a shows the virtual pipeline, and figure 2.6b shows the physical pipeline. When a configuration containing more virtual than physical stages ($v > p$), it is not possible to have all the virtual pipeline stages active at the same time. Therefore run-time reconfiguration is used to simulate a larger device.

The first physical stage is configured with the first virtual stage. When the configuration is completed the first stage starts executing while the reconfiguration system starts configuring the second physical stage with the second virtual stage. The reconfiguration process must complete the configuration of a physical stage as fast as a pipeline stage can produce output to the next stage in the pipeline. In effect the reconfiguration always stays one step ahead of the data propagating through the pipeline. This requires that piperench is able to reconfigure a single stage as fast as computation is performed in a single stage. To achieve this a wide parallel bus is used change to configuration of a pipeline stage.

When there are no more unconfigured pipelined stages in the physical pipeline, the physical pipeline stage containing the oldest configuration is reconfigured with the next virtual pipeline stage. At this point no more data will enter the pipeline, as the first virtual stage will be disabled. Instead data already in the pipeline will continue through the virtual pipeline until reaching the end of the virtual pipeline. When the final virtual pipeline stage has been configured, the first pipeline stage will then be configured into the next physical pipeline stage, and data will again be able to enter the pipeline. It can be shown that this leads to a throughput proportional to $\frac{(v-1)}{p}$ [GSB+00].

This style of reconfiguration has some requirements to the configuration and the underlying architecture. The state of a single stage must be the result of its own state and the state of the previous state in the pipeline. This leads to a requirement that all cyclic dependencies must fit within a single pipeline stage. In addition data can only flow between consecutive pipeline stages. However, it is possible to implement virtual connections between remote stages.

It is required that any virtual pipeline stage must be placeable on any physical pipeline stage. Therefore all physical stages must be identical and all physical pipeline stages must be able to be the first and last stage in the virtual pipeline. This is done to increase efficiency of the architecture, and leads to a requirement that all physical pipeline stages must have the ability to receive external data.

2.5.3 Wavefront array processor

The Wavefront Array Processor (WAP) was described by Sun-Yuan Kung et al. in *Wavefront Array Processor: Language, Architecture and Applications* [Sun82]. This processor is designed for recursive algorithms with local data dependencies.

Such algorithms often present behavior which consists of repeated execution of simple operations with localized data flow in a homogeneous computing network[Sun82]. Localized data flow in this context means that communication in the computing network is limited to the nearest neighbors in this network. These algorithms running on a computing network can induce propagation of computational activity in a way that resembles physical wave propagation. This is called a computational wave. These computational waves can be pipelined by starting a new wave after the current wave have propagated away from the wave origin. This allows for parallel execution of recursion in an algorithm. The actual cause of this phenomenon is locality, regularity, recursivity and concurrency.

WAP is designed to take advantage of the computational wave. The structure of WAP is illustrated in figure 2.7. The processor is organized as a array containing Processing Element (PE)s, which are shown in blue. These PEs are restricted to local communication. A PE can only communicate with its nearest neighbors in the horizontal and vertical directions. The memory modules, shown in green in figure 2.7, are connected to the first row and first column of the PE grid. Therefore memory access is only possible through the PEs in the first column and the first row in the array. These PE are therefore different from the rest of the array. All the interior PE are identical. This is desirable as a regular structure is easier to produce.

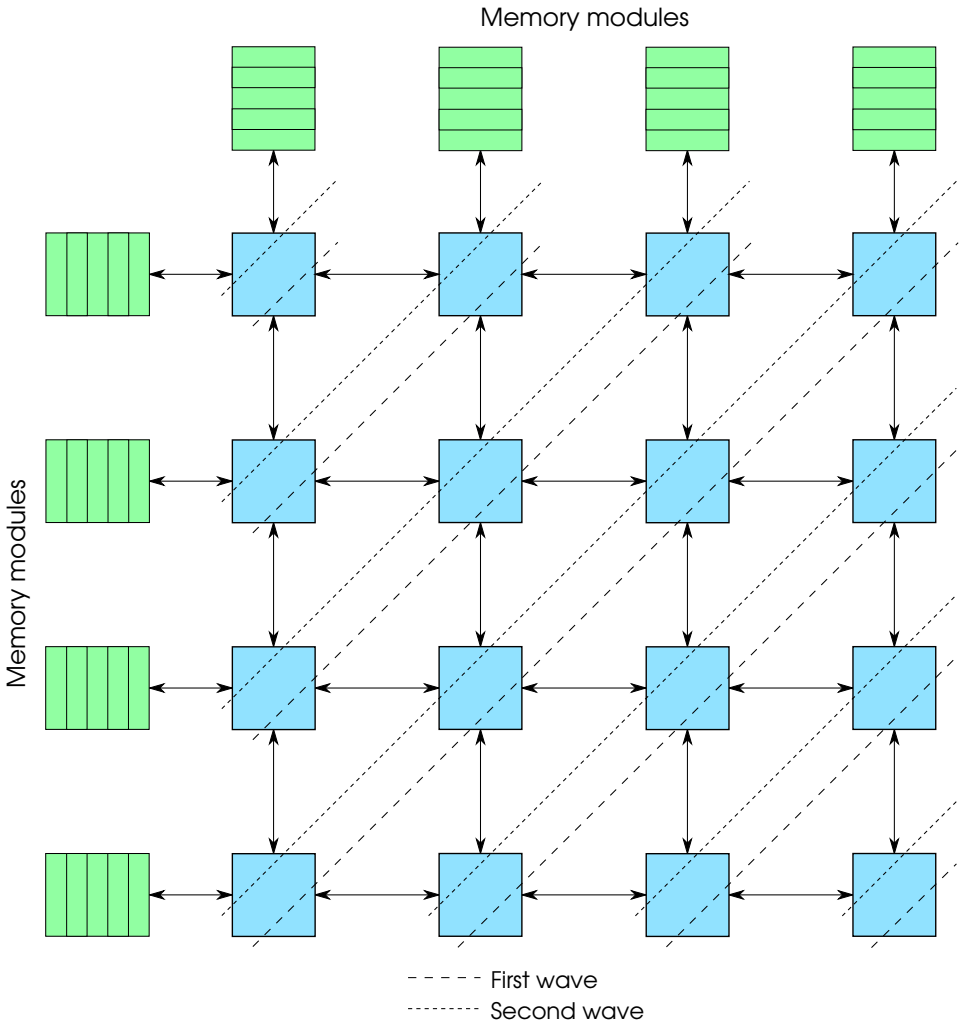

Figure 2.7: Wavefront array processor (adapted from [Sun82])

The processing array does not have a global clock. As such, the need for a low skew clock network is eliminated. However, without a global clock to synchronize communication, the PEs need another form of synchronization during data exchange with neighboring PEs. A handshaking scheme is used to overcome the asynchrony of the PEs and allow them to communicate without risking data corruption. The handshake scheme allows PE to wait for data to arrive on their input ports, in addition to block further communication if the input buffer of a PE is full. Hence the processor is said to be data flow based. Deadlocks in the processing array are prevented by requiring all processing elements to initiate sends before receiving.

Two ways of programming the array was proposed. The PEs in the grid can be programmed individually or an entire wavefront can be programmed. Both programming methods can lead to differences in computation time between wavefronts. This difference in computation time can lead to a situation, where a faster wavefront catches up to a wavefront which has reached a slow stage in the computation. Collision between these two waves is prevented by the blocking handshake scheme.

2.5.4 RISC processors

One traditional approach to improve performance was the usage of complex instructions that would cause a sequence of microinstructions to be executed. This approach is referred to as Complex Instruction Set Computing (CISC). The improvement in performance was caused by the relative difference in memory speed and processor speed. If a complex instruction could cause a large number of operations to be performed on the input data, this would improve performance as the number of instructions fetched from memory would decrease.

The available technology could implement high-speed Read-Only Memory (ROM)s. By storing commonly repeated instruction sequences as subroutines in a ROM, and using the instruction stream to call these, performance could improve. This approach did however require that sensible subroutines would be implemented, and that the gap in performance between processor and memory speed would continue.

However, these assumptions did not hold. As shown in [PD80], only a subset of the available instructions were used by the programs analyzed. Furthermore, a major proportion of the programs consisted of a even smaller subset of simple instructions. That is instructions that can be efficiently run directly on hardware. The cause of this might be found with compilers that were unable to utilize the complex instructions. Additionally, the increased use of caches also narrowed the gap between processor speed and memory speed.

Acknowledging this, several research projects into processor architecture led to a class of devices now known as Reduced Instruction Set Computing (RISC). The common factor for these processor was the elimination of the complex instruction translation. Simple instructions that were directly implementable in hardware was utilized.

Such simple instruction eliminated the need for a complex control unit, and a subroutine ROM. The Berkeley *RISC I* project went even further with simplifica-

tion of the processor [PS81], as the goal of the project was to create a single chip processor, with a reasonably sized on-chip cache.

The instruction formats should be of equal size. The instruction encoding is probably not the most size efficient. However, the fixed size instruction simplifies instruction fetch, as instruction will always be aligned in memory.

The resultant architecture should be able to execute one instruction per cycle, which puts a limit on instruction complexity. This does increase the size of programs as more complex operation must be implemented by the programmer. Nevertheless the resultant processor can have greater efficiency as the instructions are easier to decode.

Only load and store instruction should access memory. Hence, all other instructions should operate on registers. This was done to force the allocation of variables to registers, and not to memory addresses. When the variables are allocated to internal registers, and not to memory addresses the amount of memory accesses is reduced. The cause of this is that intermediate results are not stored back to memory, when they are likely to be accessed again soon.

The Berkeley RISC processor was eventually commercialized by Sun Microsystems under the name Scalable Processor ARChitecture (SPARC). Another notable RISC research project that was commercialized was the Microprocessor without Interlocked Pipeline Stages (MIPS) developed at Stanford university.

A major difference between the Berkeley RISC and the Standford MIPS was in their pipeline implementation. While Berkeley RISC relied on hardware to detect dependencies between instructions in the pipeline, and perform forwarding between pipeline stages. The MIPS relied upon compiler technology to reorder instructions and thereby remove hazardous dependencies [Pat85].

Chapter 3

System overview

In this chapter a brief overview of the implemented device will be given. First the integration point of the device is elaborated upon. Following this a wave based method for performing computation and reconfiguration, that allows for multitasking, is described. Then binary compatibility between device revision is elaborated upon. After this elaboration follows a description of a possible IO system for the device. A description of the computational properties of the device follows this. Finally an overview of a reconfiguration system that supports the low overhead reconfiguration method is given.

3.1 System coupling

During the design stages it was required to roughly specify the kind of coupling the finished system might use. If the device was designed as unit suitable for integration into a general processor it would be tied to the running task of that processor. This would probably reduce the utilization of a reconfigurable unit as not all tasks would make use of such hardware, and it would be left idle when such tasks were running.

Therefore the device was designed as a separate device that would take commands from a general purpose processor. This would allow for scheduling of the reconfigurable device to be performed by an operating system. This independence does come at a cost, which is the increased communication latency and reduced control over the device.

In figure 3.1 some possible interconnection schemes are shown. Figure 3.1a illustrates the simplest connection type. The Reconfigurable Processing Unit (RPU) is connected to the same bus as the regular CPUs and share the main memory. The CPUs can send commands directly to the RPU. Although simple to implement it requires that the existing memory system meet the bandwidth and latency requirements of the RPU without causing starvation in the CPUs.

If the existing memory system is unable to satisfy the memory system requirements imposed by the RPU, one could consider the approach shown in 3.1b. A

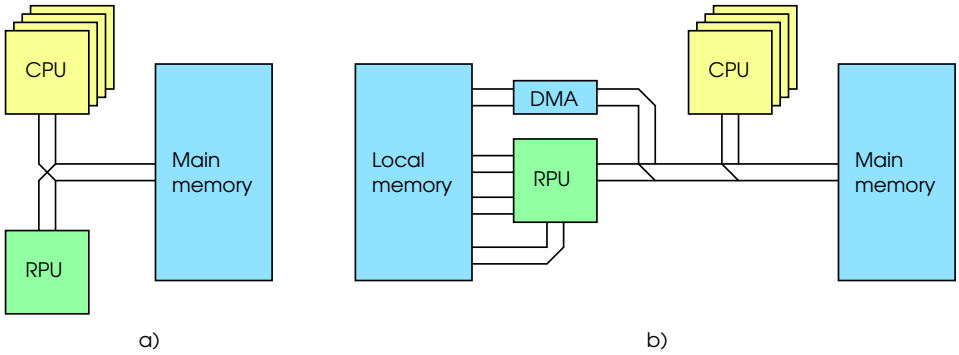

Figure 3.1: System overview

separate memory system, labeled *Local memory* in the figure, is implemented to suit the needs of the RPU. Data which the RPU is to perform computation on is explicitly copied into the devices private memory with a Direct Memory Access (DMA)-like device. The command interface of the RPU is still connected to the existing infrastructure. However, one should acknowledge that this does not eliminate the bottleneck. It will either exists as a limit upon the maximum amount of memory the unit can perform computation on, or it will consume data faster than it can be loaded from main memory unless it reuses data elements already loaded.

3.2 Device overview and properties

Multitasking on a reconfigurable unit requires that the device be carefully designed to allow low overhead reconfiguration. In this section a interconnect style that allows for a high degree of overlap between tasks is introduced. In addition, solutions for backwards compatibility is presented, and the IO system of the device is elaborated upon.

3.2.1 Device dataflow

The RC device is first divided into subunits, called cells. A cell has some amount of inputs and some amount of outputs. The internals of a cell, will for now, be left as a black box that performs some computation on the input data. The result of this computation will be put on the cell output ports.

When the RC device is divided into cells, which perform some computation before passing data to other cells, there is a worst possible data dependency case. This worst possible case is when all the cells depend on input data from other cells. Only cells that solely rely on external input can start performing computation. When these cells are done with their computation and transfers data to their neighboring cells, all the neighbors that now have all their data dependencies met can start computation, in addition to the originating cells that now can receive more external data.

This will cause computational activity to propagate through the RC device following the interconnection network. The computations performed on input data will be highly pipelined as the originating cells can start computation on their next data set as soon as data is transferred to neighboring cells, which also can start computation if all their data dependencies are now met.

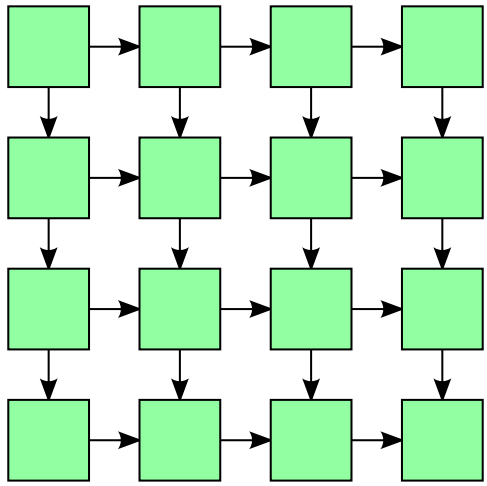

Figure 3.2: Nearest neighbor dataflow

The multitasking approach proposed will be designed for this worst possible case. The basic RC device is divided into cells. Each cell have two input ports and two output ports. The cells are organized in a grid with nearest neighbor communication as shown in figure 3.2.

Cells along the edge of the structure can communicate with external units. This organization is somewhat similar to WAP. The main difference is that the communication is further restricted to be unidirectional, while WAP uses bidirectional communication. The unidirectional communication allow us to define a worst case data dependency case, that all other communication patterns allowed in the device must be a subset of. This is the worst possible case previously outlined.

For the structure shown in figure 3.2 the worst case data dependency case would then be that all cells depend on both their inputs. Thereby all cells would depend on the outputs of the cell in the upper left corner of the grid. This is the only cell without inputs from other cells in the grid, and only relies on external data as both its input ports face toward the outside.

This cell will be the origin of a computation wave. When it has performed its first computation it will transfer data to the cell below and the cell to the right. These cells can now perform computation as their data dependencies are now satisfied. In addition the origination cell can now receive more data, which will allow the three cells to compute in parallel.

This process will continue in until all the cells in the grid are computing. This communication pattern is somewhat similar to that of WAP where computational

activity propagates through the array in waves.

3.2.2 Device reconfiguration

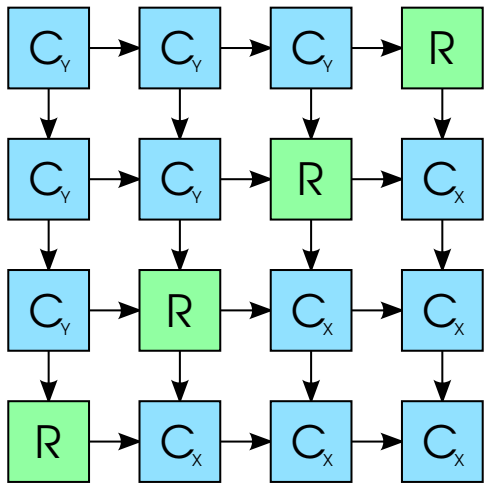

Figure 3.3: Reconfiguration wave

Multitasking in the outlined RC device can be implemented by considering the dataflow shown in figure 3.2 and the worst case scenario presented. If the grid was performing computation and the dataflow into the cell on the top left was cut, this cell would perform computation with data already received, transfer its results to neighboring cells and then become idle. This cell could at that time be reconfigured with a new configuration, while all other cells in the grid are performing computation on data that has already passed through the cell under reconfiguration into the grid.

When the reconfiguration has completed, new data can enter the newly reconfigured cell. The cells dependent on the newly reconfigured cell will run out of data when they have completed their computation on the data transferred before the reconfiguration process started, and will then stall. When this stall occurs, these cells are ready for reconfiguration.

This will cause a wave of reconfiguration to pass through the grid as shown in figure 3.3. In this figure the cells labeled C_x are using configurations belonging to task X, while cells labeled C_y belong to task Y. Cells being reconfigured are labeled with R.

The effect of this reconfiguration technique is that data drains from the cells in front of the reconfiguration wave. This reduces the amount of data that needs to be stored as state within the grid when configuration are switched. The advantage of this technique is the large amount of overlapping between two tasks during reconfiguration. This style of pipelined reconfiguration is much the same as the implementation of virtual pipelines in the Piperench architecture.

There is one important limitation one must be aware of. The reconfiguration wave must travel at the same speed as data through the cell structure. If the reconfiguration wave moves slower than the data one could end up with data being transferred into a cell under reconfiguration and data would be lost. Or if the reconfiguration wave moves faster than the data, it would overtake the computation wave in front of it and start reconfiguring cells before computation has been completed.

3.2.3 Configuration scalability

The device should be able to load configurations that are of a different size than the device. This will allow for some backwards compatibility as the production of larger devices becomes possible. It is noteworthy that only configuration sizes that are smaller than the actual grid size is considered here, as the opposite is referred to as hardware virtualization and is a topic that is briefly touched upon by *Multitasking on a reconfigurable computing system* [Oft09].

Allowing a smaller configuration to run on a device with a larger amount of cells becomes problematic when there is an external feedback loop in the configuration. This external feedback loop might take the form of a buffer or a First In First Out (FIFO), that the device both writes to and reads from. Such a buffer must be large enough to accommodate the task startup period. During this period the cell generating data to the feedback loop will not be active as it has not been configured yet. In most cases the consuming cell will start reading data before data is produced, this buffer must contain default data that the grid can safely consume, without generating erroneous output or state. The problem related to this is that the amount of time the data requires to leave the grid and then be enter the grid again must be constant regardless of the actual grid size.

Grouped scaling approach

One solution to this problem is loading the smaller configuration into a corner of the actual device grid. This is illustrated in figure 3.4. Colored cells represent cells that contain the undersized configuration. With this kind mapping of an undersized configuration to a device there can be a considerable amount of cells between the edge of the configuration and the edge of the grid. Data that leaves the configured cells must be able to reach the device edge in the same amount of time as when the configuration perfectly matches the device. There are several implementation options that will allow the data to skip past the inactive cells.

A pass-through bus can be embedded into the device grid. The outputs of the cells along the edge of the configured region would be attached to this bus. Data leaving these cells and entering the bus would then be forwarded to the edge of the actual device. The problem with the naive implementation of this approach is that the resulting bus would be a rather long parallel bus. High-speed parallel buses can suffer from skew and crosstalk, depending on their implementation. To mitigate this a serial bus could be used, which would allow for higher bandwidth. However, this would require serial converters to be embedded into the grid. If every smaller

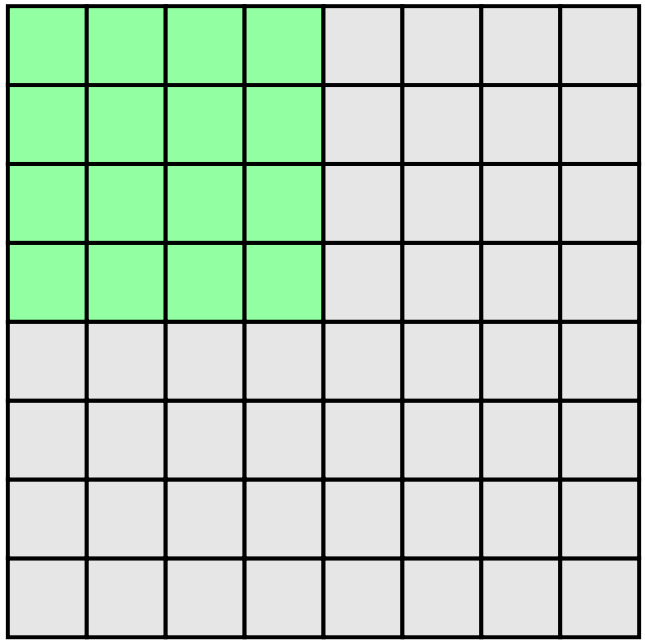

Figure 3.4: Grouped scaling

configuration size is allowed such converters must be embedded between every cell. The increased chip area required for this might be unacceptable. One solution to this is adding a requirement to the configuration sizes allowed. A configuration must match the size of a previous manufactured device size. Newer devices must be required to be a multiple of previous device sizes. Then there will be a clearly defined positions in the grid were a configuration might end, and only at these borders will serial converters be embedded.

At the inputs of every cell a MUX can be added that will bypass all input registers and connect the outputs directly to the inputs. This will allow for the reusage of the existing interconnect, and might be more area effective. However, if a small configuration is used on a large device there will be a large amount of MUXes that data must pass through in order to reach the device edge. This might cause even worse problems related to timing and delay than the pass-through bus.

All the mentioned approaches suffer from the same weakness. If a large configuration is running and the device is reconfiguring to a smaller configuration, multiple use of the same hardware will occur. The reason for this is that the smaller configuration will generate data that should leave the device before the larger configuration has been completely removed from the device. This leads to both configurations trying to output data trough the same IO-ports. If we consider this problem in relation to the reconfiguration wave it will appear that the smaller configuration will try to send data past the reconfiguration wave, into a region of the device that still is configured with the larger configuration.

To resolve this problem another wave type could be added to the device. This would be the idle wave, where all cells currently affected by the wave would not perform any operation. By inserting a number of idle waves right after the reconfiguration wave the reconfiguration system could remove the previous configuration without any data from the new configuration overtaking the reconfiguration wave. Unfortunately, the number of idle waves does however affect the amount of computational overlap that the device will exhibit.

The idle wave can also be used, without any other changes to the grid, to allow scalability. The inactive cells are given a default configuration that connect the outputs to the input registers, and idle waves are inserted between computational waves. The configured cells would be forced to idle long enough for the data to pass through the inactive cells, and when this has happened the next series of computational waves will enter the grid. However, this approach lead to a negative impact on the throughput of the device.

Multiple scaling approach

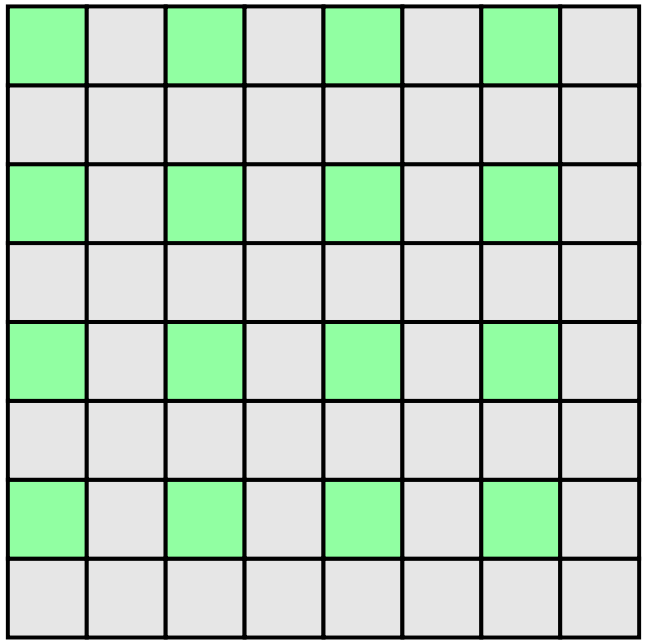

Figure 3.5: Multiple scaling

Restricting the possible configuration sizes to those who can divide the device size, allows for another scalability type. Then the configuration can be scaled up by inserting cells that contain a dummy configuration between each configured cell. This approach is illustrated in figure 3.5. The colored cells contain the original configuration, while the gray cells contain dummy configurations.

The actual dummy configuration can contain a configuration that will activate a bypass MUX on the input registers and connect them to their corresponding outputs. This is similar to the bypass strategy presented previously. The advantage gained by spacing the configured cells apart before the use of bypass MUXes is that the amount of cells the data must pass through before reaching its destination is evened out. This will mitigate some of the problems with this style of approach to scalability.

If the dummy configuration connects the outputs to their corresponding inputs in the cell. These dummy cells will then simply move data in the grid and not perform any operation on it. Then by inserting a suitable number of idle waves between each computational wave, the data will be able to propagate through the grid at a normal rate. However, computations will be performed at a lower rate, and data will be able to leave and enter the grid in the same number of computational wave steps as a correctly sized device. As with the grouped approach, the throughput of the device will be lower.

3.2.4 Configuration compatibility

The ability to use older configurations with a newer device is quite desirable property. Although the old configurations will not be able to use new computational elements introduced to the device. Performing a complex translation procedure of an old configuration to a new configuration internally in each cell would probably require precious hardware resources. In addition the time required to perform such translation could reduce the speed of the reconfiguration wave, which would also slow down computation waves.

By adding some restrictions on the future version of the configuration format and carefully designing the existing, the usage of older configurations in a newer cell can be achieved with little effort. The configuration format can be viewed as bit stream that grows in relation to the configuration version. If such a scheme is used an older configuration version will be shorter than a newer configuration version. Therefore it must be decided how an older configuration is to be expanded into a newer configuration, without causing unwanted operations or harmful side effects.

A simple solution to this is adding a restriction to future configuration versions, that will allow older version to be zero padded to match the length of the newer version. The internal units of a cell being affected by the zero padded region of the configuration should not cause any harmful interference to the units being controlled by the older part of the configuration.

This approach alone is however quite naive, and will cause the configuration to grow unnecessarily fast. As all newer configurations is forced to extend the current configuration format. Consider the problem presented in figure 3.6a. A MUX, shown in blue, is controlled by a part of the configuration, shown in green. The value of this subsection of the configuration decides which of the inputs that should pass through to the output of the MUX. As the number of inputs in use is not a power of two, there is a unused bit combination.

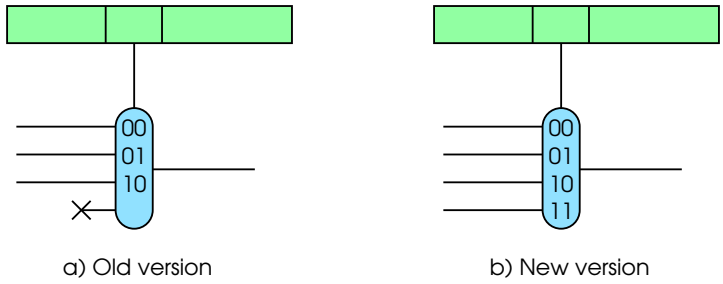

Figure 3.6: Configuration section extension

This architecture is later extended into one were the fourth input is required as shown in figure 3.6b. Requiring the configuration to be extended in size to allow this is unnecessary. As there is an unused bit combination in the region of the existing configuration that controls this MUX. However, this requires a change of the interpretation of the existing configuration. To allow for this the existing configuration must not be allowed to use unused bit combinations as default values, or exploit undocumented behavior caused by using such bit combinations.

3.2.5 IO

So far only the core of the RC unit has been elaborated upon. A complete system also requires an IO system, or more specific a memory system. The presented configuration scalability solutions does put some limitations on the construction of this memory system. When scaling the configuration the cells that should be able to perform IO might not be physically located at the edge of the grid. As such, the memory system should be separated from the edge cells. This simplifies the construction of the reconfigurable grid, as all cells will then be equal.

This solution does have some limitations. With the communication in the grid being unidirectional, there is no direct communication from the cells that receive data and the corresponding memory interface. As all cells are equal the communication from cells that sends data to the memory system, is limited as well. Therefore the memory system should be able to operate autonomously in relation to the grid.

Workloads that can benefit from reconfigurable hardware are often contain very little control flow, and contain a fixed set of operations that will be performed on data[CH02]. This allows for heavy pipelining of the computation. This often leads to a regular memory access pattern. In addition when a RC unit for streaming operations the temporal locality in the data is quite low as well[CEL+03].

Given these observations, a memory system that can deliver high performance and still be flexible in regard to the memory access pattern should be used. Such a memory system could be a vector-based memory system. It allows for autonomous operation because the memory access patterns are described in advance, and is still flexible in the access patterns possible.

A typical memory system might cause some situations that will have devastating effect on the RC grid. In a situation where the memory system is unable to deliver data fast enough to the grid, and the waves present in the grid are allowed to advance. Garbage data will enter the grid and this will most likely cause data corruption. A naive solution to this is freezing the grid. Although data corruption is avoided, it has a unfortunate side-effect. When the grid is frozen the RC unit is not able to switch tasks, as the reconfiguration scheme relies on overlapping computation of the two tasks.

The memory system might not be able to deliver enough data in three major cases. The first is bus congestion. This is generally caused by the RPU and the CPUs fighting over system resources. When this problem occurs the outstanding memory accesses are likely to complete soon, and simply freezing the grid would probably be sufficient. The alternative, which is to reconfigure, will increase the load on the memory system and worsen the situation.

The second case is when a virtual memory system is employed. In such memory systems, page misses can occur, and retrieving such pages from secondary storage is likely to be very time consuming. It is under these conditions a preemptive multitasking system helps improve performance. If the grid is stalled when this problem occurs, the RC unit would be unable to switch task, and the potential improvement to performance would be lost. This strongly suggests that a method for detecting such stalls before they happen must be implemented.

When the grid computes, the first data consumed by a computational wave is done by the cell that only has inputs from memory. When these data enter the grid, and a wave is generated, the system must be able to guarantee that all data consumed by that wave must be available, or else the grid will stall, and the ability to reconfigure will be lost. A simple approach to this is requiring that all data consumed by the waves currently present within the grid must be available on-chip. If the data that will be required during by a wave is not on-chip, it must be refused to start. The on-chip memory storage required will become quite large when the size of the grid is large, as there will be a large amount of waves present at any time.

Another approach might be to keep a page table on-chip. This can be used to predict when page misses will occur. This is possible as all future memory accesses are known by the vector memory system. When a page miss occur it must be examined whether a task switch should be performed, or if the grid should be stalled until the required data has become available.

This examination might be triggered by the RPU by sending an interrupt to a CPU, which will load the corresponding operating system routines. These routines will decide the appropriate action. There are some overhead in this method, compared to a hardware solution. However, it allows for a large amount of flexibility in the implementation of these routines.

The last case where the memory system might stall is when memory mapped IO is used. If the RC unit reads data from a IO unit directly, and the IO operations are blocking, the memory operation will not complete until data has become available. The IO unit probably has status registers that contain enough informa-

tion to discover if a read/write operation will block. However, such status registers are likely to be device specific, and implementing support for all devices in the RC unit's memory system is likely to be impossible. In addition it would not be future proof, as new devices would become available after the RC unit has been produced. The easiest way to deal with this problem is by restricting which part of the grid that will be allowed to communicate with IO devices. Only the cell that will initiate waves should be allowed to communicate with IO devices directly. If a stall occur during such operations a reconfiguration wave can still be generated.

However, implementing such a complex high-performance memory system would require a great deal of time and effort. It was therefore ultimately decided to be outside the scope of this thesis.

3.3 Cell and computational properties

The basic reconfigurable unit within the outlined device is the cell. The major design issue related to this basic unit is the discovery of the computational properties of a cell. The computation properties must be seen in relation to the granularity of the device and the size of each cell. A fine-grained cell would allow for implementation of custom logic functions, while a coarse-grained cell would allow for efficient arithmetic operation.

The importance of the cell size must be considered in relation to the unidirectional interconnect. Each cell can contain internal state that must be stored and restored during reconfiguration. This internal state may be used as an operand to the computation performed by the cell. The cell size affects the amount of of computation each cell will perform. Hence, the cell size affect the amount of computational operations the internal cell state can affect.

3.3.1 Reconfiguration issues and granularity

As shown in *Multitasking on a reconfigurable computing system* [Oft09] the configuration per cell affects the peak required memory bandwidth of the device. Some of this effect could be mitigated by a configuration write-back cache, that would distribute the load generated by storing a configuration to memory. This might not be sufficient and the bandwidth requirement should be taken into consideration during the cell design process. This limitation on the amount of configuration suggest a limit on the relationship between granularity and cell size.

And as fine-grained devices requires more configuration data the resultant cell within such a device should be small. Hence, the logic function implementable in each cell would also be small. A coarse-grained approach will allow for cells with a larger amounts of arithmetic power per configuration bit than a fine-grained device. Which, allows for cells of a rather large size with small configuration sizes. Hence, coarse-grained architecture was chosen.

3.3.2 Instructions

In *Fine- and Coarse-grain Reconfigurable Computing* [VS07] a standard methodology for the design of coarse-grained architectures is presented. This methodology uses the characteristics of the intended application domain or class to design the coarse-grained architecture. This inevitably causes the resulting architecture to be tied to a single application domain or class. Furthermore, this is a rather time consuming process. These drawbacks together with the fact that the resulting architecture should be a proof-of-concept for the multitasking reconfigurable architecture, allowed for a different approach to the discovery of a cell's computational properties.

The configuration of a cell should be of fixed length, and the configuration of a cell should be require very little or no decoding hardware. As such decoding hardware would be only used during device reconfigurations. By acknowledging these facts and considering their general purpose processor counterpart, led to the evaluation of RISC-processors(2.5.4) as a suitable basis for the cell design.

The computational properties of the cell was decided by evaluating RISC-processor instruction sets ([SPA94], [MIP09]). All control flow instructions were discarded as the computational model for the RC device and the general purpose processor differs and such instructions would probably be not be implementable, unless the cell contained an entire RISC processor. Additionally, all floating point instructions were discarded as floating point units are quite large compared to their integer counterparts. The result was a subset of common logic and arithmetic instructions.

Instruction	Operation	Can also be used as:
ADD	Addition	Multiply by 2
SUB	Subtract	Zero-source
MUL	Multiplication	Shift arithmetic left
DIV	Division	Shift arithmetic right, One-source
SLR	Shift Logic Right	
SLL	Shift Logic Left	Multiply by 2^X
OR	Logic OR	Pass through
NOR	Logic Not OR	Logic NOT
AND	Logic AND	Pass through
XOR	Logic Exclusive OR	Zero-source

Table 3.1: Instruction listing

Table 3.1 lists the resultant instruction set. The first column list the short name for the instruction and the second displays the exact operation. Some instructions can also create more specialized results, based upon the inputs given to the unit performing the operation. For example *XOR* will always result in *0* if the inputs given are identical.

3.3.3 Branching

The coupling solutions presented in *System coupling* (3.1) puts limitations on the amount of control a general purpose processor can assert upon the RC device. The independence of the RC device in relation to the CPU and thereby the running tasks, almost completely eliminates control flow from a task running on the CPU to a task running on the RC device. One could achieve such control by letting the RC device generate an interrupt when a control decision was required. This interrupt would wake the task that spawned the task running on the RC device, if this task was not already running on a CPU. The task could then examine state of the RC unit and its own state, and decide on the appropriate action. However, the overhead of such a control scheme would probably obliterate any speedup achieved by using reconfigurable hardware, if branches are taken frequently.

Internal path branching

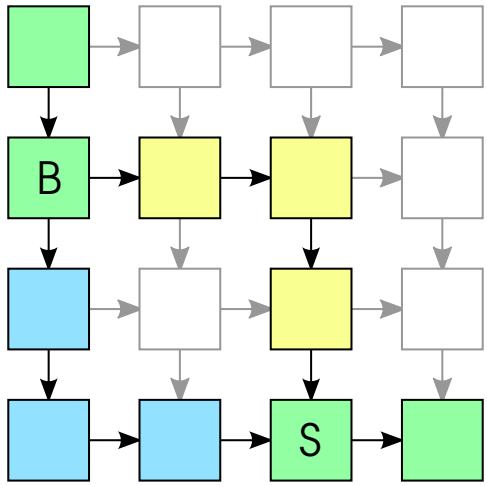

Figure 3.7: Branching

The solution to this problem is implementing elementary branch functionality in the RC device. As RC devices exploit spatial computation, rather than temporal computation, branching could be implemented using the same concept. Such an approach would cause data to follow different paths through the grid. The effect of data passing through different cells will be that different operations being performed on the data. This spatial branching is demonstrated in figure 3.7. White cells are unused in the running configuration. Green cells shows the common path that are used by data elements regardless of the path taken as a result of the branch. In the cell labeled *B* the branch configuration is located. Based upon the computation performed here a branch can be taken to the right, and will follow the yellow path, or could be taken downwards and follow the blue path. Both path merge again in

the cell labeled *S*. In this cell it must be decided which input should be selected and passed on to the remainder of the common path. With the computational propagation presented earlier, all such selection cells will have an equal amount intermediate cells on the path from the selection cell to the branch cell. Given that data propagates from one cell to another at a constant rate, only one input cell to the selection cell will have valid data at any time. Which simplifies the selection process.

Branch-based memory access

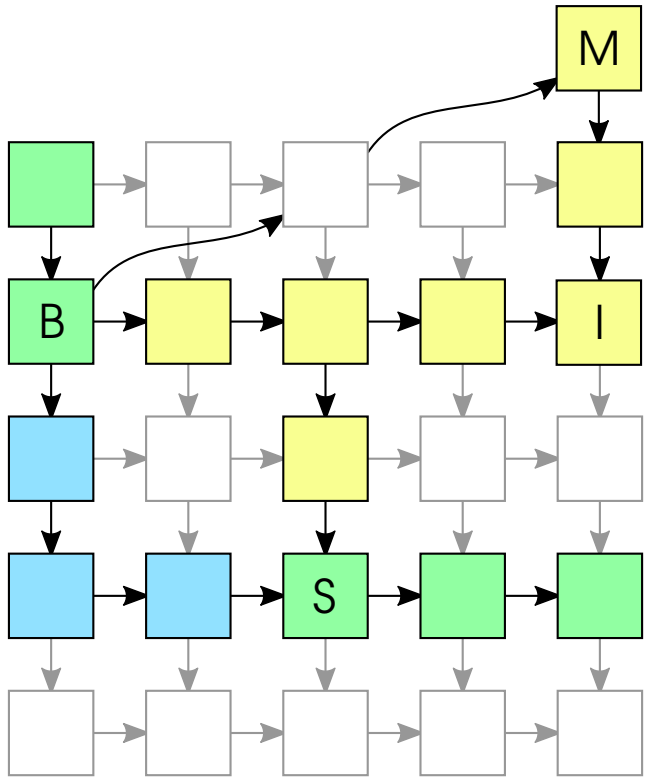

Figure 3.8: Branch-based load

One of the design issues with this branching scheme is whether or not to allow for memory access based upon branching, and whether both loading and storing based upon branching should be allowed. Adding support for branch-based writes to memory is rather simple. The only requirement is that the memory system connected to the grid is aware when there is valid data on the output ports of the grid, and do writes based upon this. This should be sufficient given that inactive branches does not propagate data that could appear to be valid.

Loading from memory based upon branching within the limitations imposed by the propagation of data and reconfiguration is somewhat harder. The main reason for this is the unidirectional communication. The communication between a cell within the grid and the memory system requires that some data will not flow directly with the wave.

To achieve this the existing interconnect can be augmented with additional signals between cells as shown i figure 3.8. This approach assumes constant wave propagation speed. The new signal will be able to stay ahead of any reconfiguration wave, and will reach the memory unit labeled after two propagation steps. The unit will then load data from memory and transfer it into the grid. The data that caused the branch and the newly loaded data should transfer into one of the cells for the branch-based load to be useful. This intersection point between the two data paths is labeled *I* in figure 3.8. To meet this demand the memory unit must be able to load data from memory in one propagation step.

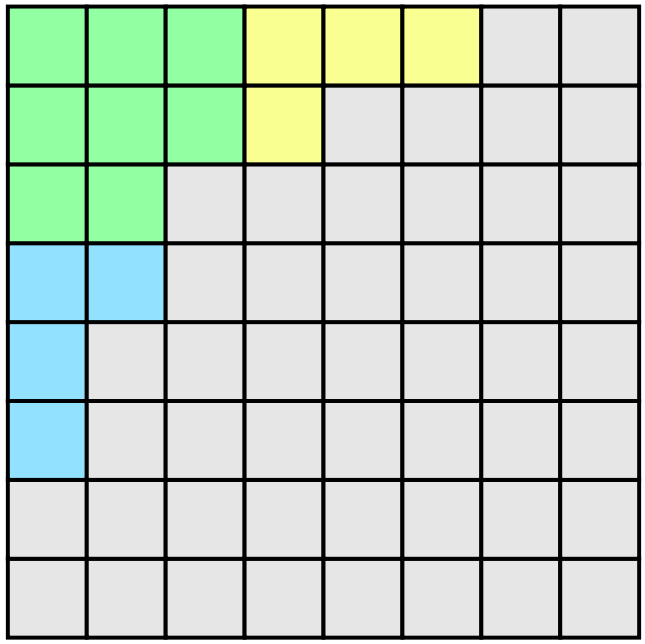

Figure 3.9: Cells load capability

This augmented structure does have its drawbacks, which figure 3.9 illustrates. Colored cells have the ability to contain a branch point that can cause external data to be loaded. There are even limitations on some of these cells, as the augmented interconnect might only reach a memory unit in one direction. These cells are colored yellow and blue in the figure. Were yellow cells can only perform a branch-based load from a horizontal memory unit, and blue cells can only perform branch-based load from a vertical memory unit. Green cells on the other hand can perform branch-based loads in any direction. Further figure 3.9 show that any branch-based

load must be decided before the majority of computations can be performed within the grid.

3.4 Reconfiguration system

The previously presented reconfiguration method requires a reconfiguration system that is able to support this style of runtime reconfiguration. The interconnect needed to reconfigure has previously been considered in *Multitasking on a reconfigurable computing system* [Oft09]. In addition to a suitable interconnect, the reconfiguration system should support configuration scalability, and configuration compatibility for the purpose of backwards compatibility.

3.4.1 Overview

As the reconfiguration system does not contribute directly to the computational power of the RC device, it is desirable that the hardware resources required by the system are low. In addition, it should be created in an expandable manner that will allow for the generation of a grid of any size. The reconfiguration system should support the scalability schemes presented in section 3.2.3.

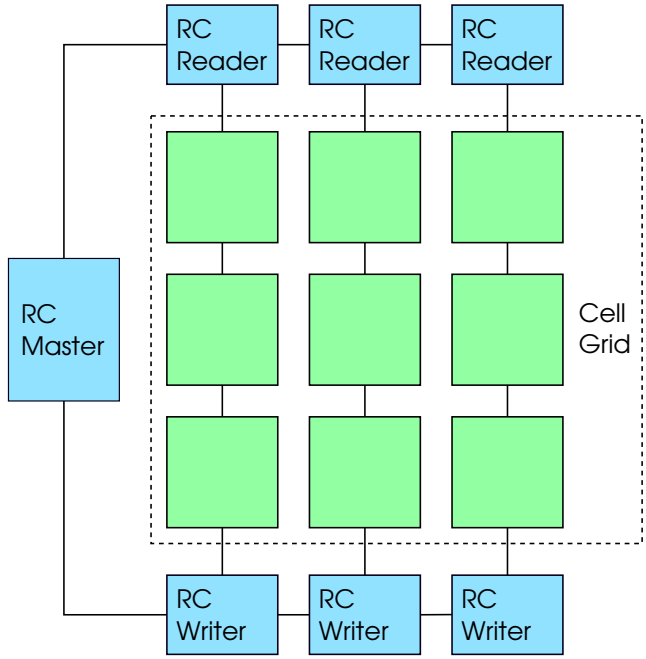

Figure 3.10: Reconfiguration system overview

In figure 3.10 the reconfiguration system is presented. The reconfiguration is separated into three distinctive units. The reconfiguration is governed by a recon-

figuration master, which contains the reconfiguration control interface exposed to the rest of the computing system. Each cell column has a reconfiguration reader and a reconfiguration writer.

The reconfiguration reader reads a configuration from memory. This configuration is then transferred to a cell. During this process, the current configuration is transferred out of the cell into the reconfiguration writer. The reconfiguration writer then proceeds by writing the configuration back to memory.

The propagation effect of the reconfiguration wave is exploited in the interconnect between the reader/writer units. The reconfiguration process will start in the left-most column of the cell grid. It will complete a cell reconfiguration here before reconfiguration will start in the next column. This allows for chaining of the reader/writer units, and propagating control signals from the reconfiguration master when a reconfiguration step has completed. In effect this will delay the start of the next reconfiguration unit in the chain, until the previous one has completed its first reconfiguration. This ensure that control signals propagate at the same speed as the reconfiguration wave.

3.4.2 Grid interconnect

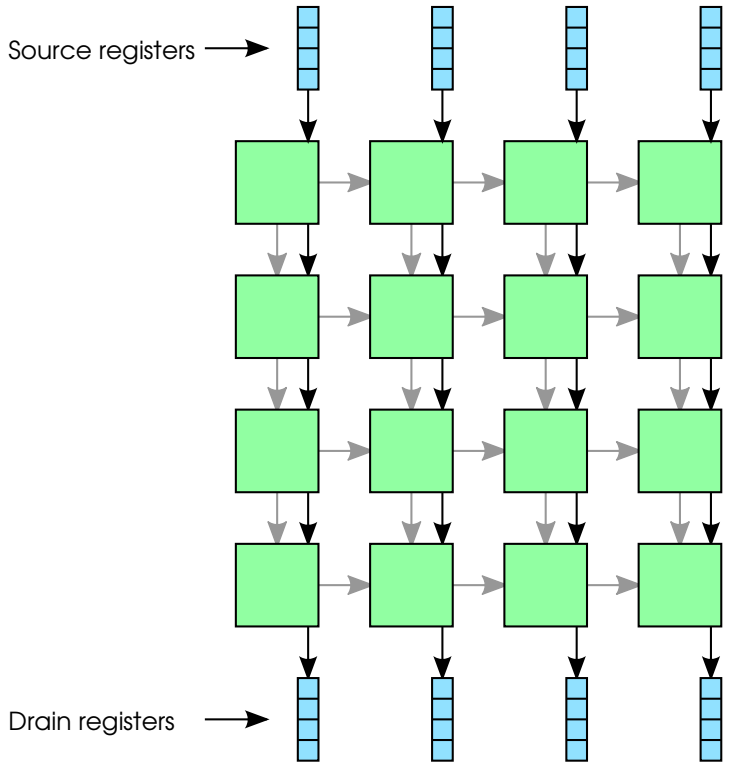

Figure 3.11: Reconfiguration interconnect

The interconnect used for reconfiguration must be able to support the wave based reconfiguration method. If constant propagation speed of waves is assumed, this interconnect can be simplified. The constant propagation speed guarantees that only a single cell is being reconfigured in each column of the grid[1]. Hence, as only one cell will be reconfigured in each column in the grid at any time, only a single reconfiguration bus is required in each column. This interconnect is illustrated in 3.11. Black arrows symbolizes the reconfiguration bus, while the gray arrows is the existing interconnect used for data propagation.

As the reconfiguration wave propagates at the same speed as the computational waves, the reconfiguration of a single cell must be performed in the same amount of time as a computational operation. This leads to a requirement that the reconfiguration bus must be able to retrieve the state of a cell, in addition to loading a new configuration and state into the cell. A wide-parallel bus, as used in Piperench (2.5.2), will probably be unsuitable, as it will be subject to skew and crosstalk. Therefore a narrower bus, that should be able to operate at higher frequencies was chosen.

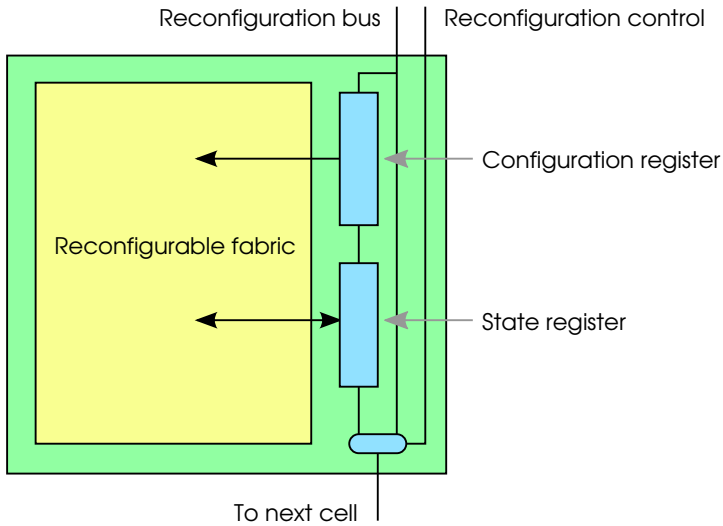

Figure 3.12: Cell reconfiguration interconnect

To explain the function of the reconfiguration bus it is useful to consider the cell model shown in figure 3.12. This is not a correct representation of a cell internals. However, it allows for a rather highlevel approach to the explanation of cell reconfiguration. The cell consists of a some unspecified reconfigurable fabric. The functionality of this reconfigurable fabric is controlled by the configuration register. The internal state of a cell is stored in a separate state register that can be read and written by the reconfigurable fabric.

The state and configuration registers are chained together on a single bus. This

[1] If multiple reconfiguration waves are not allowed

bus is connected to a MUX that can either insert the two registers into the bus, or allow the bus to pass straight through the cell. The operation of this MUX is decided by an external reconfiguration control signal. This control signal should also disable writing to the state register, so that it will not be corrupted by random writes to the register during reconfiguration.

The reconfiguration bus is connected to shift registers in both ends as illustrated by the blue registers in figure 3.11. The new configuration and state is put in the source registers by the reconfiguration readers. During reconfiguration, the content of these registers will be clocked into the configuration and state registers located within the cells that are the targets for reconfiguration. During this process the current configuration and state of the target cells will be clocked into the drain registers. The reconfiguration writer can then retrieve the contents of the drain registers and store it to memory.

3.4.3 Storage format

A suitable format for storing configuration and state in memory is needed to decide some of the reconfiguration design parameters [2]. Such a storage format should contain the information needed to automatically perform compatibility and scalability operations when loading a configuration. In addition it should have a regular memory structure that will allow for efficient memory operations.

The information needed about the configuration by the reconfiguration system is the configuration version and its dimensions. The simplest approach to incorporating this information into the storage format is using a configuration header, where all this information is kept.

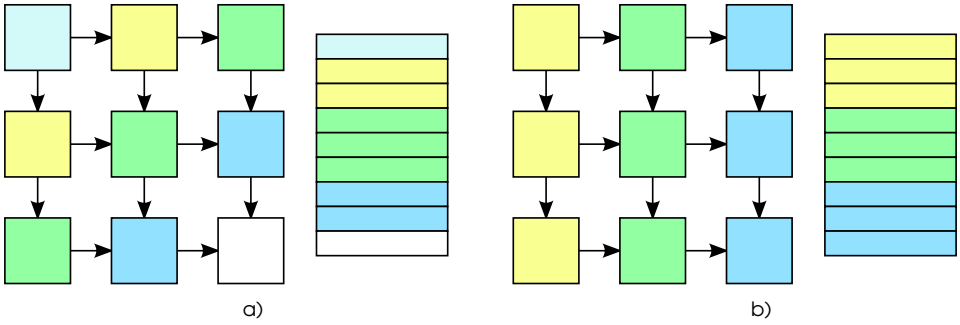

Figure 3.13: Storage formats

The rest of the configuration should be organized in such a way that will allow for simple hardware implementation and efficient memory operation. There are some tradeoffs to be made here. If locality in memory accesses is important to achieve sufficient memory performance, the cell configurations should be stored in a per wave manner. This is illustrated in figure 3.13a. The cells are colored in a per

[2]Such a format was developed for the software simulator in [Oft09] however it was deemed unsuitable.

wave manner. To the right of the cell grid, the corresponding layout of configurations in memory is shown. This approach complicates hardware implementation, as the distance between each configuration that will be fed into the same cell column will increase, as the reconfiguration wave propagates.

However, if locality in memory accesses can be slightly reduced a less cluttered storage format can be used. By grouping the configurations on a per column basis, the reconfiguration hardware can be simplified. This is shown in figure 3.13b. All the configurations that will be sent over the same RC-bus will then be stored sequentially in memory. Then the reconfiguration system connected to a cell column can simply iterate over its subsection of the configurations.

Chapter 4

Implementation: Device version 1

During the design stages of the RC unit it was decided to split the design into two versions. This was done to demonstrate the backwards compatibility capabilities of the device. This chapter describes the first version of the device. The major difference between version 1 and version 2 is that version 1 lacks the ability to branch.

First in this chapter the interconnect implementation of the device is discussed. Following this is a description of the internal structure of a cell. The the reconfiguration system that supports the reconfiguration wave is described in closer detail. Finally a unit that allows the configuration scalability approaches to function correctly on the implemented device is presented.

4.1 Grid

The interconnection network in the RC device needs to support the wave propagation phenomenon. This requires a suitable scheme controlling data transfers between cells, or at least monitoring these data transfers. This information is required in order to determine when a reconfiguration wave can be safely initiated. There are two major approaches related to data transfers between cells. The difference between them being how two cells are synchronized during data transfers.

4.1.1 Asynchronous

The WAP uses a data flow based scheme to transfer data between PEs. PEs use a handshake based protocol when data transfers are performed. This asynchronous protocol allows the PEs to operate independently from each other. This eliminates the need for a global clock network. This property is quite desirable as distributing such a global clock signal can be problematic[Fri01]. In addition such asynchronous

operation can increase performance, if the computation times required are lower with certain configurations.

However, using such an asynchronous data flow scheme does complicate the design. As each cell requires hardware resources to implement such asynchronous data transfers. In addition it can complicate reasoning around the behavior of the waves, especially if the computation times required by each cell are not uniform.

With non-uniform cell computation times, the transfer time required between two distant cells in the grid would be path dependent. It is useful to separate between the paths used by a configuration, which is a subset of the physical interconnect and the actual interconnect structure here. The reconfiguration wave is forced to follow the interconnect as the paths used by two configurations might differ.

Even though the reconfiguration wave follow the interconnect, the cell computation times will affect the wave propagation. The reconfiguration of a cell cannot commence until computation has completed and data has been transferred to its neighboring cells. The reconfiguration wave must also have reached all the cell inputs. The reconfiguration system must be able to determine when this has occurred. Hence, the need for monitoring data transfers, given that an asynchronous data transfer method is used.

Given that the cells must transfer data to the next cells in a path before it can receive new data, the cells will operate in a lock-step fashion. The cell with the longest computation time will dominate the computation time of the entire path, which can lead to trouble. When the grid is configured with a configuration that appears to contain two independent paths. Where one path feeds the other with data, through an external buffer. If one path has a lower throughput than the other, caused by higher computation times, the memory system must detect this, and stall the faster path. This complicates the memory system implementation.

4.1.2 Synchronous

To simplify the construction of the device and reasoning regarding both computation and reconfiguration waves, a synchronous data transfer scheme can be used. This will require a global clock distribution network in the grid, which can be hard to construct[Fri01]. The synchronous approach does reduce complexity in the data transfer scheme. As it can be reduced to simple registers on the cell inputs driven by a global data transfer clock.

Requiring that no computation is allowed to use more than the period between each tick of the data transfer clock simplifies the wave generation as well. The control over the data transfer and state, with this requirement, is sufficient to allow safe reconfiguration wave generation, at any data transfer tick. This also ensures constant progress in all waves within the grid. As such, all cells participating in a single computation or reconfiguration wave will have the same Manhattan distance [1] to cell where the waves originate. The uniform wave progress, eliminates any

[1] The distance between two points measured in fixed length straight line segments which are oriented in relation to north/west axises

complications with regards to buffering in the memory system, as data will be generated and consumed at a fix rate.

The simple register approach does lead to some requirements in the cell implementation. Given that the data transfer clock is the only clock in the device, and that a cell should use no more than the period between data transfer ticks to complete computations. The contents of the cell must be completely combinatorial. Implementing more advanced operation such as multiply or division, using only combinatorial circuitry is likely to consume are large amount of hardware resources.

As such, an internal clock should be used within the cells. This compute clock can complicate the simple register approach. The results of computation must be able to pass from a fast clock domain to a slower. Synchronizers can be employed to prevent metastability in this data. However, this might require several cycles on the data transfer clock to retrieve data from one cell, and make it available to another. The synchronizers will also increase the hardware area consumption.

Requiring that the internal computation clock and the data transfer clock be derived from the same source, and that the data transfer clock is generated by dividing this clock can solve this problem. The clocks should then be in sync, so that there is a rising edge on the data transfer clock and the computation clock simultaneously. Data should then be able to pass from one clock domain to the other, without synchronization.

The simplicity in the data exchange hardware, and the predictable behavior of the waves, lead to the choice of using the synchronous approach in this implementation attempt.

4.2 Cell

The cell of the version 1 architecture implements all the arithmetic operations outlined in section 3.3.2. However, the computational properties of the cell was augmented even further by dividing the computational operations between different internal units. As the hardware needed for multiplication and division is rather complex compared to that needed for simpler arithmetic operations, it was decided to separate these operations into an independent unit. Then by allowing these two unit to operate independently more complex operations often used in Digital Signal Processing (DSP), were possible, such as Multiply ACcumulate (MAC).

Operating these units independently is not enough for this operation, as this operation requires three inputs and produces one output. This output, together with one of the inputs is the accumulated value, and must be stored between operations. Therefore a separate state register was added to the cell. This state register will be saved and restored during the reconfiguration process.

Figure 4.1 presents a overview of the cell architecture. The two main arithmetic units are shown in blue, one being the ALU and the other one being a combined multiplication/division unit. The main data registers in the cell are shown in green. Both inputs to the cell are registered. There is also an internal state register. These registers are driven by the data transfer clock, while the rest of the elements are

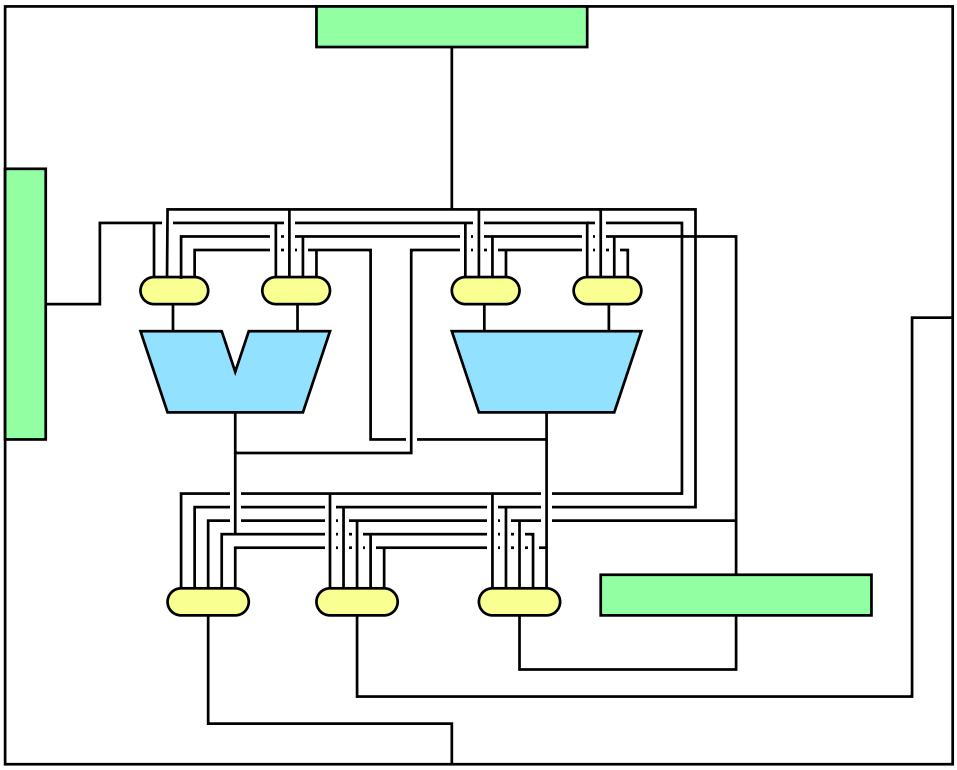

Figure 4.1: Cell version 1

driven by the computation clock. The registers holding the configuration of the cell has been omitted for clarity.

The internal interconnect in the cell is controlled by MUXes, shown in yellow. This interconnection network allows for most connection permutations of the internal units. However, feeding the output of an arithmetic unit back into the unit directly is unsupported as this will generally lead to unpredictable behavior. Feeding one arithmetic unit by the output of the other is supported.

4.2.1 Internal dataflow management

As one arithmetic unit can be the source of data for the other arithmetic unit a scheme forcing the second unit to be idle until data is ready must be implemented. This scheme must allow for any variation in the interconnect between the internal units.

One possibility for such control over the arithmetic units is a separate control entity. This unit would use the configuration of the MUXes controlling the inputs to the arithmetic units to discover the execution order. Checking this execution order is likely to become very complex if the number of arithmetic units increases.

A simpler scheme that require little additional hardware, is the usage of a valid signal in the interconnect. To each data bus, an extra signal is added that signals whether the content of the data bus is ready for computation. This extended bus will pass through the MUXes as any other input signal. Each arithmetic unit will check the valid signals on its inputs before performing any computation. So if an input to an arithmetic unit is connected to the output of the other arithmetic unit, the valid signal would be asserted when the first unit has completed its operation. The second unit will observe this and will then start operating on the input data.

How the valid signal is generated from the ordinary cell inputs is a different matter. The valid signals from the inputs could be permanently asserted, as the inputs in use should always be valid. However, passing the valid signal from cell to cell, allows for a generic interface between cells, and between the cells and the IO system. The IO units connected to the output could easily be triggered by the valid signal exiting the cells. In addition this signal can be used to perform some rudimentary debugging as it will only be asserted where proper computational paths leave the grid.

4.2.2 ALU

The ALU performs most of the computational operations in each cell. Only the multiplication and division operations has been separated into an independent unit. This leaves eight arithmetic and logical operation to be implemented in the ALU. The number of supported operations allow for efficient encoding into a three bit word. For exact operation mapping, see section B.2.2.

The implementation of the ALU was done at a rather high level in VHSIC Hardware Description Language (VHDL) to allow the synthesis tools to choose the optimal implementation of various operations based upon the target architecture.

The exception from this approach is the unit performing logic shifting. Logic shifters implemented in general purpose CPUs are often only sensitive to the lowest $log_2(datawidth)$ bits of the data word, which specify the shift length. If the programmer of such an architecture wants more than these bits to be significant, which would cause the result of the shift operation to be zero, said programmer can implement this with a simple branch. As version 1 of the RC device does not support any control flow, the shifter was augmented to allow for such long shifts, and zeroing the output automatically when a bit is asserted above the lowest $log_2(datawidth)$ bits.

4.2.3 Multiplication and division

Multiplication and division are in general more complex operations than the operations implemented by the ALU. A high-speed multiplication and division unit is likely to be very area consuming. To allow the maximum amount of cells possible from the hardware resources available an area efficient multiplier and divider should be implemented.

A basic design for a multiplier and divider was found in *Computer organization and design* [PH07]. The multiplier and divider designs presented are very similar, and consists primarily of a small ALU and a shift register. This allows for hardware reuse which reduces the amount of area needed to implement these operations. It is noteworthy that these designs, although area efficient, requires a large amount of clock cycles to perform a single operation. The algorithms used and the hardware implementation of this combined multiplication and division unit is described in appendix chapter A.

The resultant hardware structure supports both signed multiplication and division with extensive hardware reuse between these two operations. When dividing division by zero can occur. On CPUs this often causes an exception. The outlined RPU does not have support for such exceptions. This problem was therefore approached in a different manner, which is similar to mathematical limits. When a number is divided by zero the result is the largest possible number, with the appropriate sign, is returned. There is however a corner case that a RPU programmer should be aware of. This corner case is dividing zero by zero. This is in the implementation treated as a positive number divided by zero.

4.3 Reconfiguration system

A major component in the RC device is the reconfiguration system. This section will provide a detailed explanation of the inner workings of the reconfiguration system presented in 3.4.

The last storage format presented in section 3.4.3 was chosen for the reconfiguration system. This storage format allows for a simpler hardware solution, as all the configurations that is needed to configure a cell column is grouped together in memory.

4.3.1 Reconfiguration master

The reconfiguration master is the main control unit of the reconfiguration process within the RC device. This unit is responsible for parsing the configuration header, and take appropriate actions regarding the configuration size. As this is the first version of the device the master does not have extensive support for loading other configuration versions. It will only report an error when a configuration with a version higher than the current device.

The reconfiguration master contains the interface that is exposed to the rest of the computing system. This interface contains status and control registers. The status register contains information such as current state of the reconfiguration master, in addition to some information about the grid. The information that can be retrieved from this register is whether or not the reconfiguration master is busy, if an error has occurred or if the grid contains a configuration. The exact layout of this register can be found in section B.1.1.

The control register is split into several distinct parts. The main control register controls the operation of the reconfiguration master. By using this register, the computing system can instruct the reconfiguration master to load a new configuration, or store the current running one to memory, or both concurrently. This allows for a great deal of flexibility in the possible reconfiguration types that can be performed. However, it does allow for some hazardous operations, such as overwriting the current running configuration without storing it to memory. In addition to this, the control register allows the computing system to clear any error that might have occurred. The exact layout of this register can be found in section B.1.2.

The auxiliary control registers present in the reconfiguration master control the memory operations performed by the unit. Specifically, the external computing system uses these registers to control where a new configuration is retrieved from, and where the current one is stored. Although it could be required that the current running configuration be stored in the same location as it was retrieved, and in effect eliminating the need to specify the store address. This would be very inflexible. However, the ability to specify an arbitrary store address does come at a cost as the reconfiguration master must write the header of the current running configuration back to memory. Which could be omitted with the fixed memory approach as the header would already be present in this memory area.

Figure 4.2 shows the composition of a single cell column connected to a reconfiguration reader and a reconfiguration writer. The reconfiguration reader controls the internal reconfiguration hardware present in each cell, and the reconfiguration writer receives the configuration being clocked out of each cell.

One could imagine that the reconfiguration reader could control the reconfiguration writer as their function is tied closely together. However, this could complicate the hardware implementation in some ways, as the configuration leaving the grid and the configuration entering the grid could have different properties, such as size. Also, in some cases, the reconfiguration writer and reconfiguration reader should not both be activated. When the RC device is idle and a new configuration is to be loaded, there is no configuration that should be stored to memory, hence the reconfiguration writer should be idle. The opposite case is when a task has finished

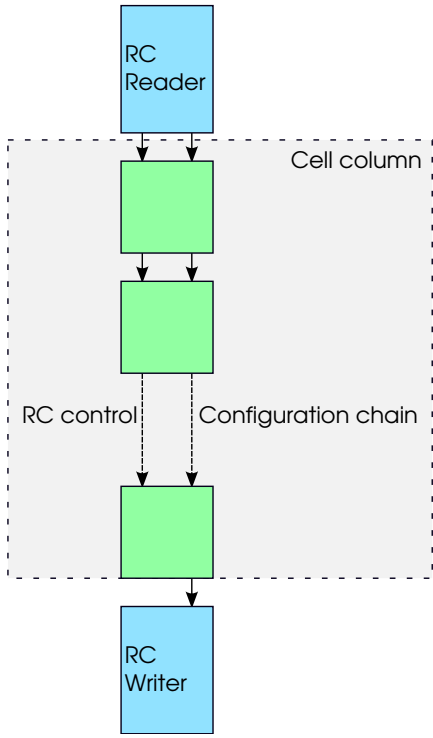

Figure 4.2: Reconfiguration column

using the RC device and there are no other tasks. As the state of each cell might be of importance to the rest of the program, the current configuration should be stored to memory. When this occurs only the reconfiguration writers should be active.

Therefore it was chosen that the actions of the reconfiguration reader and reconfiguration writer should be coordinated by the reconfiguration master. There is one important property in the structure of the cell column, presented in figure 4.2, that should be noted. The reconfiguration reader needs to access memory to read a configuration, before it can start reconfiguring the first cell in the column. During this initial memory access, the reconfiguration writer is idle, as it must first retrieve a configuration before it can be written to memory. This period can be exploited by the hardware design, to allow control signals from the reconfiguration master to travel further than those going to the reconfiguration reader. This can be accomplished by inserting registers on these signals. The propagation time, in cycles, through the registers must match the the amount of time the reconfiguration reader needs to perform the initial memory access.

Configuration loading

In figure 4.3, the internal structure used for loading a configuration is presented. Write signals to some registers have been omitted from the figure for clarity.

The interface registers are shown in light blue. Through these registers the reconfiguration master receives commands from the computing system, in addition to sending the status of the reconfiguration system.

At the start of the configuration process the reconfiguration master receives the base memory address of the new configuration from the source address register. This address is used to access the memory system and read the header of the new configuration. This header is stored in an internal register, labeled *New header* in the reconfiguration master.

The configuration header contains information about the new configuration, such as size and version. These must be verified by the reconfiguration master, so that no unsupported configurations are loaded into the device. The first parameter checked by the reconfiguration master is the configuration version, if it is greater than the current version of the device. The configuration will be rejected and the reconfiguration master halted until the error is cleared by the computing system.

Next, the dimensions of the new configuration must be verified. The scalability schemes presented in 3.2.3 have different requirements to the configurations dimensions. The grouped approach only requires that the configuration is smaller than the actual grid. The multiple approach require that the grid is divisible by the configuration size.

The verification scheme should support the two other scalability approaches. The verification criteria for the two approaches differ. The easiest way to verify the header if the grouped approach is used is a simple comparator. The naive approach to verification if the multiple approach is used, is the use of a divider and checking the results. If the configuration is loadable, the remainder of the division

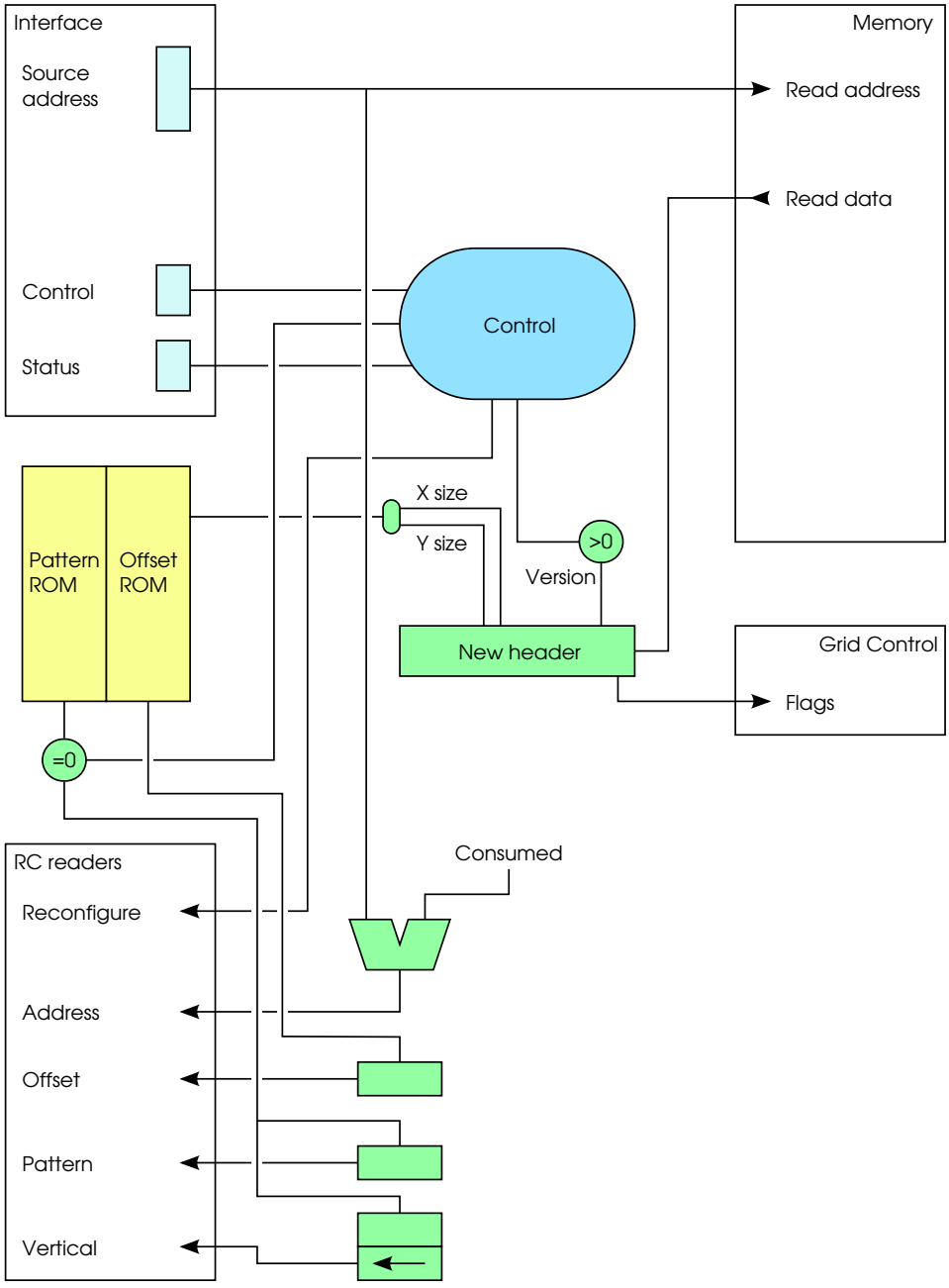

Figure 4.3: Reconfiguration master (load)

operation should be zero and the result non-zero. However, this would probably end up being a rather slow procedure, in addition to consuming area.

An alternative would be the use of lookup tables. This should be more effective than the divider approach on small to moderate grid sizes. This approach uses more area than the simple comparator approach for the grouped scaling approach. However, the ability to use the same structure regardless of which scalability approach is employed mitigates this.

These lookup tables are shown in yellow in figure 4.3, and are labeled *Pattern ROM* and *Offset ROM*. The sizes retrieved from the *New Header* register is used as addresses in these ROMs. In order to save some area, not all the bits in the sizes are used as addresses, only those who would be present in a valid size for the grid. The value of the remaining bits is checked. If they are non-zero, the size is definitely larger than the grid size.

The actual verification of the size is done by using the *Pattern ROM*. This table contains the pattern of cells that should receive dummy configurations and which cells should receive a part of the real configuration. Each row in the table contains as many bits as there is cells in a row. If a bit is asserted, the cell should receive part of the real configuration. Hence, if a row in the *Pattern ROM* contain only zeros, the entire row should receive dummy configurations. If the size used as the address to this ROM results in an all zero output, the configuration is rejected as unsuitable for the grid.

The other lookup table is the *Offset ROM*. This table contains the amount of entire configuration each reconfiguration reader will consume. The amount of configuration that will be consumed differs with the number of cells that will receive configuration in each column. Hence the *Offset ROM* is only relevant in relation to the vertical size of the configuration.

How much each reconfiguration reader will consume is dependent on the addressing modes supported by the memory system. If the memory system supports addressing configurations this table will contain the number of bits set in the corresponding row in the *Pattern ROM*. However, this might not be the case as most memory systems use byte addressing. There is no verification on the output of this table. The reason for this is that each reconfiguration reader might access a private memory space where the configuration they should retrieve all start at the same address. In this case all the entries in the *Offset ROM* would be zero.

The output of the ROM lookups are stored in registers. The outputs of these registers are available to the reconfiguration readers. In addition, the address where the reconfiguration readers should start retrieving their configuration is made available. This address is computed by taking the base address of the configuration and adding the amount that has been consumed by the reconfiguration master, which is the size of the configuration header.

The *Pattern* output to the reconfiguration readers, specify which columns should contain some of the actual configuration, and which should be configured with dummy configurations only. The vertical output on the other hand is used to control which of the cells in column that will receive dummy configurations, and which will receive a part of the actual configuration. The vertical pattern is not made

directly available to the reconfiguration readers. It is instead clocked out a bit at a time through a shift register, as the reconfiguration wave propagates. How this reconfiguration readers use this information will be explained in section 4.3.2. The last signal going to the reconfiguration readers is the *Reconfigure* signal. This signal will activate the reconfiguration readers, and is asserted by the *Control* entity of the reconfiguration master when the other outputs are ready.

The header also contains a flag section. These flags will be output trough flag port shown in the *Grid Control* section of figure 4.3. This output will only be valid when the reconfigure flag to the reconfiguration readers is active, and any flags of significance should be stored by units in the RPU that benefit from them during this period.

Configuration storing

In figure 4.4, the internal structure used for loading a configuration is presented. Components shown in gray exists in the structure used for loading configurations. Write signals to some registers have been omitted from the figure for clarity.

When a configuration has been loaded into the grid, the registers used during this process is loaded into secondary registers. These registers are shown in green in figure 4.4. The information in these registers will be used when retrieving the configuration from the grid. Although this information can be recreated by only keeping the configuration header, and then repeat the table lookup process. However, the registers would be needed anyway unless the tables are duplicated. Which would use more area than these simple registers. In addition the simple copying of the reconfiguration reader registers decreases the amount of time required from the commands are received by the control entity to the reconfiguration can begin.

As the lookup procedure is optimized away, the storing procedure is rather simple compared to the loading procedure. The only operations that must be performed is the storing of the configuration header to the address indicated by the *Destination address* register, calculating the address where the reconfiguration writer should start their work, and outputting the patterns and offsets used when configuring the grid.

It is noteworthy that the loading of a new configuration and storing of the current one should happen simultaneously in a proper multitasking reconfiguration wave. However, the reconfiguration readers and reconfiguration writers should not be activated at the same time, as mentioned previously. This is caused by the period the reconfiguration readers needs to access memory before configuration a cell. From the reconfiguration masters perspective the readers and writers are activated simultaneously, and the signals to the writers are delayed by registers, which are omitted from figure 4.4 for clarity.

4.3.2 Reconfiguration reader

The role of the reconfiguration reader is to retrieve a part of a configuration from memory and configure a cell with it. The reconfiguration reader must be able to overlap these two tasks, as it is required that the configuration of the next cell in a

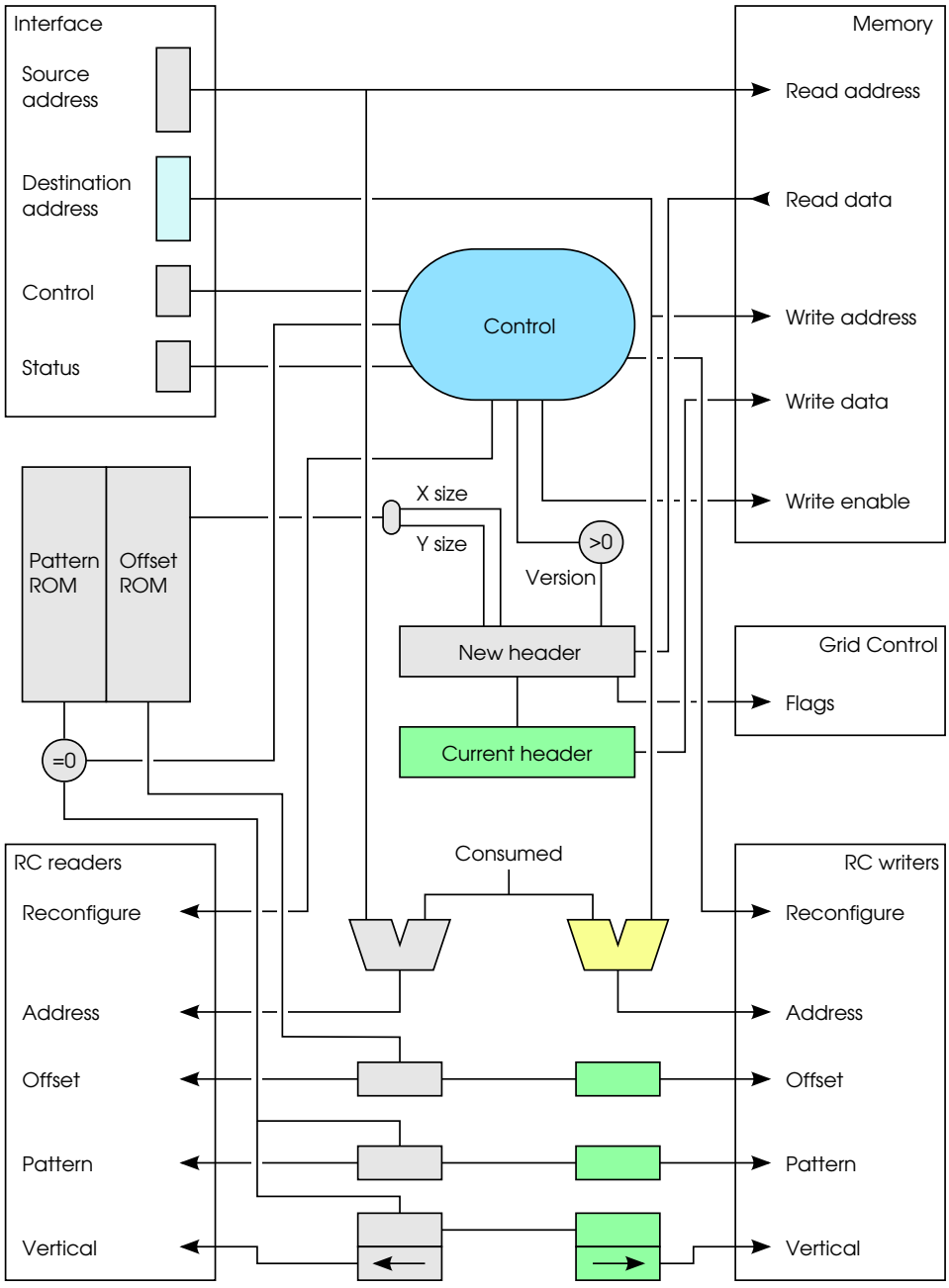

Figure 4.4: Reconfiguration master (store)

column must start immediately after the previous. It must also be able to configure a cell with a suitable dummy configuration if configuration scalability is employed. Further, the interconnect between reconfiguration readers should be expandable, without major changes to its composition.

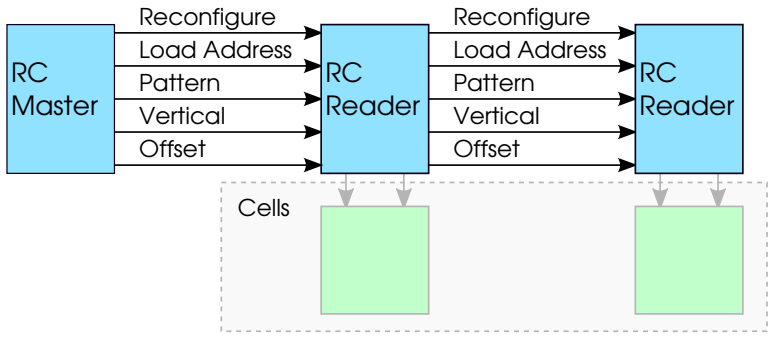

Figure 4.5: Reconfiguration readers connectivity

In figure 4.5 the interconnect between the reconfiguration readers is illustrated. For easy extendability the reconfiguration readers are chained together, such that the output of one reconfiguration reader is the input of the next reconfiguration reader.

The inputs of the reconfiguration readers are also registered, and is driven by the data transfer clock. As such, the input signal to one reconfiguration reader will not propagate to the next until the next tick of the data transfer clock. The clock driving these registers match the propagation speed of the reconfiguration wave. In effect this leads to the reconfiguration readers automatically generating the reconfiguration wave properly. As they will start configuring the first cell in a cell column one cycle apart.

The chaining of the reconfiguration readers also allow for one reconfiguration reader to modify data going to the next reconfiguration reader.

Figure 4.6 illustrates the internal structure of the reconfiguration reader. The reconfiguration process is governed by a control unit. The internal structure is driven by a separate configuration clock. The approach to transferring data between different clock domains presented in section 4.1 is used. This requires that the configuration clock and the data transfer clock are in sync. This configuration clock is also used on the high-speed bus. One might argue that the hardware complexity of the reconfiguration reader would then dominate the frequency possible on this bus. However, the wires in the reconfiguration reader are shorter which reduces propagation delay in the hardware, and the reconfiguration reader has multiple cycles that the operations can be spread across.

This control unit will start operating when the *Reconfigure* signal is asserted. The type of operations is decided by the least significant bit in the *Pattern* bus, and the *Vertical* input. The *Pattern* bus is rotated before it leaves the unit. This causes the reconfiguration readers to be sensitive to different parts of this bus.

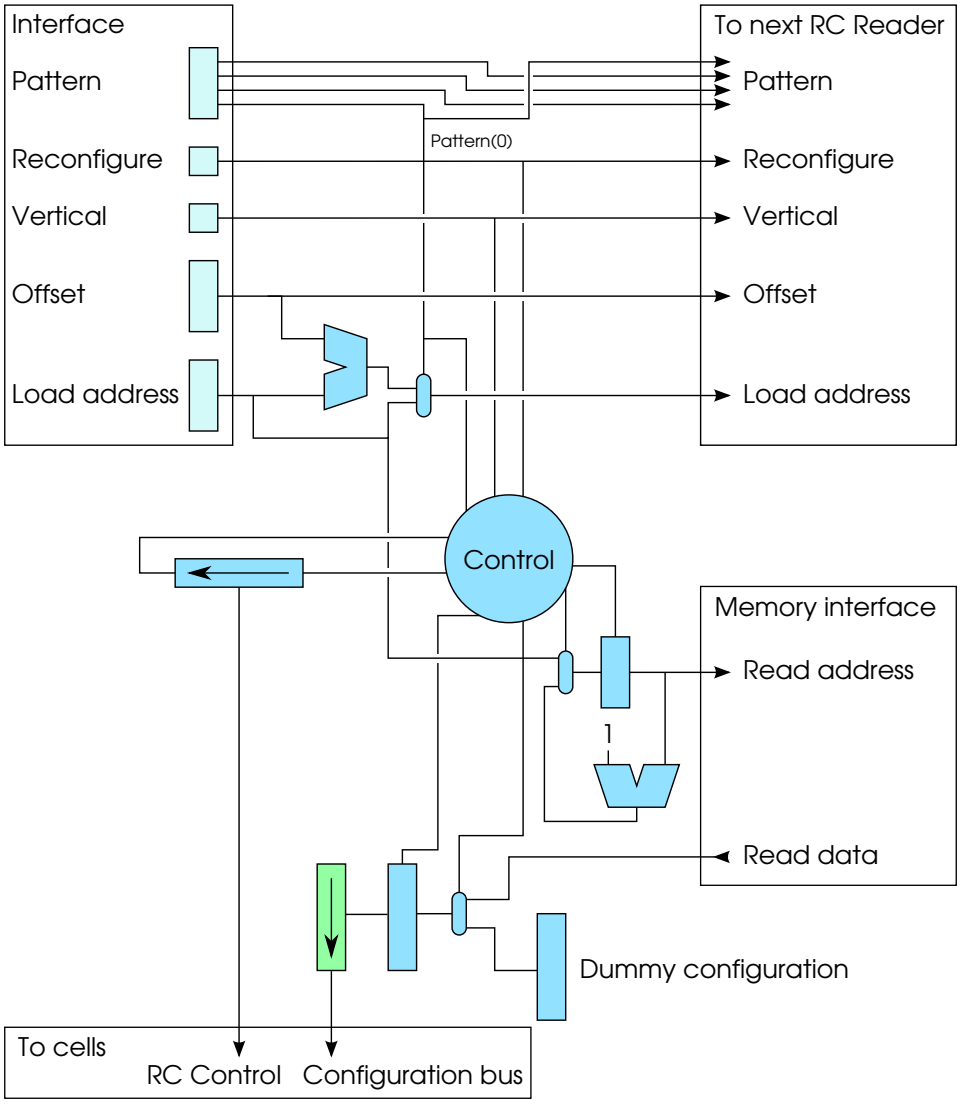

Figure 4.6: Reconfiguration reader

When the pattern bit is asserted, the reconfiguration reader is connected to a column that should be configured with real configuration. If deasserted the entire column should be configured with dummy configurations. Therefore the pattern bit also reflects whether the reconfiguration reader will consume any of the configuration data.

The *Offset* input contains the amount of configuration a reconfiguration reader will consume, and the *Load address* input contains the base address where memory operations should begin. When the pattern bit is asserted, the offset input will be added to the load address before it propagates to the next reconfiguration reader. In effect this splits the configuration into parts divided among the non-dummy reconfiguration readers.

The *Vertical* input is used to perform scaling in a column. That is, decide which cells in a column should be configured with dummy configuration and which should not. This *Vertical* signal is generated, as presented in section 4.3.1, by a shift register in the reconfiguration master. This shift register contains the configuration pattern of a column. When this input is asserted in a reconfiguration reader that also has the least significant bit in the *Pattern* input asserted, the reconfiguration reader will configure the current target cell with configuration fetched from memory. If the *Vertical* input is deasserted, a dummy configuration will be used instead.

As the *Vertical* signal propagates through the reconfiguration readers, it will ensure that an entire row in the grid consists of dummy configuration when it is deasserted. If it is asserted the data on the *Pattern* input will decide which cells will receive dummy configurations.

When the *Reconfigure* signal is asserted, the reconfiguration reader starts the column configuration procedure. It starts by loading the *Load address* input into an internal register. This register is then used to access memory through the *Read address* output.

The result of this memory read operation is stored into a second register, which is connected to a reconfiguration register, shown in green in figure 4.6. The reconfiguration register internals will be explained in section 4.3.4. This register will read from the secondary register and start clocking this data through the *Configuration bus* output at the next rising edge of the data transfer clock, which signals that the reconfiguration of a cell can begin.

The read address register will then be incremented by an adder, which uses the value of the *Read address* output. The result will be stored, and will be used during the memory fetch required to configure the next cell. In figure 4.6 the other input to this adder is one. This is mainly a illustrative number, as the configuration might span more than a single address. However, for simplicity this assumption has been used throughout the design. As the opposite would require multiple memory operations, and a unit that could fuse the data from memory into a continuous configuration.

If the reconfiguration reader is to reconfigure a cell with a dummy configuration, neither the memory read operation, nor the address increment will be performed. In stead the configuration will be loaded from the *Dummy Configuration* register.

The reconfiguration of the cells in the column is controlled by the *RC Control*

output. The width of this signal is equal to the number of cells in a column, and each cell has a dedicated connection. The *RC Control* signal is connected to a shift register. When the reconfiguration process is ready to start, the control unit clocks a logic one into this register. The first cell of the column will then enter the reconfiguration state at the next rising edge of the data transfer clock.

When configuring the rest of the cells in the column, the shift register will advance by a single step for each cell that finishes configuring. This will in effect put the next cell in reconfiguration mode. When the logic one has propagated through the entire shift register the output of the shift register will alert the control unit to the fact that all cells in the column has now been reconfigured. The reconfiguration reader can now return to idle, or immediately start a new reconfiguration dependent on the *Reconfigure* input.

4.3.3 Reconfiguration writer

The reconfiguration writers task is to retrieve a configuration from a cell and store it in memory. The reconfiguration writer must be able to overlap these two tasks, as it must begin retrieving configuration from the next cell in the column, immediately after it finishes retrieving the configuration from the current cell. It must also be able to discard the retrieved configuration if the cell was configured with a dummy configuration. As with the reconfiguration readers, the interconnect should be expandable, without major changes to the composition.

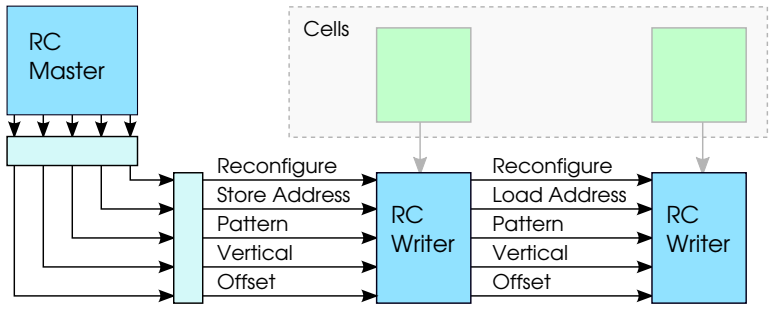

Figure 4.7: Reconfiguration writers connectivity

In figure 4.7 the reconfiguration writer interconnect is illustrated. The interconnect used by the reconfiguration writers is very similar to the interconnect used by the reconfiguration readers. Both chain together the reconfiguration units, with a bus running trough registers. This bus delays the signals going to the next unit in the chain. This structure also allow for a reconfiguration unit to modify the input to the next unit in the chain.

The main difference is two additional delay registers, which were briefly mentioned in section 4.3.1. As the reconfiguration readers need to access memory before they can start configuring a cell, the start signals to the reconfiguration writers can be delayed.

However, this would only warrant a single delay register. A second delay register can be inserted because of the internal structure of a reconfiguration writer. Given that the unit responsible for receiving data from the configuration bus is always active, the configuration being clocked out of the cells can be received without any regards to the control signals from the reconfiguration master. The signals would only control if the information received would be stored to memory. One should be aware that this approach does waste power. This could be mitigated by transmitting an additional signal from the reconfiguration master earlier in the process that would start up the receiver. However, this was not a priority in this implementation.

In figure 4.8 the internal structure of a reconfiguration writer is illustrated. This is quite similar to the internal structure of a reconfiguration reader. The reconfiguration writer uses almost the same bus interface as the reconfiguration reader. The difference being cosmetic, as the address signal is now named *Store address*.

The *Pattern*, *Vertical*, *Offset* and *Reconfigure* signals have the same function. If the least significant bit of the *Pattern* signal is asserted the reconfiguration writer is connected to a column that contains real configurations, and not just dummy configurations.

A reconfiguration writer that is not connected to a column that only has cells with dummy configurations, will add *Offset* to *Store address* before the address propagates to the next reconfiguration writer. This will reserve a memory area that the reconfiguration writer will store the actual cell configurations to.

The dummy configurations does not need to be stored to memory as they contain no state, and they can be recreated by the reconfiguration readers. This keeps size parameters of the configuration constant. This can be quite useful in a system with more than one RC device. If the two RC devices are of different size the following scenario can occur.

A configuration, that matches the size of the smallest of the two RC devices, is loaded into the larger RC device. The largest RC device employs its scalability scheme to the configuration and in effect increases the configuration size. If the configuration was then stored to memory, including the dummy configurations that would be inserted, the configuration would be unloadable by the smaller RC device. Keeping the configuration size static would allow the smaller RC device to load the configuration, even after the configuration has loaded by the larger RC device. In addition to this, discarding the dummy configurations inserted during scaling reduces the load on the memory system by the reconfiguration writers.

The *Vertical* signal, has the same functionality as in the reconfiguration reader, which is to decide whether the configuration currently being retrieved from a cell was inserted by the scalability scheme or if it contains actual configuration.

The configuration being retrieved is first clocked into a reconfiguration register, shown in green in figure 4.8. The internals of this register will be explained in section 4.3.4. When a configuration has been fully clocked into this register the result is stored in a second register. So that the reconfiguration register is ready to receive a new configuration while, the reconfiguration writer stores current configuration

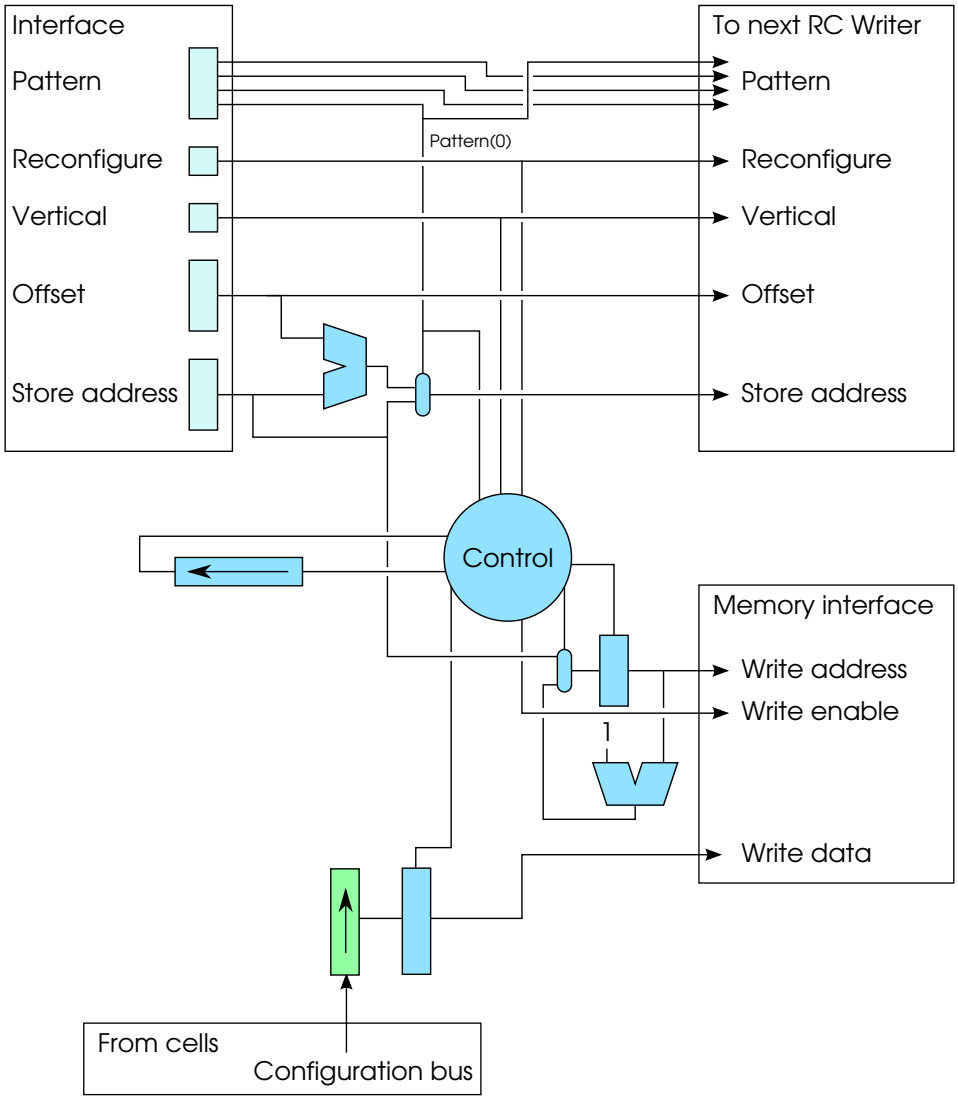

Figure 4.8: Reconfiguration writer

to memory.

The *Reconfigure* signal instructs the control unit to initiate reconfiguration procedures. The address the configuration should be written to is loaded from *Store address* into an internal register connected to the *Write address* output. As the startup is delayed by the external registers, the first configuration should already be present in the reconfiguration register. The configuration is loaded into a secondary register which is connected to the *Write data* output. The *Write enable* will then be asserted by the control unit if the configuration is decided not to be a dummy configuration by the *Pattern* and *Vertical* inputs.

After the write operation completes, the address is incremented. As in the reconfiguration reader, the increment amount is somewhat illustrative. As this assumes that each address in memory is able to store a complete configuration. Whether this is correct or not is dependent on the memory system implementation. For simplicity in the design this was assumed.

The reconfiguration writer uses the same termination detection as the reconfiguration reader. This is a shift register that has the same length as the number of cells in a cell column. When a logic one has been shifted through the entire register a feedback connected to the output will alert the control unit that the current reconfiguration has completed. The reason for using this shift register, and not terminating based upon the *Reconfigure* input, is to enable the reconfiguration system to start a new reconfiguration right after the previous has completed.

4.3.4 Reconfiguration register

This section presents the reconfiguration register structure. This register is used by both the reconfiguration readers, reconfiguration writers, in addition to be present in each cell. In each cell the reconfiguration register is used as both the state register, and the registers holding the configuration.

The reconfiguration register must be able to store data on the rising edge of the data transfer clock, as this clock trigger state storing within the cells. The reconfiguration bus connected to the reconfiguration register, as mentioned in section 3.4.2, is a narrow high-speed bus. Therefore the register has two potential clock sources, which is problematic as the synthesis tools for FPGAs s has no support for generating such registers.[2]

Since the high-speed clock is needed to clock data out of the register this clock is chosen to be the main clock of the register, and writes to the register should only be performed when there is a rising edge on both the configuration clock and the data transfer clock. The challenge then is generating a write pulse based upon the data transfer clock. This is quite hard. The problem being that the rising edge of the data transfer clock should occur simultaneously with the rising edge of the configuration clock. Therefore any signal generated by this event would be to late to be of any significance.

Skewing the clocks in relation to each other might be a solution. If the data transfer clock rises a little earlier than the configuration clock, it is possible to use

[2]Xilinx ISE Foundation 11

this event as a source for some circuit that will generate a suitable write pulse. However, skewing the clock for the entire cell, could cause the cell input registers to loose stability before this write operation occurs. Are the input registers then connect to the state register directly, the state data could be corrupted.

Another solution might be to use an internal counter in the reconfiguration register, that increments on the rising edge on the configuration clock. When this counter has reached the number of edges there is between each data transfer clock edge, the counter will be reset and the input value written to the register. This keeps the entire register within a single clock domain, at some additional hardware cost.

Combining the two clocks by using a logic and gate, and using this as the clock input to the register will not solve the problem directly. Because there is still two clocks used by the register, the configuration clock is still needed to clock data, and the modified clock used to control the write operation. The new clock generated will also have a rising edge, every time the config clock falls and rises while the data transfer clock his high, which is far from correct behavior.

However, this approach can be used if the duty cycle of the data transfer clock is modified. This requires that the duty cycle be severely reduced, to the point where the high period of a data transfer clock cycle, is equal to the high period of a single configuration clock cycle. Then there will only be a single rising edge, each time the configuration clock and data transfer clock rises simultaneously. This cannot be used as the clock input for the register as there will still be two clocks in the register design.

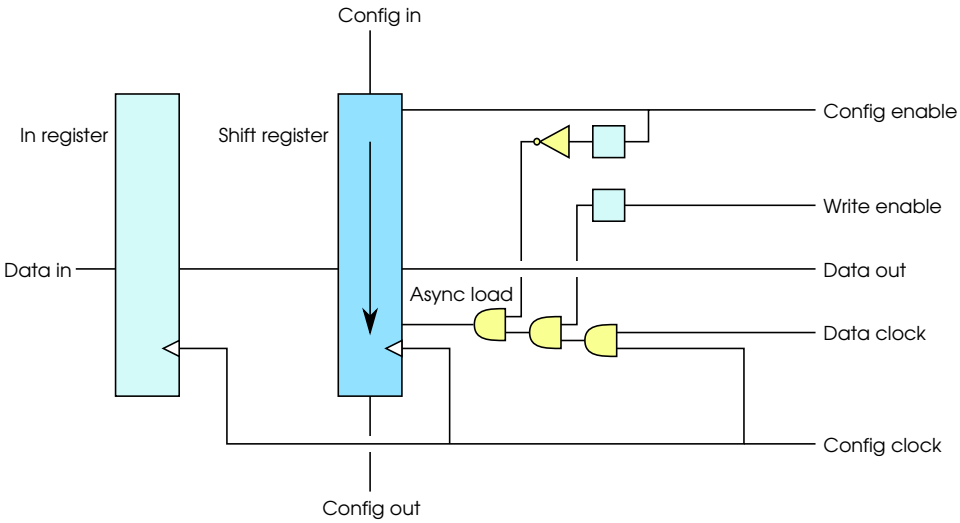

Figure 4.9: Reconfiguration register

A solution that uses this approach is shown in figure 4.9. To the left in this figure there is an input register. As the name suggests, the input to the reconfiguration register is first clocked into this register. The result of the combining the clocks

does not drive any clock inputs. It is used to trigger an asynchronous load from the input register into the main shift register. The function of the input register is to keep the input stable during this asynchronous load.

However, the combined clock signal does not drive the asynchronous load input directly. Shown in figure 4.9 it is also ANDed together with two other signals. The *write enable* controls whether a write operation is allowed, and will keep the input low if this signal is deasserted. This signal is registered as the source might be a part of a different reconfiguration register containing the configuration of a cell.

The *config enable* signal controls whether shifting should be performed or not. The inverted *config enable* signal is also combined with the *write enable* signal and combined clocks. The reason for this might not be that obvious. Consider the reconfiguration of a cell. During this period the contents of a the reconfiguration register must be clocked out and replaced with data from the new configuration. If this reconfiguration register is the state register in a cell, it is likely that *write enable* signal is continuously asserted. When the new data has been clocked in, it would immediately be overwritten by the asynchronous load. However, the inverted *config enable* signal is delayed through a register that will prevent this write operation.

There are some limitations on the reconfiguration by using this structure. The shifting cannot be started before the asynchronous load has ended. Which is after the first configuration clock cycle after a rising edge on the data transfer clock.

When the reconfiguration register is used in reconfiguration readers, it must be possible to write new data into it at every rising edge of the data transfer clock. The cause of this is that it will be used to clock data into cells at every data transfer clock cycle, while reconfiguring. This was solved in the VHDL with generics, that overrides the config enable inverter structure.

4.4 Runtime manager

The task of the runtime manager is generating idle waves based upon the current running configuration. The implementation of this unit does however assume that the grid of the unit is square. Scaling of rectangular configuration might be possible. However, this type of scaling has not been given a large amount of consideration, as square scaling will be sufficient to demonstrate the principle.

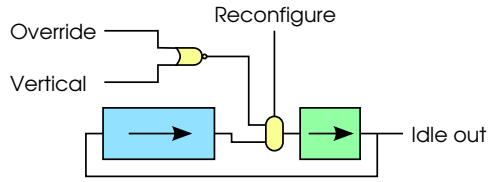

Figure 4.10: Runtime manager

Figure 4.10 illustrates the internal structure of the runtime manager. The runtime manager uses the data being communicated from the reconfiguration master

to the reconfiguration readers to determine the amount of idle waves that should be generated between each computation wave.

The *Vertical* signal in the reconfiguration reader bus is used to determine which cells in a cell column should be configured, and which should be dummies. When deasserted, the cell should be configured with a dummy configuration. When asserted, the cell should receive real configuration. By inverting this pattern, the asserted periods would correspond to the idle wave pattern.

During reconfiguration a new idle pattern must be loaded into the runtime manager. The first vertical pattern signal is transferred from the reconfiguration manager to the first reconfiguration reader, before the new configuration is loaded into the grid. The runtime manager is constructed with the delay between the command to reconfiguration reader, and the first finished cell reconfiguration in mind.

The current idle wave pattern is stored in a shift register. The output of this shift register is connected to *Idle out*, which initiates idle waves in the grid. The output is also connected to the shifter input, which prevents the idle wave pattern being lost after a given amount of shift operations.

The delay needed before a new pattern is shifted into the grid is achieved by splitting the shift register in two pieces. This is shown in figure 4.10 as the green shift register and the blue shift register. When a new pattern is to be loaded the connection between these registers is broken by a MUX, and the input of the green register is connected to the *Vertical* input. The size of the green shift register decides the delay before the new idle wave pattern will reach the *Idle out* output. The content of the blue shift register will be lost, as the output will be unconnected during reconfiguration. When the new pattern has filled both shift registers, the connection between them is restored. That way the new pattern is retained.

There is one additional signal, which is the *Override*. This signal is connected to the *Flag* output of the reconfiguration master. This flag can be used to override any idle wave pattern generation. This will increase throughput when a configuration that does not have any external feedback loops is scaled, as the idle waves are not required.

The *Idle out* output is connected to the cell where all waves originate. From this cell the idle signal is distributed along the regular grid interconnect, at the same speed as data propagation.

Chapter 5

Implementation: Device version 2

In this chapter the second version of the device is presented. This includes a new cell version, that has the ability to perform spatial branching, and an extended reconfiguration system.

5.1 Cell version 2

The major cell extension added in version 2 is the ability to perform spatial branching. To create an efficient hardware structure that would support such branching, it is paramount to specify which branch types that should be supported.

The basic idea behind spatial branching is the ability to conditionally activate a path through the device grid. A simple approach to this is conditionally setting the output valid flags of a cell. The cells in a path will then be missing a valid input and will thereby not perform any computation.

This simple approach might be considered to be limiting. A simple example of this is comparing two values in a cell. The larger value should leave the cell through one output, and the smaller value should leave through another output. This is not possible with the simple conditional valid approach, as it can only perform a conditional validation of the value that is already being forwarded to the output port by the internal cell interconnect. To achieve this the paths in the internal cell interconnect must be changed based upon predetermined conditions.

Conditional data output from a cell is only a part of the problem. A cell that is the end point of two conditional paths must be able to select which input that it should use. Fortunately, the constant propagation speed of the interconnect guarantees equal propagation time in all paths from one cell to another. Hence, if there are two paths from a cell two another cell, and the path taken is conditional based upon the input, only a single input to cell were the paths end will be valid at any time.

However, this might not be the case if there is a default path that data always flows through, and a secondary path that has higher priority. The cell where these paths end must then be able to select the higher priority path based upon the valid indication of the higher priority path.

A conditional path might contain some state that must be retained when the path is inactive. In the version 1 architecture, the state register is updated at every data transfer clock tick. If this happens in a path that is currently inactive, the state that should be retained might be lost. As such, the write operation should be controllable by the valid indication of the state input. Tying the write signal directly to the valid signal would fix this. However, this might not be the desired behavior in all configurations. Some configurations might use the state register as a delay register. Under this condition the state register should be updated, as in the version 1 architecture, which is at every tick.

A way of determining if a spatial branch should be taken has thus far been ignored. Using flags, such as zero and negative, generated from every computational unit in a cell might be appealing, as it is a very generic approach. The result of this approach would be a large number of available flags that could be used to branch. A large number of flags that could be used, leads to an increased number of configuration bits required to select which flag that should be used.

An alternative to this is letting only a single unit generate flags, and requiring that computational results that should generate flags should be forwarded to this unit. This simplifies the implementation, in addition to reducing the amount of flags that can be generated and thereby the size of the configuration. Therefore this approach was chosen, and the computational unit that will generate the flags is the ALU. To use this unit for flag generation it could be instructed to perform logic or, and putting the value that should be evaluated on both inputs.

5.1.1 Output valid

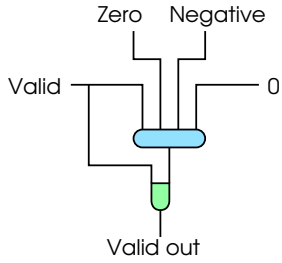

Figure 5.1: Conditional valid output

In figure 5.1 the hardware structure that supports conditional valid output is shown. This is connected to the cell output, where the original valid input signal is intercepted and connected to the MUX. The input to which the existing valid signal is connected to is not arbitrary. It must be connected to the input that will be selected if the MUX configuration consists only of zeros. The reason for this is

the configuration compatibility requirements, as all older configurations must be able to run on the newer device version if the older configurations are zero padded by the reconfiguration system.

The *zero* and *negative* input to the MUX is generated by the ALU. These will allow conditional validation of an output, based upon the whether the output value of the ALU is negative or zero. The last input to the MUX is fixed at logic zero. This can be used to force an output to never be valid. This will ease the input selection process.

The output from the MUX is connected to an AND gate, shown in green. This gate ensures that the output will never be valid if the output value is not valid in the first place. The reason for this is that the conditional output might be decided by the ALU based upon other conditions than the value that will be present at the output port. Hence, if the value at the output port comes from a source that is not valid, garbage data will be made valid by the ALU.

5.1.2 Output switching

Several possible implementations allow for conditionally changing the output interconnect based upon flags. They vary in hardware size, generality and most importantly configuration size.

Alternative 1

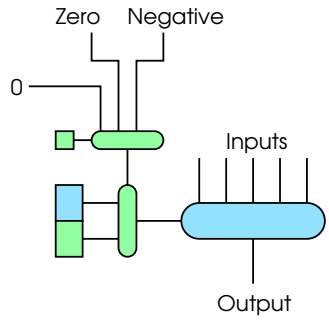

Figure 5.2: Conditional output switch (Alt. 1)

In figure 5.2, a simple approach to changing the output MUX configuration of a cell is illustrated. The blue elements in the figure represents the hardware that exists in cell version 1. The units shown is a single output MUX and the configuration register holding the configuration for this MUX.

The approach illustrated in this figure is based upon adding storage for another MUX configuration, and selecting which configuration that should be forwarded to the output MUX. Switching the configuration of a MUX, rather than inserting more MUXes on the output path, reduces the hardware area required. The reason for this is that the width of the configuration is narrower than the data width of the device.

Which configuration is forwarded to the output MUX is selected by the ALU flags, or can be tied to the default value with a secondary MUX. This select MUX requires 2 bits of configuration. If the configuration size of an output MUX is 3 bits, this approach would increase the configuration size with 5 bits per output port. Given that the input to the state register counted as an output the configuration would increase with a total of 15 bits.

Alternative 2

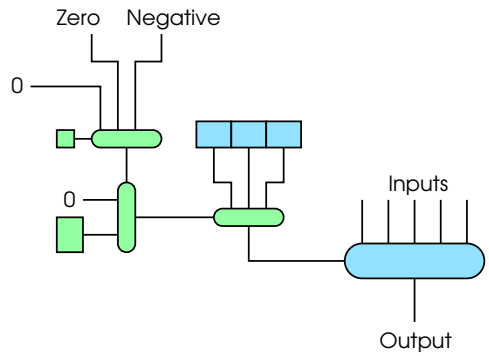

Figure 5.3: Conditional output switch (Alt. 2)

Alternative 1 can be improved upon by acknowledging that there already exists storage for 3 output MUX configurations within the existing configuration format. One of these can be selected based upon the ALUs flags. This approach is shown in figure 5.3. The components shown in blue already exists in the first version of the cell. The green components are added to support the selection of one of the 3 existing configuration. A MUX is connected to ALU flags and a default value of zero.

In stead of selecting one of the output MUX configurations directly, the flag MUX is connected to an intermediate MUX. The input to this intermediate MUX is which configuration of the existing configurations that should be selected. The 0 input on this MUX is used to select the default configuration for a given output MUX. This reduces the configuration bits required, at the cost of little generality. The second input on the intermediate MUX is used to select which of the other configurations that should be forwarded to a given output MUX based upon the flag input.

The cost of this switch alternative in configuration bits is 4 bits per output MUX. As the flag MUX requires 2 bits and the input to the intermediate MUX must be 2 bits large to select any of the other configurations. Given that the input to the state register counted as an output, the configuration would increase with a total of 12 bits. This is somewhat better than the previous approach, although it reduces flexibility.

Alternative 3

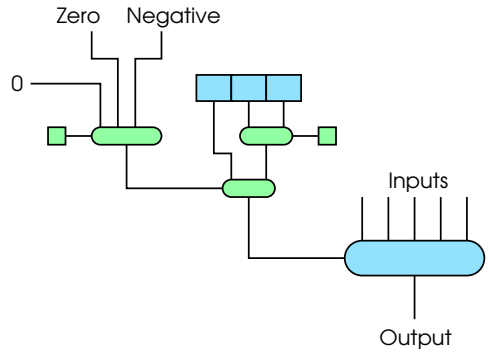

Figure 5.4: Conditional output switch (Alt. 3)

The previous output switching alternative has some redundancy in the configuration data used. The secondary configuration data, which can be selected by using flags, can select the default output switch configuration. This is unnecessary, as switching the default configuration with the same configuration based upon flags, does not change the outcome. There are two ways to eliminate this redundancy.

This redundancy can be removed by using the structure illustrated in 5.4. The regular flag selection MUX is now connected to a MUX that can select between the default configuration and one of the others. Which of the two other configurations that will be used is selected by a secondary MUX.

The result is that the configuration required for this secondary MUX is a single bit. Together with the flag selection MUX configuration size, this solution requires 3 configuration bits. Given that the input to the state register counted as an output the configuration would increase with a total of 9 bits.

There is an alternative selection structure that also uses a total of 9 bits of configuration that exploits the previously mentioned configuration redundancy. If the configuration data that can be selected by the use of flags in figure 5.3 (Alt. 2), is set to zero when output switching is not required. The 0 input to the flag selection MUX can be eliminated, as the flags will only switch the MUX between two zero outputs. Hence, the configuration selected by the configuration MUX will be fixed, regardless of the ALU flags. This reduces the configuration required in the ALU flag MUX, rather than the alternative configuration.

Alternative 4

The other approaches allow for the switching a single output configuration individually. However, if this is not required, the size of the configuration extension required for output switching can be reduced even further. The alternative is exchanging the configurations of two of the output MUXes.

A structure that allows this is illustrated in figure 5.5. In this approach, all the output MUXes will be tied into the same structure in stead of duplicating a

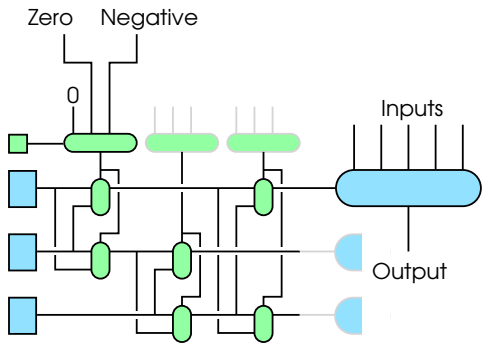

Figure 5.5: Conditional output switch (Alt. 4)

simpler structure. A single output MUX is shown in figure 5.5. The connection points of the two others are indicated, by the blue grayed out partial MUXes.

The standard ALU flag selection MUX is used in this structure as well. The output of this MUX is connected to two minor MUXes. When the output of the flag MUX is asserted both these minor MUXes will switch which input that is forwarded. This will in effect exchange two configurations. The output of these minor MUXes is connected to a second pair of minor MUXes that is driven by a second ALU flag selection MUX. This second MUX is shown in green and gray. This second pair of minor MUXes can exchange one of the results from the first selection with the third output MUX configuration register. This is again connected to a third pair of exchange MUXes. This leads to a priority based configuration exchange, which allows for any permutation of the existing configuration data.

The resultant structure is somewhat complex, and the RPU programmer must be aware of the priority between configuration exchanges, as the outcome of one exchange will affect the input configuration to other exchanges. However, this might not be a problem as only one of the available ALU flags can be asserted at any time. Hence, this only becomes complex if several exchanges is triggered by the same ALU flag.

The structure uses three ALU flag selection MUXes. Each requires 2 bits of configuration. The configuration extension total comes to 6 bits with this approach. This is lower than all the other approaches. This reducing comes at the cost of some generality. Compared to the other alternatives this structure has longer paths, and can affect the critical path of a cell. Despite these problems, this approach was chosen for implementation, because of the very low number of configuration bits required.

5.1.3 Input selection

The ability to select witch input a cell should use is paramount in this spatial branching scheme. The most powerful of the implementation alternatives, is similar to the output switching schemes. Only that the switching would be done based

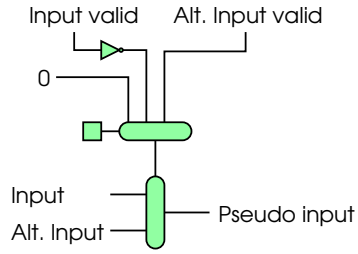

Figure 5.6: Input selection

upon the valid flags of the inputs, and all the input MUXes to the computational units would be subject to change.

This would, with the most configuration size efficient output switch approach, require 8 configuration bits. In the outlined spatial branching approach, where only a single path is activated at any time, only a single input to the end cell of a path will be valid at any time. By acknowledging this the approach illustrated in figure 5.6 can be utilized. This approach requires half the configuration size of the output switching scheme. This is although quite limited in the possible switch methods possible.

The structure is inserted at the cell input, in such a way that the value that would be read from this input port is intercepted and fed through the structure. Both cell inputs is actually connected to this structure. The other cell input is connected to the *Alt. Input* port.

Which of the inputs that should be forwarded to the *Pseudo Input* is based upon their respective valid flags. The structure can be configured to never perform a select, and always forward the local input. This is the default setting. Selecting the *Alt. Input* can be done based upon one of two criteria. The first is selecting the other cell input when the one the structure is connected to is not valid. With this selection policy, the local input has priority. The other selection policy is using the other input when it is valid. This gives the other cell input priority.

The *Pseudo Input* is connected to the rest of the computational units of the cell. From the computational units point of view, data will come from fixed input point, as they are not able to detect whether the data is silently replaced by the data from another input. As such, no changes to the computational units input MUXes are required.

5.1.4 State write policy

To retain state in a conditional path it is important to be able to control when writes to the internal cell state register occurs. The write policy employed in the version 1 architecture is writing at every data transfer tick. The structure illustrated in figure 5.7 makes the write programmable. Writes can either be triggered at every data transfer tick, or when the valid indication of the data on the state input port is asserted.

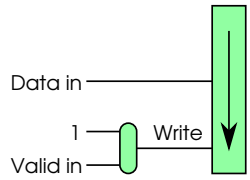

Figure 5.7: Write policy selection

This structure alone is not very flexible. However, the *Valid in* could be driven by a conditional valid structure. This would allow writes based upon ALU flags.

5.1.5 Cell version 2 architecture

Figure 5.8 illustrated the internal cell version 2 architecture. All components shown i gray exists in the cell version 1 architecture. It should be noted that some of the components might not be found in the cell version 1 figure, as they have been omitted for clarity. More specific, the omitted components from the version 1 figure is registers that hold configuration. A few of these must be shown in the version 2 figure as their contents is manipulated, by the outputs switch structure.

The blue components perform the input selection operation. There is one for each cell input of these structures. Their function has been described in *Input selection*. The yellow components is used to exchange the configurations of the output MUXes based upon ALU flags. This is the output switch alternative 4 structure previously described. The green components are used for conditionally setting the output valid, the state input valid flags, and specifying the state register write policy. These components has previously described in *Output valid* and *State write policy*.

The RPU programmer should be aware that the conditional valid structures are connected directly to the outputs of the cells. As such, any configuration exchange on the output MUXes, will be using the existing port valid output policy.

5.2 Reconfiguration system version 2

The reconfiguration system must also be changed to support the new cell version. The reconfiguration readers and writers must be able to load the larger configurations used by the new architecture. They must be able to extend version 1 configurations, in such a way that they are loadable by the version 2 cell, to maintain binary compatibility.

5.2.1 Reconfiguration master

In figure 5.9, version 2 of the reconfiguration master is presented. The parts shown in gray also exists in the first version of the reconfiguration master. Colored components are added or changed to support the new system version.

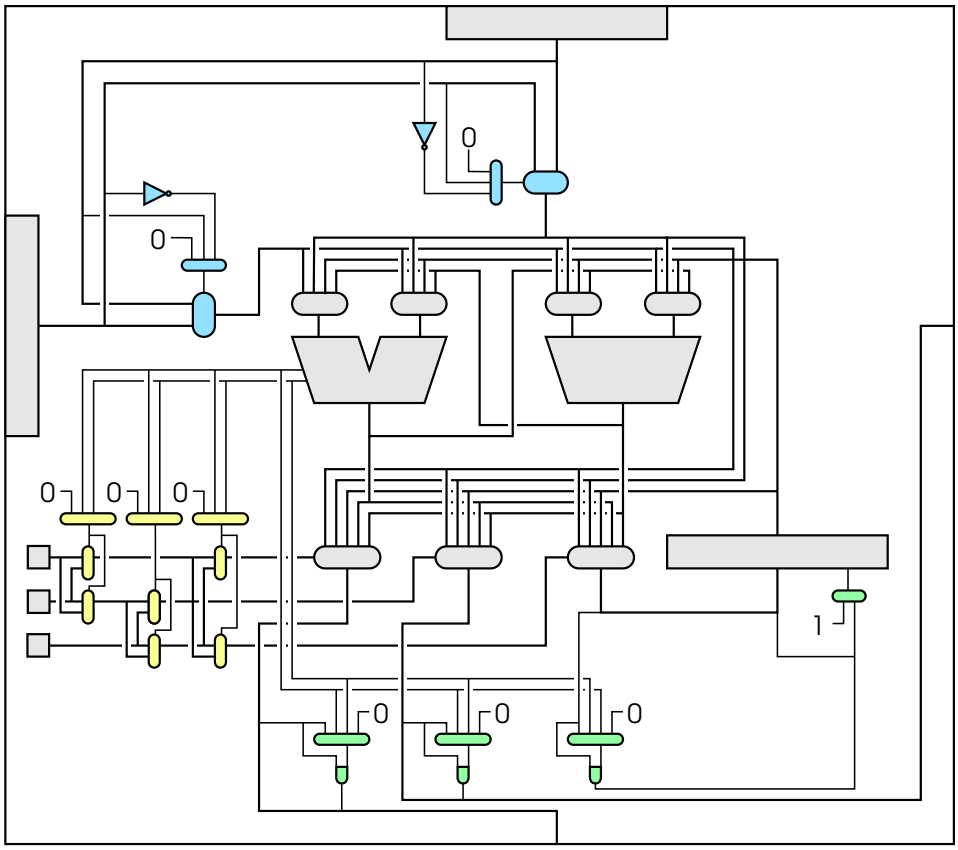

Figure 5.8: Cell version 2

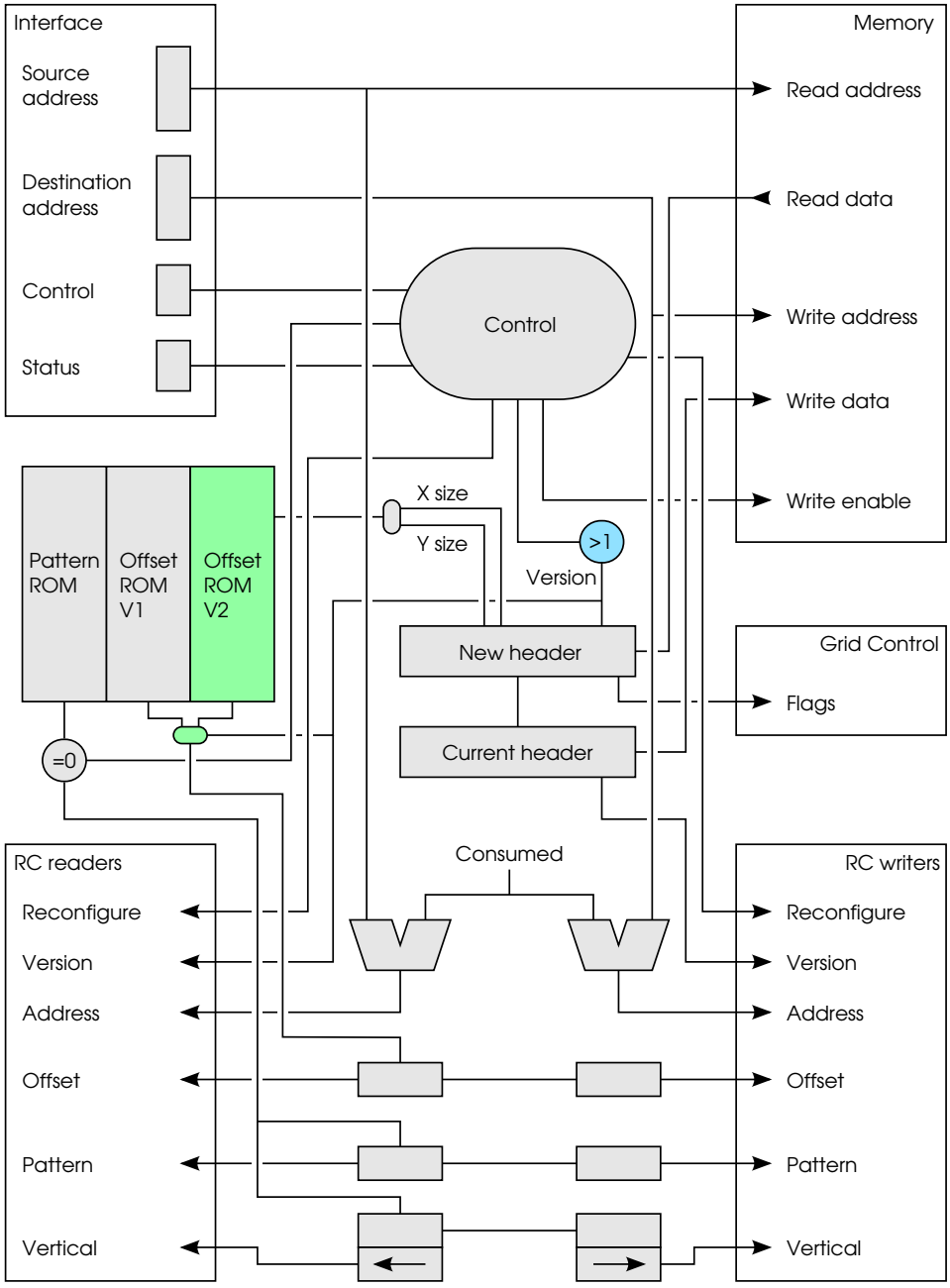

Figure 5.9: Reconfiguration master version 2

The comparator used to verify the new configurations header number have been changed to signal the control unit if the version is above the increased range of supported version numbers.

A second *Offset ROM* has been added to the structure. This is used to lookup the amount of the memory a reconfiguration reader will consume when loading a version 2 configuration into a cell column. The amount of memory consumed will differ as the version 2 configuration is larger than the version 1 configuration.

Which of the ROMs that are used is decided with a MUX hardwired to the version field in the *New header* register. As such, no changes has to be made to the control unit.

Actually the contents of the second version ROM could probably be calculated based upon the contents of the version 1 lookup table. There is a linear relation ship between the number memory addresses consumed when configuring a cell column with a version 1 configuration, and the number of memory addresses consumed when configuring with a version 2 configuration. Hence, multiplying the the result of a lookup in the version 1 table with a suitable constant would yield the version 2 sizes. Even though this multiplier could be optimized as it has a fixed multiplier, the ROM would probably be smaller and faster. However, if the number of versions become large adding an additional ROM for each might not be viable.

In addition to these changes an additional signal is added to the *RC readers* and *RC writers* interfaces. This is labeled *Version* in figure 5.9. This is used to indicate the version the configuration that should be loaded or stored. The reconfiguration readers and writers must take appropriate actions based upon this signal.

5.2.2 Reconfiguration reader

The reconfiguration reader must be changed to support loading of different sizes. One simplification has been done here to ease construction of the reconfiguration reader version 2. It is assumed that a single version 2 cell configuration is stored in memory with some padding. This removes problems that related to alignment. As such, it can be assumed that no cell configuration will begin in the middle of a data word.

With this assumption there is no need for hardware that can load a configuration that might start at an arbitrary bit within the read data and shift it to the correct position in the internal reconfiguration reader registers. Before loading the next part of the configuration. This assumption is unlikely to the true in a real computing system. However, the actual supported memory operations are memory system dependent. As the actual memory system is treated like a black box in this implementation, it will suffice that the number of memory operations performed by the reconfiguration reader differs with the configuration version.

In figure 5.10 a reconfiguration reader for the version 2 architecture is illustrated. The gray components are also present in the version 1 structure. Added or changed components are shown i color.

The reconfiguration register has been extended to match the size of version 2 configuration. The reconfiguration register input is connected to two separate

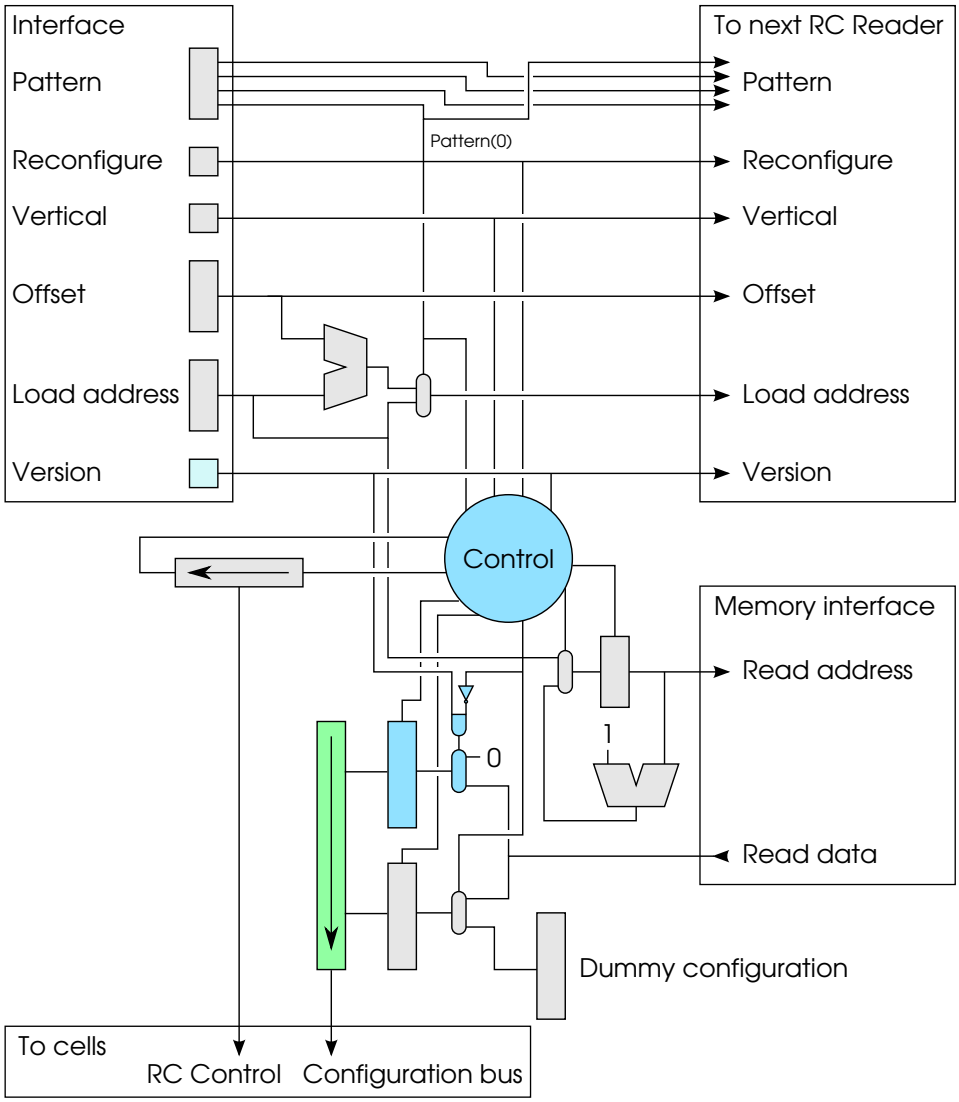

Figure 5.10: Reconfiguration reader version 2

registers. One of the registers can fit a version 1 configuration, and the other has room the version 2 part of a configuration.

When loading a version 2 configuration the control unit will initiate two memory accesses. The results of these memory accesses will be stored in these registers. Storing the data from memory in the correct register is achieved by controlling the write enable signal to each register.

Loading a version 1 configuration on the other hand requires only one memory access. However, the register holding the version 2 part of the configuration must be loaded with zeroes, to achieve backwards compatibility. This is done by a MUX connected to the input of this register. When the *Version* input to the reconfiguration reader is deasserted, a version 1 configuration is to be loaded. This signal is used to select a MUX input that contains only zeroes.

The last challenge is the loading of a dummy configuration witch is required for configuration scalability. This could be implemented by having a dummy configuration register that matches the size of the version 2 configuration. This can be improved upon, by acknowledging the fact that a version 1 dummy configuration that is zero extended should be a valid version 2 dummy configuration. So using the existing dummy configuration register, and forcing the input MUX, to the register that should hold the version 2 part of the configuration, to zero should suffice. This is obtained by the and gate and inverter connected to the version 2 register input MUX.

5.2.3 Reconfiguration writer

The reconfiguration writer must be altered to support the version 2 configuration. In addition, it should be able to filter away any zero padding inserted by the reconfiguration reader. This will allow a version 1 and a version 2 device to coexist in the same system, and the version 1 device would be able to load a version 1 configuration even after it has had an execution period on the version 2 device.

Version 2 of the reconfiguration writer is shown in figure 5.11. The gray components exist in the version 1 structure. The colored components are added or changed in the version 2 implementation.

The reconfiguration register is extended to fit the version 2 configuration format. The output from the reconfiguration register is connected to two registers. One for the part of the configuration that consist of version 1 configuration data, and one for the configuration data that only exists in version 2 configurations.

The control unit is extended to write the contents of these registers back to memory. Whether the contents of one register, or both registers are written back to memory is done based upon the *Version* input.

5.2.4 Reconfiguration register

During the testing of the version 2 implementation, a flaw in the reconfiguration register was discovered. In the version 1 architecture, the *write enable* input to the reconfiguration register is set at a fixed logic level immediately after the asynchronous load has completed.

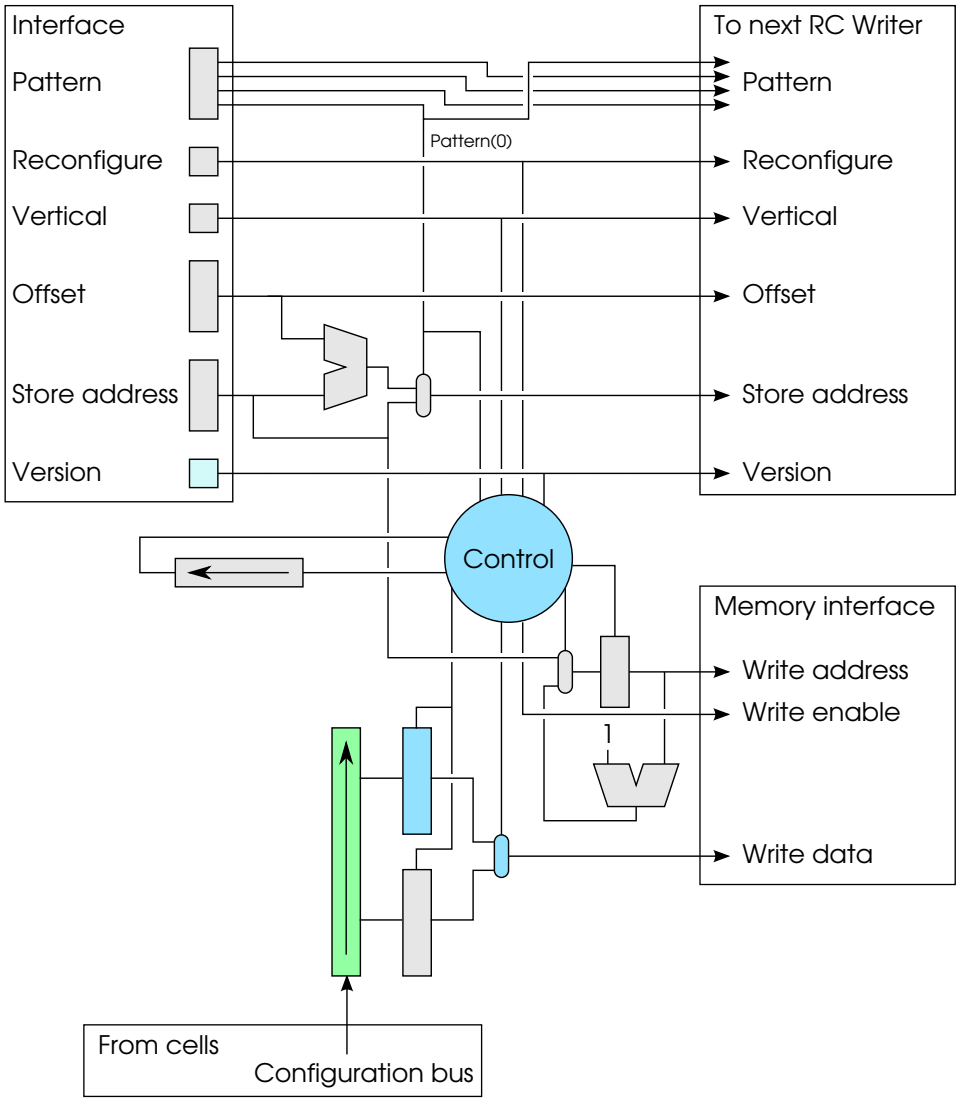

Figure 5.11: Reconfiguration writer version 2

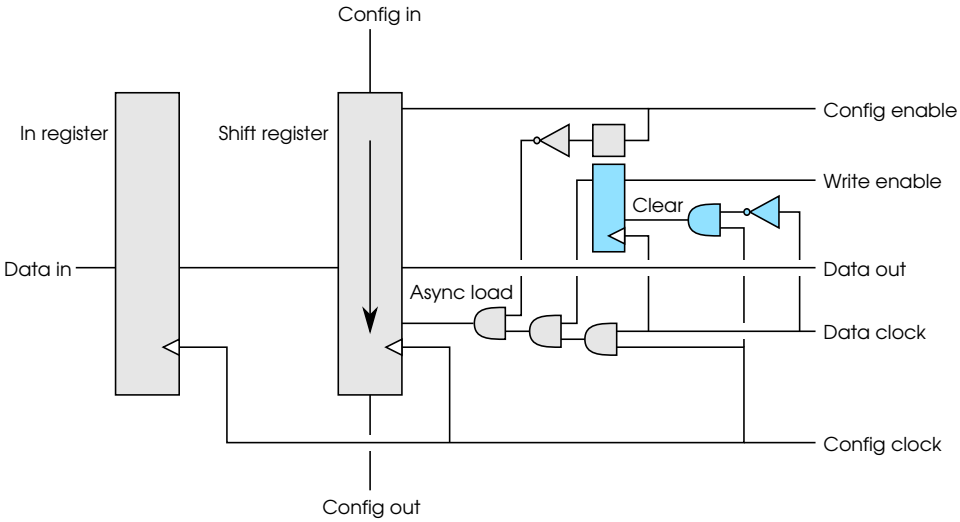

Figure 5.12: Reconfiguration register version 2

In the version 2 architecture the *Write enable* signal of the cell state register can be controlled by the ALU flags. When the ALU must wait for the combined multiplier and divider to complete the level of the *Write enable* signal can change very close to the rising edge of the data transfer clock. This is problematic with the version 1 implementation of the reconfiguration register.

With certain configurations the *Write enable* signal can be high past the last tick of the config clock, and then be driven low. The input register on the *Write enable* signal will then have a high to low transition at config clock tick that corresponds with the rising edge of the data transfer clock. This will cause a glitch in the *Async load* signal. As such, the contents of the input register will be loaded into the shift register. This is clearly not the intended behavior.

In figure 5.12, a structure that removes this glitch is illustrated. The components shown in gray exists in the version 1 implementation, and components shown in blue are added or changed. The input register on the *Write enable* is now driven by the data transfer clock. This ensures that the level of this signal will be stable during the *Async load*, however the high to low transition can still occur with this clock source.

Therefore this register has an asynchronous clear as well. This signal is labeled *Clear* in figure 5.12. This signal is generated by combining the data transfer clock and the config clock, in such a way that the register will be cleared when the config clock is high and the data transfer clock is low. As such, the *Write enable* will always be low during the computation, and the actual level of the *Write enable* signal will be stored at the rising edge of the data transfer clock. This eliminates any high to low transitions at the rising edge of the data transfer clock, and thereby the potential for glitches on the *Async load* signal caused by this is removed.

Chapter 6

Implementation: Other

In this chapter an assembler and a inherently parallel assembly language for the device is presented.

6.1 Assembler

To aid in testing of the RC device, an assembler for the architecture was developed. The assembler uses a custom made assembly language and is described in section 6.1.1. The internal software structure of the assembler is described in section 6.1.2.

6.1.1 Language

The input language is not an instruction stream as in most general purpose processor assembly languages. The language is a spatial description of the operations that is to be performed, and takes the form of a cell by cell description. The language grammar is located in appendix C.

The language itself was in general designed to be expandable and not be tied into the actual hardware implementation. Therefore, the language supports certain types of descriptions that is not supported by the current hardware implemented. However, such features might be supported in the future. There are some aspects of the language that might seem to be tied into the current hardware implementation, such as label names. The list of such label names might be extended upon when needed.

A simple example

```
1 cell[1][0] {
2     south = north;
3 }
```

Listing 6.1: Spatial description

```
1 cell[0][0] {
2     aluout = north + west;
3     east = aluout;
4 }
```

Listing 6.2: Using the ALU

In listing 6.1 a very simple assembly example is shown. This example will generate a cell configuration that connects the north input to the south output, the rest of the cell will be configured with default configuration. On line 1 the position of the cell within the grid is specified. The position is specified on the form `[y][x]`. As such, position of the cell will be on the second row of the first column of a 2 dimensional grid.

The language specification has no limits on the number of dimensions. If the target RPU has 3 dimensions, the most significant dimension would be added to the front of the dimension specification. The resultant position specification would then be `[z][y][x]`.

On line 2 of listing 6.1 the `north` input is connected to the `south` output with a simple assignment. Most statements in the language are on this form. There is no support for variables. Hence, the target of an assignment must be one that is defined by the cell hardware.

Using the computational units

In listing 6.2 an assembly example employing the cell ALU is shown. The statement on line 2 will be mapped to the ALU. The operation that will be performed is the addition of the `north` and `west` inputs.

The target of the assignment selects which computational unit that should perform the specified operation. Explicitly specifying a target in this manner allows for several computational units that can perform the same operations to be present within the same cell. An example of this would be a cell with two ALUs. The target statement would specify which of the ALUs that should perform a given operation. Automatic mapping of such operations to suitable computational units could be done. However this was deemed to be the task of a compiler, that would operate on a language of a higher level than this assembly language.

On line 3 of listing 6.2 the output of the ALU is assigned to the `east` output. As the result of the ALU operation is assigned to the pseudo target `aluout`, the result of the ALU operation can be assigned to several outputs. This eliminates the need to restate the ALU operation on each output. In addition to simplify the assembler implementation, as there is no need to search the statement list, and check that all the ALU operations assigned to a single ALU is identical.

In listing 6.3 an example of cell operation specification that feeds on computational unit with the output another computational unit is shown. The operation performed will be $2 \times$ `north` $\times$ `west`.

On line 2 the familiar ALU is shown. However, on line 3 the output of the

```
1 cell[0][0] {
2     aluout = north + north;
3     mulout = aluout * west;
4     south = mulout;
5     east = aluout;
6 }
```

Listing 6.3: Chaining units

```
1 cell[0][0] {
2     aluout = north + state;
3     state = aluout;
4     south = state;
5 }
```

Listing 6.4: Computation with state

ALU statement is used as input in a multiplication operation. The result of this multiplication is assigned to a pseudo-target, like the ALU output, named `mulout`. On line 3 this pseudo-target is assigned to `south`, which in effect connects the output of the multiplier to the south output.

The assignment performed on line 4 is somewhat noteworthy. The intermediate value passed from the ALU to the multiplier is connected to the `east` output. The effect of this will be that the value on the `east` output will be $2 \times$ `north`, while the value on the `south` output will be $2 \times$ `north` $\times$ `west`.

Cell state

The cells presented in previous chapters also have a state register. The assembly language has several statements that was added to support the various ways of using this state register.

Listing 6.4 illustrates computation that uses the state register of a cell. The resultant configuration will accumulate the values of the `north` input, the pre-addition accumulated value will be present on the `south` output.

On line 2 the state is used as any other source for computation. However, on line 3 the result of the ALU operation is assigned to the state. This does not lead to a runaway operation where the value of the `north` input is continuously added to the state register. The state register value will only be updated when new data has arrived at the `north` input. This is enforced by the register clocking described in section 4.2. Line 4 illustrates that the state can also be connected directly to an output.

The example shown in listing 6.4 does have a flaw. When the computation starts for the first time, the value of the state register would at best be zero. The value is, however, not likely to be valid, so the ALU will never perform any operation. Which causes the state register to be updated with a new value that is not valid.

```
1 cell[0][0] {
2     aluout = north + state;
3     state = aluout;
4     south = state;
5     init {
6         state = 34;
7     }
8 }
```

Listing 6.5: State initialization

```
1  cell[0][0] {
2     state = north;
3     south = state;
4     write( state ) = always;
5  }
6  cell[0][1] {
7     state = north;
8     south = state;
9     write( state ) = valid;
10 }
```

Listing 6.6: State write policy

Therefore it is required that the language supports a way of specifying the initial value of the state register.

The solution is presented in listing 6.5. The init block, on line 5, specifies the initial value the state register should have. This assignment also instructs the assembler to indicate in the resultant configuration bitstream that the state value is valid.

In version 2 of the architecture it is also possible to specify how the state register should be updated. This is described in section 5.1.4. The language support for the write policies of the state register is shown in listing 6.6. In the cell starting at line 1, the state register will always be written to, regardless of the input validity. This is equivalent to the version 1 write policy.

In the cell starting at line 6, the write policy is set to be dependent on the valid flag of the state input. In effect the value of the state register will only be updated when the `north` input is valid.

Branching

The spatial branching implemented in the version 2 of the architecture allow for a variety of settings in relation to output switching, forcing of valid flags, and selecting inputs.

```
1 cell[0][0] {
2     aluout = north + west;
3     south = north;
4     valid(south) = negative(aluout);
5     east = west;
6     valid(east) = never;
7 }
```

Listing 6.7: Valid setting

```
1 cell[0][0] {
2     aluout = north + west;
3     state = north;
4     south = west;
5     if ( zero(aluout) ) {
6         switch(state,south);
7     }
8 }
```

Listing 6.8: Output switch setting

In listing 6.7, the language counterpart to forcing of valid flags is shown. The valid flag output can be tied to the actual signal valid, it can be dependent upon ALU flags, or it can be forced to never be valid. On line 4 of listing 6.7 the valid signal on the `south` output is set to be dependent upon whether the result of the ALU operation is negative or not. On line 5 the `west` input is connected to the `east` output. However, line 6 sets the valid policy of the `east` output to never. Hence, the `east` output of this cell should never be valid with this configuration, regardless of the `west` input.

The version 2 implementation also support the switching of output configurations. Listing 6.8 shows the assembly language support for such operations. During normal conditions the outlined assembly would cause the `north` input would be stored in the state register, and the `west` input would be forwarded to the `south` output. However, when the result of `north` + `west` is 0, the statements beginning a line 5 will come into effect. When the pseudo-target `aluout` is 0, the configurations of state register input and `south` output will switched. Which will cause the `west` input value to be stored in the state register, and the `north` input value will be forwarded to the `south` output.

In one such if/switch statement, several switches can be specified that will be caused by the same flag. An RPU programmer should be aware of the priority the hardware imposes upon such switches. When two switches are triggered by the same `aluout` condition, and these two switches operate on the same target, one of the switches will have priority. The effect would be that the configuration of two outputs will be switched first, and then the second switch will operate on the result

```
1  cell[0][0] {
2      east = north;
3      south = west;
4      if ( not valid(north) ) {
5          north = west;
6      }
7      if ( valid(north) ) {
8          west = north;
9      }
10 }
```

Listing 6.9: Input select setting

of that switch.

The last of the branch related operations supported by the hardware is the selection of an input based upon their respective valid flags. The language support for such operations are shown in listing 6.9. The standard operation for this configuration is connecting the `north` input to the `east` output, and the `west` input to the `south` output. However, the statements starting at line 4 specifies the input select policy. When the `north` input is non-valid, the `west` input will be used instead. So if the `north` is not valid the `west` input will be connected to both the `south` output and the `east` output. When the `north` input is valid, the statements beginning at line 7 will come into effect, and the `north` input will be connected to both outputs.

Putting it all together

A more functional example of the assembly language is shown in listing 6.10. The resultant operation performed by the 2x2 cells, is taking the absolute value of all input values received at the `north` input of `cell[0][0]`, and checking if this absolute value is larger than any previously received value. The largest absolute value received during operation is output through the `south` output of `cell[1][1]`.

The operation performed by `cell[0][0]` is checking whether the input value is positive or negative. If the value is negative the `south` output will be valid. The `east` output will always be valid. As such, the inputs to the end cell of the conditional path might be both be valid at some point. The solution to this problem is located in `cell[1][1]`.

`cell[1][0]` will only have a valid north input when the input value to `cell[0][0]` was negative. Subtracting this value from 0, will yield a positive result of the same magnitude. This 0 value is stored in the state register. Note that the state register input is connected to the state register output, so that this value will not be overwritten by garbage data.

`cell[0][1]` on the other hand is a simple dummy cell, only present to move data from the input cell to the last cell in the chain.

The problem of choosing the input value remains. By default all numbers will

```
1  cell[0][0] {
2     aluout = north or north;
3     south = north;
4     east = north;
5     valid(south) = negative(aluout);
6  }
7  cell[1][0] {
8     aluout = state - north;
9     east = aluout;
10    state = state;
11    init {
12       state = 0;
13    }
14 }
15 cell[0][1] {
16    south = west;
17 }
18 cell[1][1] {
19    if( valid(west) ) {
20       north = west;
21    }
22    aluout = state - north;
23    state = north;
24    init {
25       state = 0;
26    }
27    valid(state) = negative(aluout);
28    write(state) = valid;
29    south = state;
30 }
```

Listing 6.10: Largest absolute value

pass through `cell[0][1]`, and reach the `north` input of `cell[1][1]`. Negative numbers on the other hand will pass through `cell[1][0]` and reach the `west` input of `cell[1][1]`. The path of the negative numbers will only be valid when a negative number has been converted to a positive number. As the propagation speed of the two paths are equal, the `west` input will be valid when in fact the `north` input is negative. Selecting the `west` input when valid, will ensure that only positive input data will reach cell `cell[1][1]`.

In `cell[1][1]` the current largest value received is stored within its state register. This value is compared with the input value by the ALU. When the input value is larger than the current state value, the result of the subtraction performed will be negative. This combined with the valid policy and the write policy of the state register, will cause the input value to be written state register when it is larger than the current state value.

6.1.2 Software architecture

The software architecture of the assembler resembles traditional compiler design. The software is divided into a front end and a back end. The task of the front end is reading input files, generating a machine understandable representation of these, and verifying that the representation is valid. The back end uses this machine understandable representation to generate an output bitstream than can be used to configure the RPU.

Front end

The main software components of the assembler is illustrated in figure 6.1. The front end consists of 3 major components, which is shown in yellow and green.

The first component in the structure is the lexical analyzer. The role of this component is reading the input file, and generating tokens based upon this file. Tokens are string of characters with some predefined meaning within the implemented language. This component was implemented by using Fast Lexical Analyzer (FLEX). Which is a tool for generating C code that can perform lexical analysis on a specified input language.

This component is labeled *Lex* in figure 6.1, and is connected to the second component which is the component labeled *YACC*. This is the syntactic analyzer. This component checks that the token stream received from the lexical analyzer corresponds with the language grammar. In addition to generating a machine understandable representation of the input tokens. This component was implemented by using a Yet Another Compiler-Compiler (YACC). YACC generates C code that can read tokens generated by lexical analyzer, and check that these correspond to a grammar specified on a BNF-like form. The YACC implementation used in this particular implementation was Bison.

The last unit in the front end of the assembler is the *Analysis* module. This unit receives a list of cells and their corresponding statements from *YACC* module. It then performs rigorous analysis of the statement list of each cell. The purpose of this analysis is to uncover if some invalid combination of statements has been given

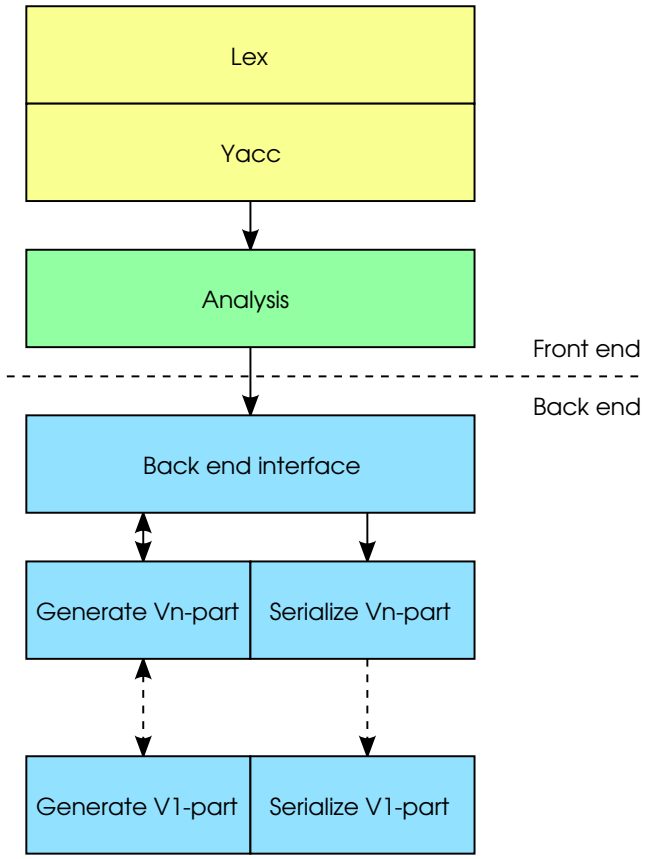

Figure 6.1: Assembler architecture

as input to the assembler. This can be duplicate assignments to the same target, invalid select combinations, duplicate valid policy specifications and so forth. This analysis process will also label all valid statements with a device version number required to support it.

Back end

The assembler software structure, shown in figure 6.1, includes a back end. The task of the back end is generating a configuration bit stream based upon the cell and statements lists passed from the front end to the back end.

The back end consists of a *back end interface* that is connected to the front end. The data structure used in this interface resembles physical layout of the device. The front end places the cells in their correct position in this data structure before passing it to the back end. The *back end interface* then simply makes a pass over this data structure, generating a part of the configuration from the cells statement lists.

The configuration generation is a two-stage process. As shown in 6.1 the *back end interface* can be connected to a chain of other modules. This chain will contain a module of each device version supported by the assembler. When a new device version is specified and assembler support is required a new module will simply be inserted into this chain closest to the *back end interface*.

The chaining of the modules allow a module of higher version to modify the input data to the next module in the chain. The effect being that a high version module can modify the input data to a lower version module. This might be required if some extension of the hardware requires that a part of the lower version configuration has a specific value for correct operation.

An example of this might be if the opcode of the ALU is extended with an additional bit in a newer version. This new bit will be set by the high version module. However, the lower version module will still generate the configuration for the other bits in the ALU opcode. The high version module must therefore insert new data in the input data to the lower version module to ensure that the ALU will perform the intended operation.

The first stage of configuration generation is populating data structures based upon the statement list of a single cell. This is done by passing the statement list to the first module in the chain. This will traverse the statement list and search for all statements that have a version number matching that chain module version. It will translate the matching statements into a proper representation for configuration bitstream generation.

All statements of lower version than the current module will be passed to the next unit in the chain. Which will perform similar operations. When the end of the chain has been reached, the newly populated data structures will be passed back to the *back end interface*. The data structures could be modified by higher version modules when the data structures are returned. However, this is discouraged as it would require that a module understands the data structure of a lower version module.

The *back end interface* uses these data structures to generate the configuration bitstream in a process similar to the statement translation. The chain modules also have the ability to generate their part of the configuration bit stream based upon their own data structure. The populated data structures are therefore passed back into module chain, and the configuration bit stream for a cell is generated by traversing the chain.

The reason for returning to the *back end interface* before starting the actual configuration bit stream generation is the desirable bit order. Generating the configuration bit stream in a big endian fashion, with the configuration belonging to the highest version closest to the Most Significant Bit (MSB), allows for easier interfacing with the implemented hardware modules and their testbenches.

Chapter 7

Results and testing

In this chapter the testing procedure and synthesis results for the implemented device is presented.

7.1 Testing

Thorough testing of the implementation is paramount to discover any flaw in the hardware description.

7.1.1 Component testing

To test the individual components extensive use of both automatically generated test vectors and manual test vectors have been used.

The manual test vectors perform very basic tests that might not be covered by automatically generated vectors. Such tests might be testing that all signals are connected, checking that the basic operation is as intended and some corner cases.

The automatically generated vectors are used to perform extensive testing of the components. The advantages of the automatically generated vectors is that large amounts of these can be generated in a short amount of time. These can also uncover potential problems that the hardware designer did consider during manual test vector generation.

The automatic vectors are generated by a component specific program. This custom program generates a text file containing the vectors. This text file is then read by a automatic testbench. This testbench applies the vectors from the file to the unit under test, and checks that the response corresponds with the contents of the text file.

The entire test procedure was automated. The manual test vectors was stored within a different testbench than the automatic. This was required as the manual testing might check some behavior more closely, such as reset behavior. Both these testbenches would then be executed by an automated test system that uses

the Xilinx ISim simulator[1] combined with Makefiles. This allows for automated testing to performed on all the components. This approach can the be used for easy discovery of faults in one component as the result of a change in a subcomponent. For closer inspection of faults Mentor Graphics Modelsim[2] was used. An overview of the tests performed can be seen in appendix D.

7.1.2 Device testing

The exception from the outlined approach was the testing of the entire unit. At this level the function of the RPU is quite complex. Generating manual test vectors for this would be a quite extensive time consuming undertaking. The amount of vectors required for testing would also be immense. Developing a automated test program for it was also abandoned as an unfeasible option as it would also be very time consuming.

Normal testing

In *Multitasking on a reconfigurable computing system* [Oft09], a software simulator is presented. This was developed to check the soundness of reconfiguration waves. In this software simulator, a basic reconfiguration system is simulated that is connected to cells that are treated as black boxes. The reason for this was that the computational properties of the device was not established at that time. As such, the cells can perform any function.

Simply setting the software simulator cells to perform the same operations as the implemented cells, would in effect turn the software simulator into a simulator of the implemented device. However, some changes had to be made to the simulator. The simulator was built to verify the reconfiguration wave, and generate statistical data based upon the properties of the wave based approach. Therefore it would require some modification is generate test vectors suitable for design verification.

The modified simulator structure is presented in figure 7.1a. The reconfiguration system is shown in yellow. This part of the simulator differs from the reconfiguration system implemented in hardware. The difference being that it only has a single unit per cell column, while the implemented hardware has two units. In addition the hardware reconfiguration system is controlled by the reconfiguration master. No such clearly defined unit exists in the software simulator.

The software simulator also includes a memory system, which is shown in dark blue. This memory system is a simple vector based memory system. Between the memory system and the cells, which are shown in gray, interceptor code has been inserted. This interceptor code will store all data going into and leaving the cell grid, in a text file. This interceptor code is presented with light blue, in figure 7.1a.

In addition to the output generated by the interceptor code, some data is collected from the reconfiguration system. This data indicates when system simulator initiates a reconfiguration wave.

[1] ISim 11.5
[2] Modelsim SE PLUS 6.3f

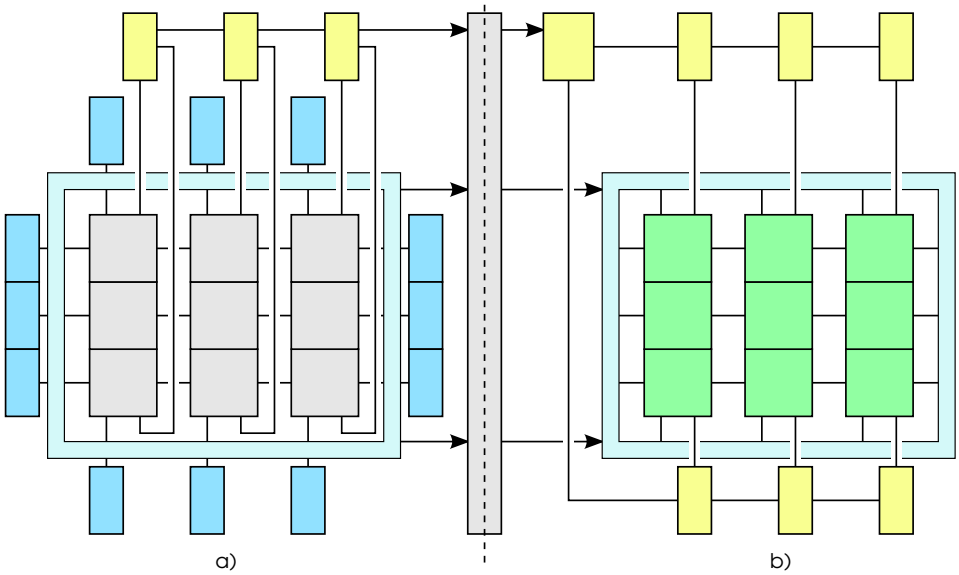

Figure 7.1: System simulator based implementation testing

In figure 7.1b the hardware test method is shown. The cells are shown in green. The cell grid is fed with the data the software simulator interceptor stored. This stored data is also used to govern the actions of the reconfiguration system, which is shown in yellow in figure 7.1b. This causes the implemented reconfiguration system and the software simulator system to perform reconfiguration at the same place in the test vector stream. With this approach the data leaving the grid in both the software simulator and the simulated hardware should be identical.

Identical data is actually not achievable. During reconfiguration the simulated hardware generates random outputs on the cells being reconfigured. Cells being reconfigured does not have the valid flag set on their outputs. So as long as the valid flags matches the valid flags generated by the software simulator, and the outputs with the valid flags asserted, matches the expected output data, it can be concluded that the hardware operates as expected.

Test of scaled configurations

Testing of scaled configurations has been treated as a special case. The easy way to do such testing is scaling the configuration on both the software simulator and the simulated hardware implementation. This will however, not reveal if there is any differences between running a configuration on hardware of correct size and a larger grid. It will only show if the scaled hardware will operate equally wrong when on both systems.

The solution to this was running configurations on a correctly sized grid in the software simulator. With this solution, the configuration should not suffer any

computational effects caused by the scaling of the configuration. However, when scaling and using idle waves the speed of the computation drops. As such, the speed of the output comparisons should be reduced. When scaling a configuration the grid inputs that will be connected to cell that contains a portion of the real configuration will also be scaled apart.

The simple solution would be to scale the test data apart with a separate program, that would fill in dummy data between the real data. This is actually not as easy as it sounds. The reconfiguration system can still react at the same speed to commands as when running a non-scaled configuration. As such, scaling the test data in both time and grid position becomes hard.

The chosen solution to this was to modify the system simulator. The computations will still be performed on a grid that matches the configuration size. However, the interceptor code will scale the data before it is stored to file. As the interceptor code also has access to the current state of the reconfiguration system in the device, it can also move the data that will be applied to the reconfiguration system of the simulated hardware in the text file, so that reconfiguration will be started at the correct time.

This approach does have a drawback. It requires that the software simulator be run twice, if a configuration is to be scaled to two different sizes. However, as the software simulator is quite fast this is considered to be a large problem.

Configuration generation

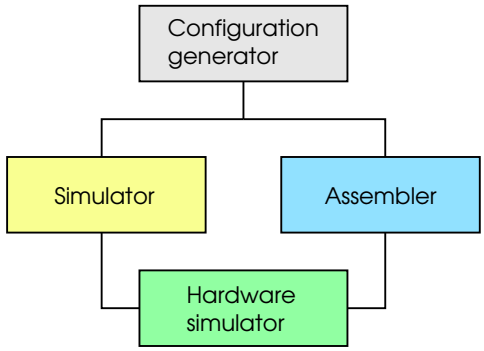

Figure 7.2: Configuration generation

Until now the actual configurations that are used during testing has not been mentioned. These are automatically generated by the process shown in figure 7.2. The *Configuration generator* entity automatically generates a configurations. These configurations are output on two different formats. One on the assembly language used by the hardware implementation, and one suitable for loading into the software simulator.

The configurations is run on the software simulator and the output is stored as previously described. In addition the configurations are translated by the *As-*

sembler. The output of the processing being bitstreams that can be loaded by the simulated hardware. This is implemented by simulating simple block RAM that is connected to the reconfiguration master, reconfiguration readers and reconfiguration writers. The configurations are loaded into this memory by the simulator before the testing start.

Actually several such block RAMs are connected to the implemented structure. One to the reconfiguration master, and one for each cell column. This allows for parallel memory access, between the components, and as no part of the configuration will ever be loaded by a different column this will approach suffice.

7.2 Synthesis results

In this section synthesis results of the implemented hardware will be presented. The target FPGA for all the results presented here is a Virtex 5 LX330. The reason for using a Virtex 5 as a target and not a Virtex 6, which is the newest Virtex-series FPGA, is a limitation in the FFs employed by the Virtex 6 series. Between the Virtex 5 and Virtex 6 series an input has been removed from the FFs. This input allows for the instantiation of a FF with both asynchronous set and reset. This type of FF is used by the reconfiguration register to achieve asynchronous load. Although the synthesis tools can work around this when a Virtex 6 is targeted, the resultant implementation is suboptimal. Hence, the Virtex 5 family was chosen.

All results are taken from reports generated by the place and route process. This is the last step in the process and should be the most accurate.

The results are from synthesizing the device with a cell data path width of 16 bits.

7.2.1 Size version 1

In figure 7.3, the increase in FFs and LUTs as larger cell grids are synthesized is shown. Both appear to be growing at a linear rate. To verify this the same data normalized to the number of cells in the grid is shown in figure 7.4. This shows that the growth of the is not quite linear. The number of FFs and LUTs does drop a considerable amount before stabilizing at a lower level.

This is probably caused by the overhead created by the reconfiguration system. This overhead does not grow at the same rate as the cell grid. As an entire cell column is added when a single reconfiguration reader and writer is added. As such, with large cell grid sizes the reconfiguration system overhead becomes very small. To the point where it can almost be neglected.

7.2.2 Clock speed version 1

In figure 7.5, the resultant clock speed estimates are shown, as the grid size is increased. The number are as mentioned taken from the post place and route report. They were obtained by synthesizing with constraints on the clock domains that matches the relationships between these clocks.

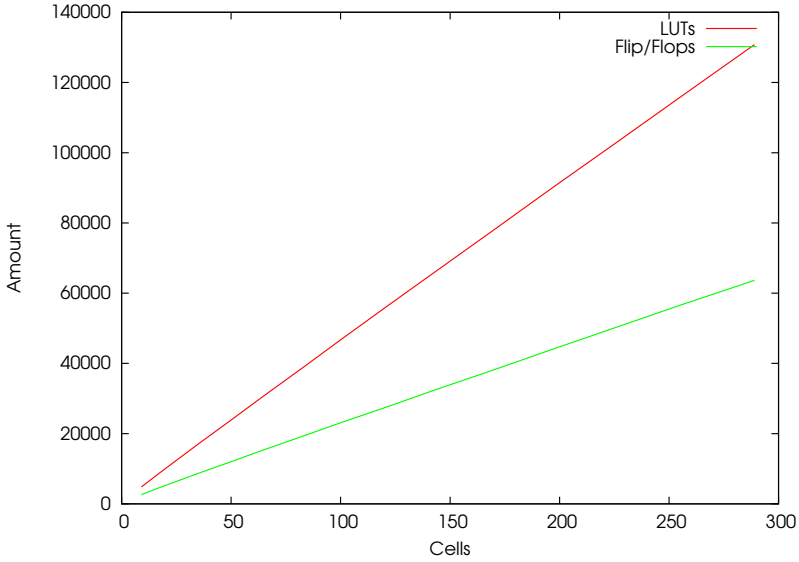

Figure 7.3: Resource utilization (version 1)

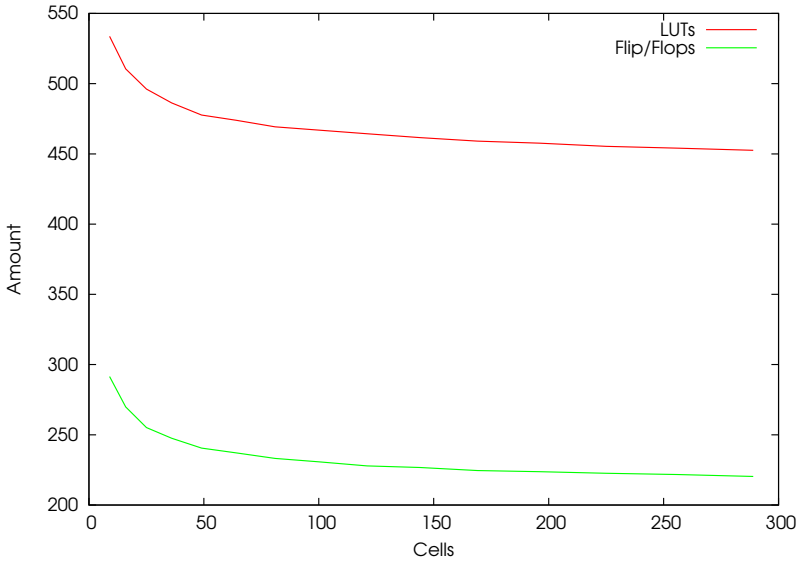

Figure 7.4: Normalized resource utilization (version 1)

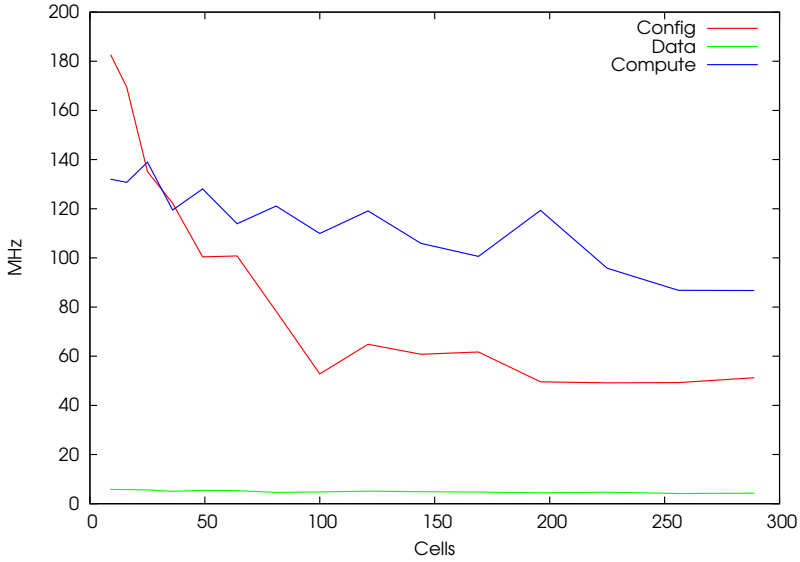

Figure 7.5: Clock speed estimates (version 1)

The results are however not as expected. There is a drop in the speed of the computational clock as the grid size grows. This should not happen as there are no paths in the computational clock domain that should grow with the device size. This might be caused by the optimization strategy employed by the synthesis tools. During the synthesis runs a focus on minimal area has been chosen. This might cause the resultant circuit to reuse hardware across cell boundaries. This is not intended.

To test this the synthesis was rerun with some constraints on optimizations. More specific, the synthesis tools were restricted not to remove the concept of a cell by setting the hierarchy constraints on this design entity.

The resultant speed estimates can be viewed in 7.6. The computational clock remains stable as the number of cells increases. This indicates that intended cell structure with synchronous data transfer has been implemented as intended, and there exists no paths governed by the computational clock that grows with the number cells.

The estimated configuration clock speed remains static at very small grid sizes, between these two synthesis runs. Unfortunately, it has a worse fall in the estimated speed that the other synthesis approach.

7.2.3 Reconfiguration bus speed exploration

The drastic reduction in the speed of the configuration bus warrants closer examination. To check if this was caused by the optimizing for area, the synthesis process was rerun with speed as the optimization target. The results are shown in

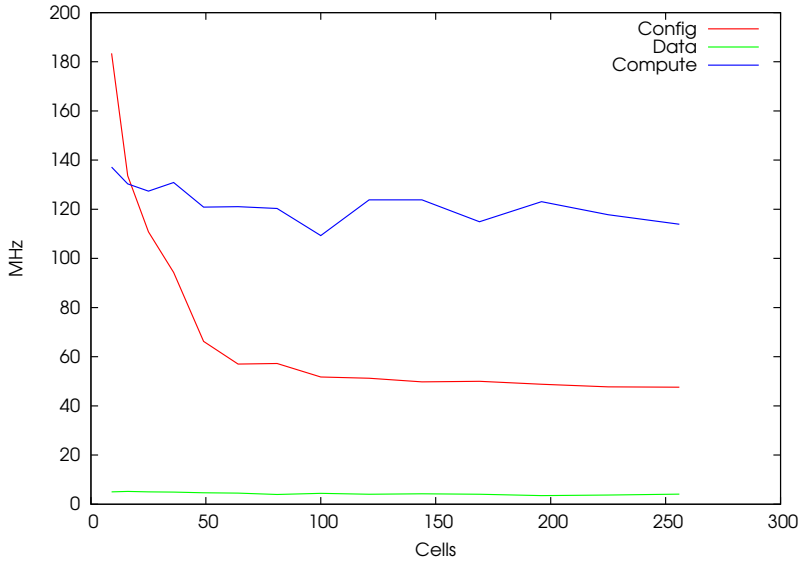

Figure 7.6: Clock speed estimates with cell hierarchy (version 1)

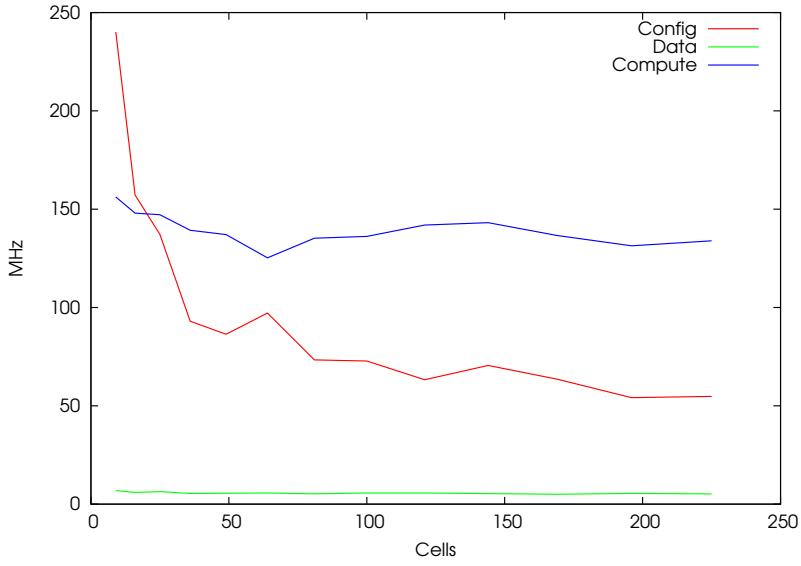

Figure 7.7: Clock speed optimized results

figure 7.7. The clock speeds are in general higher. However, there is still a drastic reduction in the configuration clock speed as the grid is scaled up.

After analyzing the critical paths within the configuration clock domain the cause of this was uncovered. The MUX that connects the configuration bus input to the configuration bus output within a cell, begins to dominate the achievable speed. The reason for this is that a MUX is inserted into the configuration bus in every cell, and this generates some delay in each cell.

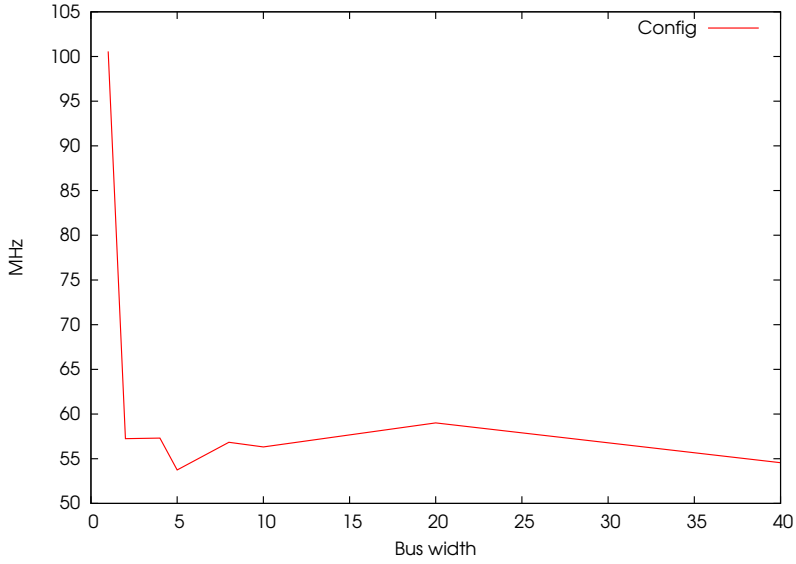

Figure 7.8: Reconfiguration bus speed

The impact of the reconfiguration bus width should therefore be investigated. In figure 7.8, the configuration clock speed is examined as the width of the configuration bus is scaled on a 8x8 cell grid, with a fictive configuration size of 40 bits. This shows that a completely serial bus will achieve the highest clockrate. Although the bus employed in the design is narrow, it is not serial.

It should be checked if this has caused a performance reduction. This can be done by evaluating the number of configurations that can be transferred over the bus in a given period of time. This is shown in figure 7.9. Throughput is given in mega-configurations per second.

This graph indicates that in a FPGA environment a wider bus will actually increase performance. Further, no performance has been lost by using a narrow bus instead of a true serial bus. Whether this is true for a Application-specific integrated circuit (ASIC) implementation of the outlined RPU is unclear.

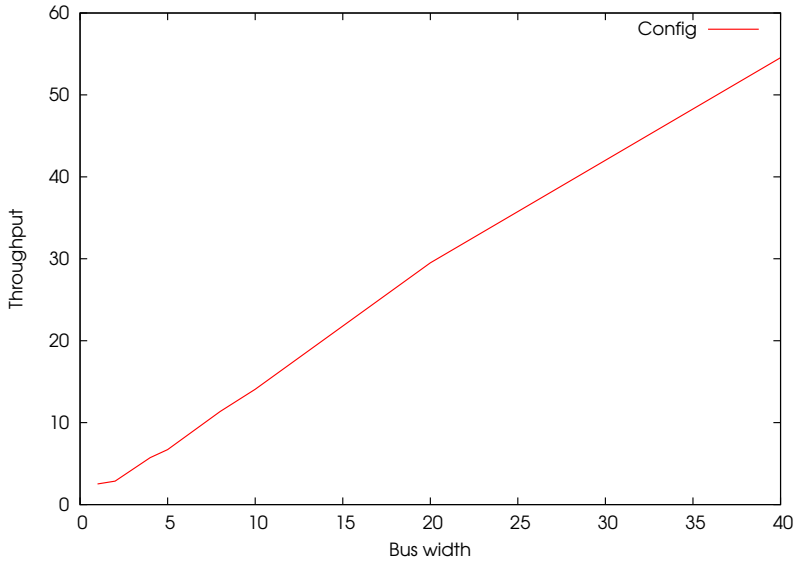

Figure 7.9: Reconfiguration bus throughput

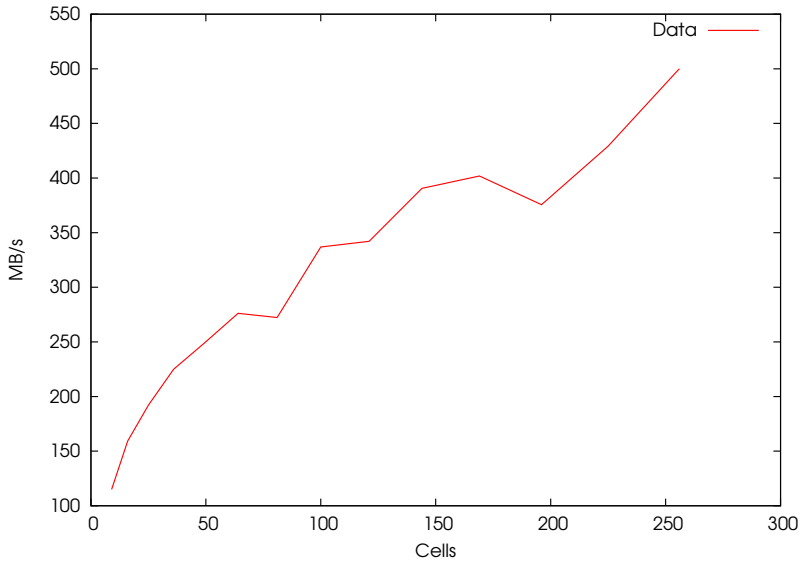

Figure 7.10: Data consumption development

7.2.4 Data consumption development

The large number of cycles required on the configuration and computation clocks between each tick of the data transfer clock is of great concern, as the data transfer clock governs the throughput of the device. Therefore the bandwidth the RPU will consume as the device size is scaled up has been investigated. This was done by multiplying the estimated data transfer clock speed with the amount of input and output port on the cell grid and the width of a single port. The result is shown in figure 7.10.

The result is that even though there is a drop in the data transfer clock as the device is scaled up, the overall bandwidth required is considerable, considering that the data transfer clock speed is in the sub 10MHz region.

7.2.5 Size version 2

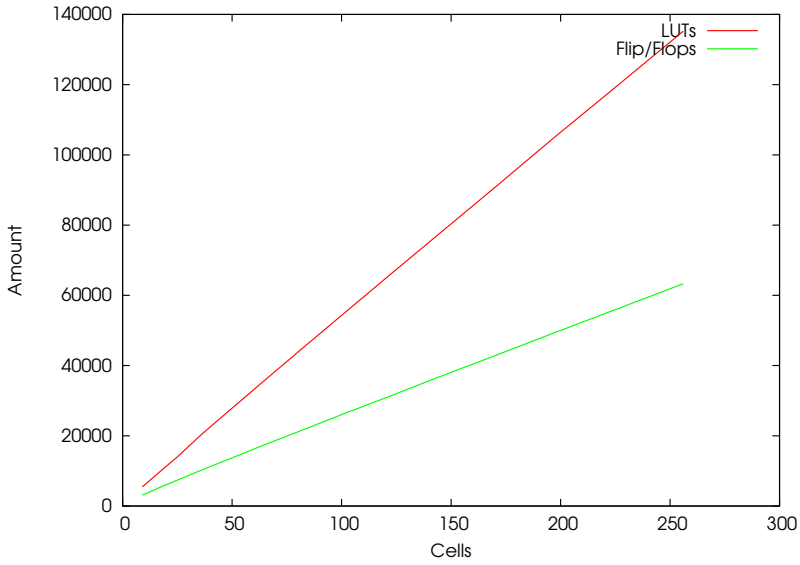

Figure 7.11: Resource utilization (version 2)

In figure 7.11, the size of the version 2 device is shown as the number of cells is increased. This has the same growth as the version 1 device.

The same data normalized to the number of cells in the device is shown in figure 7.12. The device roughly exhibits the same growth pattern as the version 1 device. The added circuitry added in version 2 does elevate the growth pattern with about 50-100 LUTs and 25-50 FFs per cell.

There is a small discrepancy on the graph when the grid size is 25 cells. This will be elaborated upon in the next section.

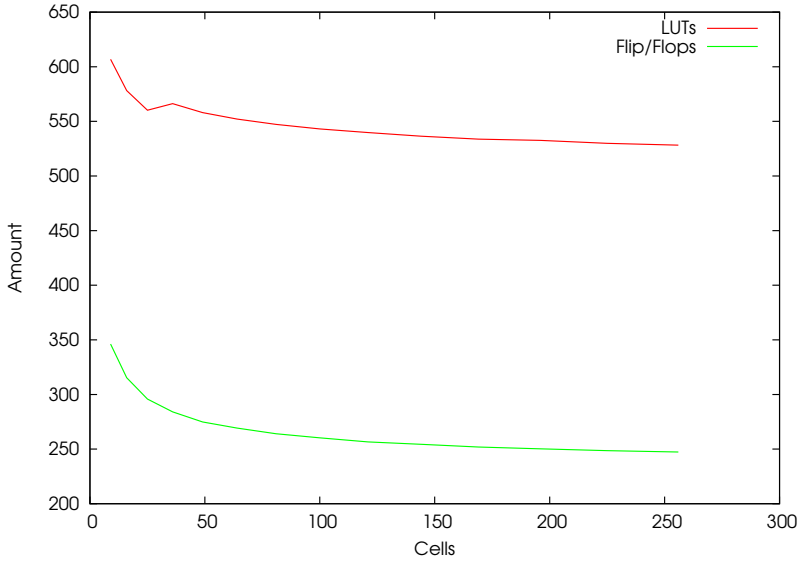

Figure 7.12: Normalized resource utilization (version 2)

7.2.6 Clock speed version 2

In figure 7.13, the speed estimates of the version 2 device is shown as the grid size is increased. The results are not as expected. The most surprising result is that there is a significant drop in the speed of the configuration clock, when the grid contains 25 cells.

However, if the point of the speed drop is compared with the normalized size graph of the previous section there is a similar occurrence there. Hence, it is reasonable to assume that the drop in the configuration clock speed is the result of an area optimization that are possible on a 5x5 cell grid. This minor saving comes at great cost in the configuration clock speed, which is not desirable.

The same drop in computational speed as with the version 1 device occurs here. Hence, the synthesis tools might performs the same area optimization operations. The synthesis was retried with the same hierarchy constraints that was employed in the version 1 speed evaluation.

The result of this resynthesis is shown in figure 7.14. The speed of the computational clock now remains rather stable as the grid size is increased. However, the same drop in the configuration clock speed occurs.

7.2.7 Speed comparison

In figure 7.15, the change in clock speed between the two versions is illustrated. The speed of the various clocks are generally lower in the version 2 design. This is as expected as the size of the design is larger.

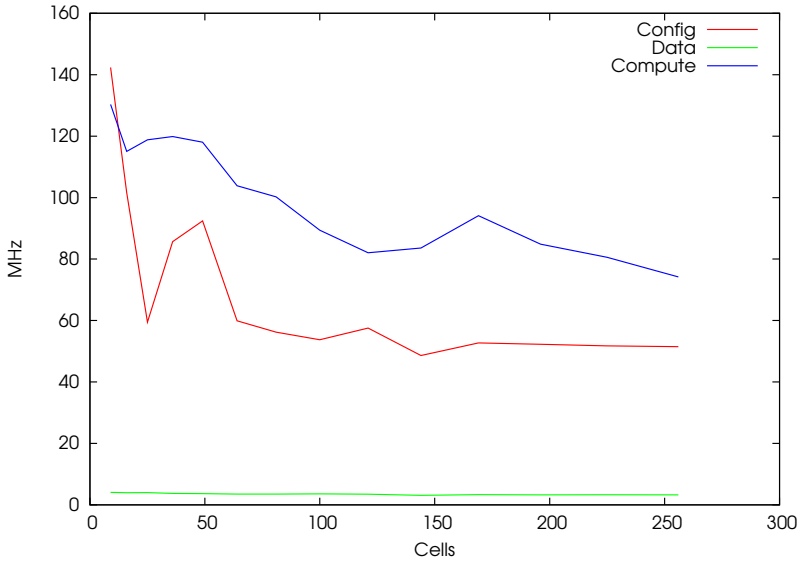

Figure 7.13: Clock speed estimates (version 2)

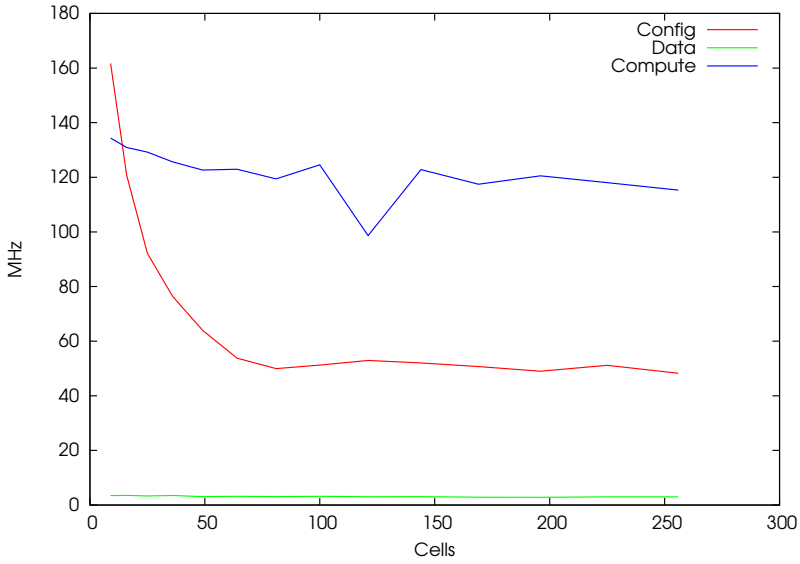

Figure 7.14: Clock speed estimates with cell hierarchy (version 2)

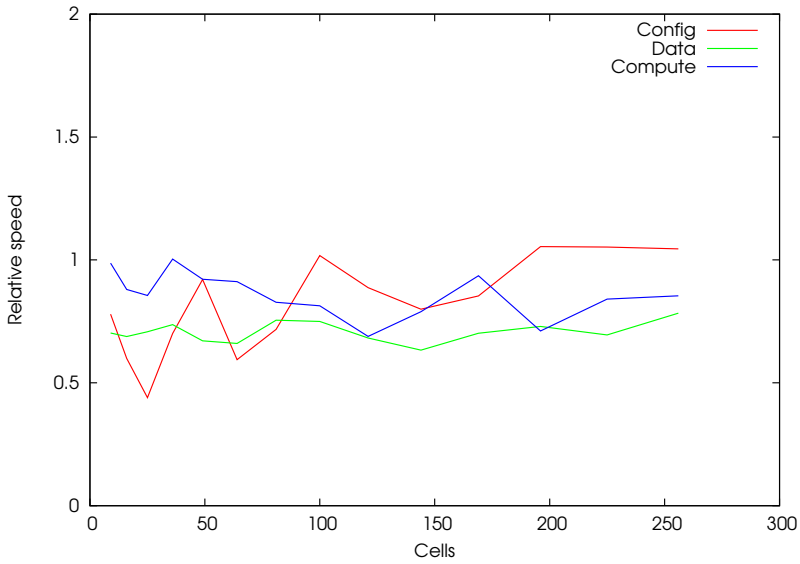

Figure 7.15: Relative speed change

7.3 Other results

Here some of the results that are based upon the device properties and not the results off a synthesis run is presented.

7.3.1 Configuration size

In figure 7.16, the configuration sizes of the two versions are compared. These include the state part of the configuration, as it can be used during initialization of a cell. This state size is set at 16 bits.

The configuration size per cell remain small, even though the growth in size is substantial.

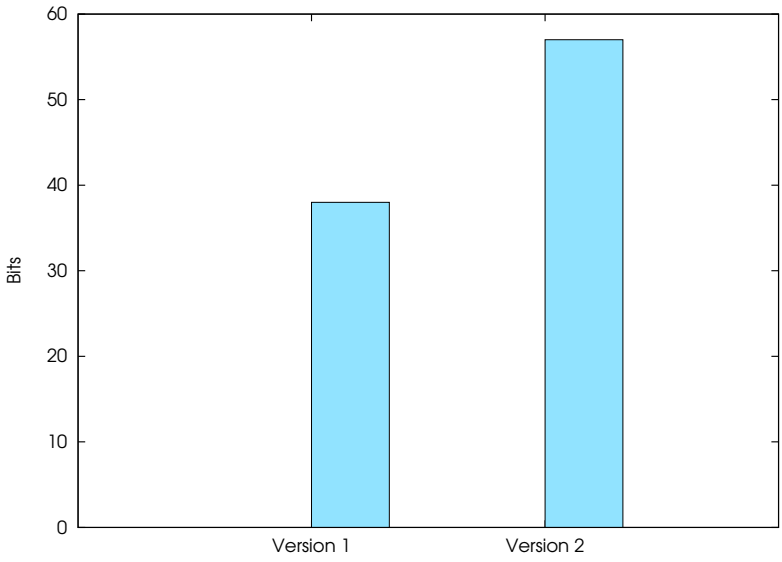

Figure 7.16: Cell configuration size development

Chapter 8

Discussion and conclusion

8.1 Device performance

The data transfer clock speed of the implemented device is shown in section 7.2 to be rather low. This is caused by the restrictions imposed on this clock by the multicycle operation of the configuration bus and the multicycle computation. The largest contributor to this is the multicycle combined multiplier and divider. Redesigning this into a unit that requires more area and less clock cycles might be beneficial for the entire design efficiency.

It was also shown in section 7.2.4 that even with a low speed data transfer clock, the data bandwidth of the device should not be neglected. A large cell grid will even in the sub 10MHz region require a memory bandwidth of several hundred megabytes per second, if the grid is fully utilized. This supports the previous statements with regards to the requirement of a high-speed memory system.

However, any estimation of the device performance can not be estimated based upon the bandwidth requirement alone. Although units based upon spatial computations tend to have lower clock rates than units based upon temporal computing, they can beat temporal computation devices by allowing for massive parallel operation, and allowing for more operations that better fit the problem at hand. So even with the slow data transfer clock speed, a device, with an extreme number of cells, could potentially beat a temporal computational unit in peak performance.

The pitfall here is that the data transfer clock has a large impact upon the latency of the device. As such, the low speed data transfer clock, will probably ruin any hope of meeting some low real time computational demand. To increase the speed of the data transfer clock, a beneficial approach might be to reduce the size and thereby complexity of the cells. This would reduce the number of cycles required to perform computation and is likely to increase the data transfer clock speed, given that the problems related to the configuration bus can be resolved.

As mentioned, the largest contributor to the data transfer clocks slowness from the cells is the combined multiplication and division unit. Removing this unit will probably remove a large amount of the impact on the data transfer clock that can

be contributed to the cells. However, the ability to perform multiplication, and thereby multiply accumulate is important within digital signal processing.

An alternative to removing the multiplication and division unit is reconsidering the employment of a data transfer clock. The data transfer clock provides a fixed deadline for computation completion within a cell. However, this is a worst case deadline. Even the slowest computation, which is combining the ALU and Multiply Divide (MD) unit, is able to complete within the allotted time.

If no cells in the device combines these to units internally in a cell, the cells will be idle for a cycle of the computational clock every data transfer tick. This becomes even worse if only the ALU is used for computation, as this unit is much faster than the MD unit, it will idle for extended periods every data transfer clock tick.

One alternative to this is to remove the demand that computations must have completed before the deadline, and rather require that the computational unit must save their state every data transfer clock tick. If the cell encounters a reconfiguration wave at this data transfer clock tick, this state can be saved and later restored. When the state is restored the interrupted computation can be resumed. However, this will increase the amount of state that must be stored, as the amount of intermediate results that is required to resume computation later, can be large. An example of this would be the implemented multiplier and divider. One of the operands must be stored, and the contents of a shift register, that is twice the width of an operand, must be saved.

This is quite large compared to the size of a single cell configuration. The configuration sizes of a single cell is shown in section 7.3.1. These configuration sizes assumes that a 16 bit dataword is employed. With this data word size the amount of state that must be saved to resume an interrupted multiplication operation would be 48 bits. This is larger than an entire version 1 cell configuration. Hence, this is likely to negatively impact the reconfiguration time of a single cell, in addition to increasing the load on the memory bus during reconfiguration with a substantial amount.

Another alternative is to remove the data transfer clock entirely and using asynchronous communication. With asynchronous grid communication, the slowest cell in a path governs the speed of the entire path. As such, the worst case will roughly be the computation times the slow data transfer clock enforce. However, if chaining of internal cell units is not performed, the performance is likely to increase. Although there will be some additional overhead during communication and in the required hardware resources.

This would require solutions to the problems that wave propagation on an asynchronous grid can cause. These problems mostly relate to propagation speed, and thereby data exchange between cells. A handshake based data exchange protocol, like the one employed by WAP, could be used to provide secure data exchange between cells. However, it must then be extended to provide information to the reconfiguration system about the state of a cell, so that the reconfiguration system can determine when it is safe to reconfigure a cell.

A naive approach to determining this is to check if the input ports of a cell

contain no unprocessed data. This might not be sufficient as the a cell might be configured to generate data without external input. An example of this would be a pseudo-random number generator. Once initialized with a seed, no external input is required. Such cells can not be blindly reconfigured with a new configuration when encountered by an asynchronous reconfiguration wave, as the cells that receive data from this cell might require more data before they can safely reconfigured. Hence, the state of the input ports to the connected cells must be checked.

To further complicate this, one could add input buffers to the cell inputs. This could increase efficiency as a cell sending data does not have to wait until the receiving cell have completed its current computation and is ready to receive new data. However, the reconfiguration system must be able to determine when these input buffers have emptied, or only contain data that belongs to the configuration being loaded. This could be implemented with a simple tag on the data in the buffer.

8.2 Computational properties

The computational properties of the implemented proof-of-concept device have their root in a classic temporal computing device. As such, these might not be the optimal operations for a spatial computing device. In essence each device cell can perform the core operations of a general purpose processor.

However, the power of reconfiguration computing device comes from removing generality from the circuitry. With this perspective it could be concluded that the design of an individual cell is flawed. The reason being that the cells contain two very generic computational units. Such units could be optimized in a device that supports fine grained configuration, as constants could be used to optimize the design.

The entire device on the other hand is inherently pipelined. As such, it allows for the creation of deep specialized pipelines. The power of this heavy pipelining should mitigate the effect of the cell generality. Although less general hardware would probably achieve even higher performance.

However, these pipelines are word oriented. This is unfortunate, as this imposes some of the computational limitations of general purpose processors onto the device. To mitigate this, support for carry chains could be added. One cell could use the carry out from a neighboring cells as carry in when performing computation. This would allow the device to perform operations on data that exceed the width of a single cells data word.

8.3 Spatial branching

The spatial branching scheme presented is quite alluring because of the way it fits into the spatial computation model. However, it might not be the best branching method possible based upon efficiency. The efficiency problem being caused when a configuration contains a branch that exclusively selects a path the data should

follow. That is, a branch where only a single path is used at any given time. The opposite being the comparison of two values that would each be sent down a different path based upon the outcome of the comparison. Then both paths would be active at any given time.

The exclusive path selection based upon branching, causes a cell in the selected path to be active. However, the cell in the path not chosen that is affected by the same computational wave, will idle as there is no valid data flowing into it, at this time. As such, the throughput in the hardware used to implement exclusive branch paths is halved.

The alternative would be to use a branch approach that is closer to temporal computation. In essence, this would change the the operations performed based upon flags set by a cell in the beginning of a conditional computation path. This can be viewed as folding the two exclusive conditional paths into a single path through the grid. Folding the paths would require a larger amount of configuration per cell, as the actions of both paths are now contained within a single super-path. How much the configuration would increase is uncertain, as it would be anywhere between a few bits, and a doubling of the original configuration size.

It could be argued that doubling of the configuration size is not all that bad, as this amount of configuration would be required anyway to implement the separate paths. In effect only hardware resources have been saved. However, if the entire unit is considered it is unlikely that all the cells would make use of second branch activated configuration. As such, the extra configuration room would only increase the amount of configuration that needs to be loaded.

This effect can mitigated somewhat by allowing two branch-less configurations to exists within the device. This would turn the RPU into a multi-context device. However, as previously mentioned, the amount of configuration required during reconfiguration effects the required memory bandwidth quite severely. The required memory bandwidth would still increase, as both branch-less configurations must be stored in memory when a configuration with branches is loaded.

Hence, if a temporal approach to branching is used it would probably be restricted to distorting the original cell configuration. As the spatial paths are folded into a single path, it is likely that they will use the grid interconnect in a similar way. Given this, the branch configuration only needs to change the operations performed by the internal computational units, and the interconnect between these units. This will reduce the amount of configuration required to perform exclusive branches. In addition to freeing up cells that would be used by one of the spatial branch paths. These can the be used to perform other operations.

However, this approach does not supersede the spatial approach in every possible way. The spatial branching approach is not limited to the exclusive path branch style. It can also process two elements in parallel in different paths, and decide which element that should be processed in the paths based upon input data. This can be achieved with the temporal based approach by letting the cell where the branch is decided to pass data to two separate paths, and in effect combining the two approaches.

Behind this argument, with regards to branch efficiency, is an assumption that

wave propagation speed is constant. This might not true in all cases. If non-uniform computation times are allowed, the processing of data elements performed within each branch of a spatial computation path, is likely to be more time consuming than deciding which path an element should enter. Hence, if the propagation speed of the elements slows down after entering the conditional part of the path, and the load on each exclusive path is even, the both paths can fill with elements. This will in effect be a rudimentary load balancing between paths, and can increase efficiency, as the number of idle cells is reduced.

A potential problem with this is the point where the two exclusive paths join. If this cell does not allow for fast consumption of the data provided by the paths, the performance gain would be reduced. Even if this does not happen undefined behavior might occur if the elements of one exclusive path overtake the elements in another exclusive path. This might happen if one of the path processes data much slower than the other.

8.4 Configuration scaling

In this thesis two configuration scaling method has been presented, which would allow a smaller configuration to run on a larger device. They differ both in configuration support and hardware requirements.

A common subset of both these approaches have been implemented in the device. However, this implementation has a negative impact on throughput on the device. The reason for this is that cells are forced to idle when data propagates through the unconfigured area. Both approaches have different ways of removing this requirement. These are however, approach device specific. To have the ability to test both approaches on the proof-of-concept device, the common subset implementation was chosen, even though it hurts performance.

However, both approaches have low efficiency when it comes to device utilization. When a configuration is scaled, a part of the cell grid is left unused in both approaches. Assuming that the data now forced to propagate through unconfigured cells, can be transferred to the memory system in a manner that does not involve the unconfigured cells, the device utilization can be improved.

The unused cells are now free to perform other tasks. These tasks could include running another configuration. This will require that two configuration can co-exist in the same grid. In addition, the configurations must be loaded into the grid together. That is with the same reconfiguration wave. This is not an absolute demand. However, it could be hard to initiate a reconfiguration wave at an arbitrary position within the device grid, when a secondary configuration is loaded. Further, the system must be able to select two or more configurations that fit into the device grid. This would probably be a task for the scheduling software.

Actually one of the implemented approaches does partially support multiple configurations. The grouped scaling approach will utilize an area in one corner of the device grid. Data leaving this configured area will travel vertical and horizontal in relation to the configured area. This leaves an area in the opposite corner of the device grid untouched. A second configuration can be loaded into this corner.

The primary configuration only uses one input and one output of the unconfigured cells. The second input and output of these cells can be used to transport data to the secondary configuration.

To load to configurations into the device in this manner the reconfiguration system must be reengineered to support the loading of two separate configurations, as they are likely to reside in different parts of memory, and might be of different sizes.

Given that these modifications was performed, the result would be a grid containing two configurations. Which also increases the amount of memory consumed by the grid. The effect of this could be that page misses, when accessing memory, would occur more frequently. This might be caused by two configurations with radically different memory access patterns.

The device would also have to guarantee that the amount of memory accesses will succeed for the secondary configuration. This is caused by the fact that a memory miss that does not affect the corner where reconfiguration waves originate will cause the grid to stall to prevent data corruption. Therefore a closer inspection of whether running two configurations on the same device would impede the performance growth this might achieve, as reconfigurations might be performed more frequently. This reconfiguration might be the result of a imminent page miss in a single of these configurations, which will force the other to be removed from the grid as well.

8.5 Configuration compatibility

The implemented architecture supports backwards binary compatibility. The procedure used to achieve this is padding older smaller configuration with zeroes, and requiring that the newer versions of the device perform as a legacy device, when affected by this zero padding.

The implementation approach to this might not be the most effective, as the configuration being loaded is zero padded before it is loaded into a cell. This reduces the amount of decoding hardware each cell needs to use the configuration. However, it increases the load on the reconfiguration bus, which can increase the period required for reconfiguration.

To reduce the amount of zero padding newer version is allowed to make use of reserved bit combinations in the existing configuration words. Such bit combinations in the implemented configuration format is likely to be tied directly to a specific unit in a cell. On example of this would be the output MUXes in the cells. Each of these have 3 bits of configuration, however, only 5 of these combinations are in use. Using these bit combinations to something other than the MUX, would require additional decoding logic. Thus increasing the required hardware resources.

One alternative would be to inform each cell of the version the current configuration has, and then disable all units that has been added after that version. This would reduce the amount of configuration that have to be loaded when a older version configuration is utilized. However, it would be harder to utilize the ability to change the decoding of the configuration format in the way the implemented

approach does, as the unit cannot be directly disabled by the version specification.

The solution to this would be to combine the approaches. This would in effect reserve some bit combinations for future use and load a variable size configuration into a cell. However, this might be an exercise in futility. As the worst case still exists were the newest, and largest, configuration must be loadable, within the time limit imposed by the reconfiguration wave.

8.6 Reconfiguration system

In section 7.2, it is shown that the reconfiguration bus speed plummets when the cell grid becomes large. This will increase the overhead of RPU reconfiguration. The cause of this is an inherent flaw in the design of the reconfiguration bus. The number of MUXes inserted into to the configuration bus becomes a problem. Hence, the configuration bus should probably be redesigned. One alternative might be to use a dedicated wiring to each cell in a column, and selecting between these in the reconfiguration reader and writer. This would severely reduce the number of MUXes present on the configuration bus, and thereby increase the achievable speed.

Another alternative might be to consider the computational units included in each cell. These are not in use during reconfiguration. If these could be use to perform some form of rudimentary compression of the current configuration, and decompression of the new configuration, some of the load on the reconfiguration bus and memory system could be reduced.

However, as with configuration compatibility the reconfiguration bus must still support the worst case scenario. The potential gain with this approach would therefore be limited to reducing the load on the memory bus. Unfortunately this is likely to complicate the reconfiguration system, as it then must have the ability to load cell configurations of variable length. As such, this might reduce the number of cells that is implementable in a single chip, which would reduce performance.

In section 7.2 the overhead of the reconfiguration system is indicated as the grid size grows, in both the version 1 and version 2 architecture. With small grid sizes the overhead has a larger impact on the required hardware resources.

The reason for this is the growth rate of the reconfiguration system compared to the cell grid growth. When a single reconfiguration reader and reconfiguration writer is added to the system an entire column of cells are added. Given that the grid is square, a cell will also be added to the existing columns. As such, the area consumed by the cell will quickly begin to dominate the amount of hardware required.

It could argued that the overhead of the reconfiguration system therefore is irrelevant, and the focus of optimizations should be the cells. However, the reconfiguration system does not contribute directly to computational power. Therefore it should be reduced so that more cells can be added to the device. It is noteworthy that any cell optimization is likely to have a larger effect, as it can be applied to all the cells in the grid. Given that a large grid is used an area optimization can make room for an entire cell column if applied to all the existing cells.

These two optimization goals conflict when manpower is limited. Which of the systems that should receive most optimization efforts is hard to tell. Based upon a given grid size, it could be determined by the relative size difference between the cell grid and the supporting reconfiguration system. However, performing optimizations on one of them might be easier.

In the current implemented reconfiguration system, some functionality can be removed, which would decrease the size of the system. Both the reconfiguration reader and reconfiguration writer has its own internal method of detecting when a cell column has been reconfigured. This is done to provide the ability to perform to successive reconfigurations. Under these conditions the terminate detection mechanism is used to determine when a new base address should be loaded into an internal register used to iterate over a memory area.

Given that this functionality is not needed, the reconfiguration reader and reconfiguration writer can be informed when the entire cell column has been reconfigured by the reconfiguration master. This can simply be signaled by the same signal that initiates these units in the first place. It is noteworthy that the hardware structure used to determine when an entire cell column has been reconfigured in the reconfiguration reader, is also used to control which cell is being reconfigured. This structure can also be optimized away, by letting a cell that is being reconfigured signal the next cell in the column, that it should begin reconfiguring soon.

8.7 Conclusion

This thesis has elaborated upon the hardware design of a reconfigurable computing device. This reconfigurable computing device utilizes a specialized reconfiguration system and technique that allows for a high degree of overlapping between two task. This reduces the overhead of performing multitasking on a reconfigurable computing device.

This hardware design has been implemented and tested with extensive simulation. This has proven that this approach to multitasking in a reconfigurable computing environment is possible.

The hardware has also been implemented in two versions. The newest of these hardware version have the ability to run configurations belonging to the older version. This has demonstrated that with sufficient care reconfigurable computing device that support binary compatibility can be created. In addition approaches that allow a smaller configuration to run on a larger device has been elaborated upon and basic support for these have been implemented. This further increases the ability to reuse configurations across device versions and sizes.

The second hardware version supports a form of rudimentary branching, that fits into the spatial computational model of reconfigurable computing devices. This allows some control flow to exist within the reconfigurable hardware. As such, larger amounts of a program should be implementable in the reconfigurable hardware.

Evaluation of the resultant designs have pointed to some key areas that should be investigated further and improved. Most notable of these is the bus between

the reconfiguration system and the computational grid. High efficiency of the bus is paramount to achieving low overhead reconfiguration.

It has been shown that cell design implemented here might be area efficient, however, it has a negative impact on the throughput of the device. However, this throughput is still substantial compared to the interconnection speed, when the number of cells in the device is large.

To aid in testing and use of hardware an assembler and assembly language has been developed that support the inherent parallelism of the resultant device core.

8.8 Future work

The design of the implemented reconfigurable computing core has room for improvements. One of the largest weaknesses that have been identified is the reconfiguration bus design. This should be reevaluted. It should be closely examined of this problem is caused by the way MUXes is implemented in FPGA and whether this problem would be reduced if the chip was implemented as an ASIC. The design flaw in the reconfiguration bus would still exist, however, it might not be that prominent.

To do proper testing in FPGA a test memory system should be developed. The high-speed memory system previously outlined might be to complex for basic FPGA testing, and a simpler specialized system for testing might an alternative. For such testing debug support should also be integrated into the cell grid. This could be a simple scan chain to retrieve the internal state of each cell. This would aide in problem detection.

Such a debug system would also aid in testing if a future attempt at ASIC implementation of the outlined device. Such a implementation should also include a proper high-speed memory system suitable for integration into some form of test computing system, that includes general purpose processors.

In such a computing system operating systems could be extended to perform scheduling of the RPU. This would show if the existing base of operating system task schedulers can be modified to suit the new reconfigurable hardware unit.

However, to make use of such hardware the existing assembler software should be extended with a compiler. This compiler would operate on a high-level language, and would perform automatic mapping of operations onto the device grid.

The communication patterns allowed within the outlined device grid is very limited. This might complicate this mapping of operations. As such, the possibilities for bidirectional communication should be investigated. This would complicate the reconfiguration waves, as all data flowing against the wave must be stored in memory. To make this even harder the data intercepted by the reconfiguration wave, must be moved slightly within the grid before it can be restored. The reason for this is that it must be made available to the data transfer target cell, when this is restored to its previous configuration.

Bibliography

[BLM+07] Koen Bosschere, Wayne Luk, Xavier Martorell, Nacho Navarro, Mike O'Boyle, Dionisios Pnevmatikatos, Alex Ramirez, Pascal Sainrat, André Seznec, Per Stenström, and Olivier Temam. High-performance embedded architecture and compilation roadmap. pages 5–29, 2007.

[CEL+03] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. The reconfigurable streaming vector processor (rsvptm). In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 141, Washington, DC, USA, 2003. IEEE Computer Society.

[CH02] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[EAGEG09] Esam El-Araby, Ivan Gonzalez, and Tarek El-Ghazawi. Exploiting partial runtime reconfiguration for high-performance reconfigurable computing. *ACM Trans. Reconfigurable Technol. Syst.*, 1(4):1–23, 2009.

[EBTB63] G. Estrin, B. Bussell, R. Turn, and J. Bibb. Parallel processing in a restructurable computer system. *Electronic Computers, IEEE Transactions on*, EC-12(6):747 –755, dec. 1963.

[Est60] Gerald Estrin. Organization of computer systems: the fixed plus variable structure computer. In *IRE-AIEE-ACM '60 (Western): Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference*, pages 33–40, New York, NY, USA, 1960. ACM.

[Fri01] E.G. Friedman. Clock distribution networks in synchronous digital integrated circuits. *Proceedings of the IEEE*, 89(5):665–692, May 2001.

[GSB+00] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Reed Taylor. Piperench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, 2000.

[HD07] Scott Hauck and Andre DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[JTY+99] Jack S. N. Jean, Karen Tomko, Vikram Yavagal, Jignesh Shah, and Robert Cook. Dynamic reconfiguration to support concurrent applications. *IEEE Trans. Comput.*, 48(6):591–602, 1999.

[LMSS00] L. Levinson, R. Männer, M. Sessler, and H. Simmler. Preemptive multitasking on fpgas. In *FCCM '00: Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, page 301, Washington, DC, USA, 2000. IEEE Computer Society.

[MB07] Uwe Meyer-Baese. *Digital Signal Processing with Field Programmable Gate Arrays*. Springer Publishing Company, Incorporated, 2007.

[MIP09] MIPS Technologies, Mountain View, CA, USA. *MIPS32 Architecture for Programmers Volume II: The MIPS32 Instruction Set*, 2009.

[Oft09] Kjetil Wathne Oftedal. Multitasking on a reconfigurable computing system. Technical report, IDI, 2009.

[Pat85] David A. Patterson. Reduced instruction set computers. *Commun. ACM*, 28(1):8–21, 1985.

[PD80] David A. Patterson and David R. Ditzel. The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News*, 8(6):25–33, 1980.

[PH07] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[PS81] David A. Patterson and Carlo H. Sequin. Risc i: A reduced instruction set vlsi computer. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 443–457, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[QSN06] Yang Qu, Juha-Pekka Soininen, and Jari Nurmi. A parallel configuration model for reducing the run-time reconfiguration overhead. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 965–969, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[RCG+08] Javier Resano, Juan Antonio Clemente, Carlos Gonzalez, Daniel Mozos, and Francky Catthoor. Efficiently scheduling runtime reconfigurations. *ACM Trans. Des. Autom. Electron. Syst.*, 13(4):1–12, 2008.

[RMC05] Javier Resano, Daniel Mozos, and Francky Catthoor. A hybrid prefetch scheduling heuristic to minimize at run-time the reconfiguration overhead of dynamically reconfigurable hardware. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 106–111, Washington, DC, USA, 2005. IEEE Computer Society.

[SCMG00] Herman H. Schmit, Srihari Cadambi, Matthew Moe, and Seth C. Goldstein. Pipeline reconfigurable fpgas. *J. VLSI Signal Process. Syst.*, 24(2/3):129–146, 2000.

[SPA94] SPARC International, San Jose, CA, USA. *The SPARC Architecture Manual: Version 9*, 1994.

[Sun82] Sun-Yuan Kung, and Arun, K. S. and Gal-Ezer, R. J. and Bhaskar Rao, D. V. Wavefront Array Processor: Language, Architecture, and Applications. *IEEE Trans. Comput.*, 31(11):1054–1066, 1982.

[TCJW97] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed fpga. In *FCCM '97: Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, page 22, Washington, DC, USA, 1997. IEEE Computer Society.

[VS07] Stamatis Vassiliadis and Dimitrios Soudris. *Fine- and Coarse-Grain Reconfigurable Computing*. Springer Publishing Company, Incorporated, 2007.

[Xil09] Xilinx. *Virtex-6 FPGA Configuration, User Guide, v1.1*, 2009.

Appendix A

Multiplication and division

A basic design for a multiplier and divider was found in *Computer organization and design* [PH07]. The multiplier and divider designs presented are very similar, and consists primarily of a small ALU and a shift register. This allows for hardware reuse which reduces the amount of area needed to implement these operations. It is noteworthy that these designs, although area efficient, requires a large amount of clock cycles to perform a single operation.

A.1 Multiplication algorithm

The multiplication algorithm employed in this design is a basic *shift and add* algorithm, which is the computer science equivalent of long multiplication. A flow chart for a 32-bit multiply operation using this algorithm is shown in figure A.1. The product and the multiplicand are both stored in 64-bit registers, while the multiplier resides in a 32-bit register.

At the start of algorithm the Least Significant Bit (LSB) of the multiplier is tested. If the bit is set the multiplicand will be added to the product register. This corresponds to the multiplication using a single digit in the familiar long multiplication approach. As this is multiplication algorithm uses binary numbers, there is no need for the multiplication of a single digit of the multiplier with the multiplicand. A simple add based upon the digit is sufficient.

Then the multiplicand is shifted left. The multiplicand register is 64-bits wide, so no part of the multiplicand is lost. It only changes the digits in the product that will be affected by the multiplicand add. In addition the multiplier register is shifted right. As the multiplier is already stored near the end of this register, the shift operation will cause the current LSB to be lost.

Thereafter the algorithm checks if the termination criteria has been met, which is 32 repetitions of the algorithm. The termination criteria is more formally equal to the data width of the input operands. Has the termination criteria not been met the algorithm returns to the start, and the operations are performed again with the newly shifted operands.

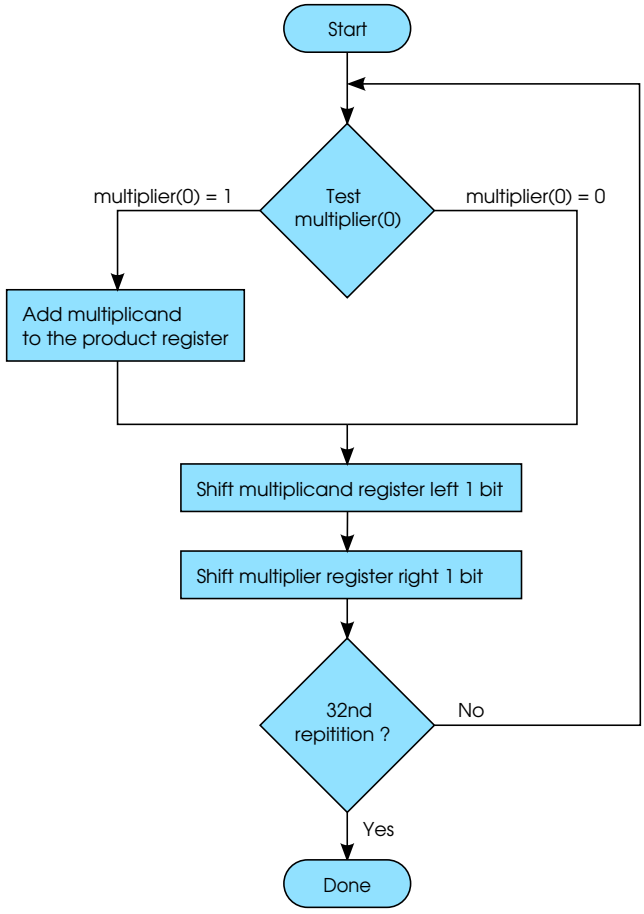

Figure A.1: Multiply operation flowchart (Adapted from [PH07])

Several optimizations can be applied to this algorithm as suggested in *Computer organization and design* [PH07]. The most important one allows for the reduction of the number of registers needed in addition to reducing the width of the adder. The change in the algorithm that allow this is shifting the product, in stead of the multiplicand. Also the shift direction is reversed. This is allowed because the of the number of times each digit in the product can be added to the multiplicand. The LSB can only be affected by the first addition operation, the second least significant bit can only be affected by the two first addition operations, and so forth in the presented algorithm.

However, the product must be stored at the top of a 64-bit register so that the digits being shifted will not be lost. Only the top 32 bits of this register will be used in the addition. These top 32 bits are used as one of the inputs to a 32-bit adder, which is a reduction from the 64-bit adder required earlier.

The shift direction of the product and the multiplier are now the same. And the bottom 32 bits of the 64-bit product register is unused when the algorithm starts, as no data of significance has been shifted into this part. The separate multiplier register can therefore be removed and the multiplier can be stored in the lower part of the product register.

When the algorithm has completed the combined product and multiplier register will only contain the product, as the entire multiplier has been shifted out. As such, a 64-bit result of the multiplication operation can be read from it.

A.2 Division algorithm

The division algorithm corresponding to the multiplication algorithm previously presented is illustrated in figure A.2. This algorithm is the binary variant of the basic manual division algorithm, were division is reduced to basic subtractions. At the start of the algorithm the remainder equals the dividend. Both the divisor and the remainder are stored in 64-bit registers, while the quotient is stored in a 32-bit register. The divisor is not stored at the bottom of the 64-bit register, but at the top.

The divisor is subtracted from the remainder. The algorithm presented here and the one from *Computer organization and design* [PH07] differ a bit at this stage. Here the result is compared directly, in [PH07] the result is stored to the remainder register before the comparison. The latter approach require an additional stage where the original value is restored if the result was negative.

If the result of this operation is negative a zero is shifted into the quotient register. Is the result positive a one is shifted into the quotient register, and the result of the operation is stored to the remainder register. The bits in the quotient register symbolizes the subtraction operations that lead to a non-negative result. This is similar to the basic manual division algorithm. However, in that algorithm the number of times the divisor could be subtracted from the remainder must be calculated as well. This is not needed when calculating using binary numbers.

Next, the divisor register is shifted to the right. This reduces the value of the divisor relative to the remainder. At the end of the algorithm the termination

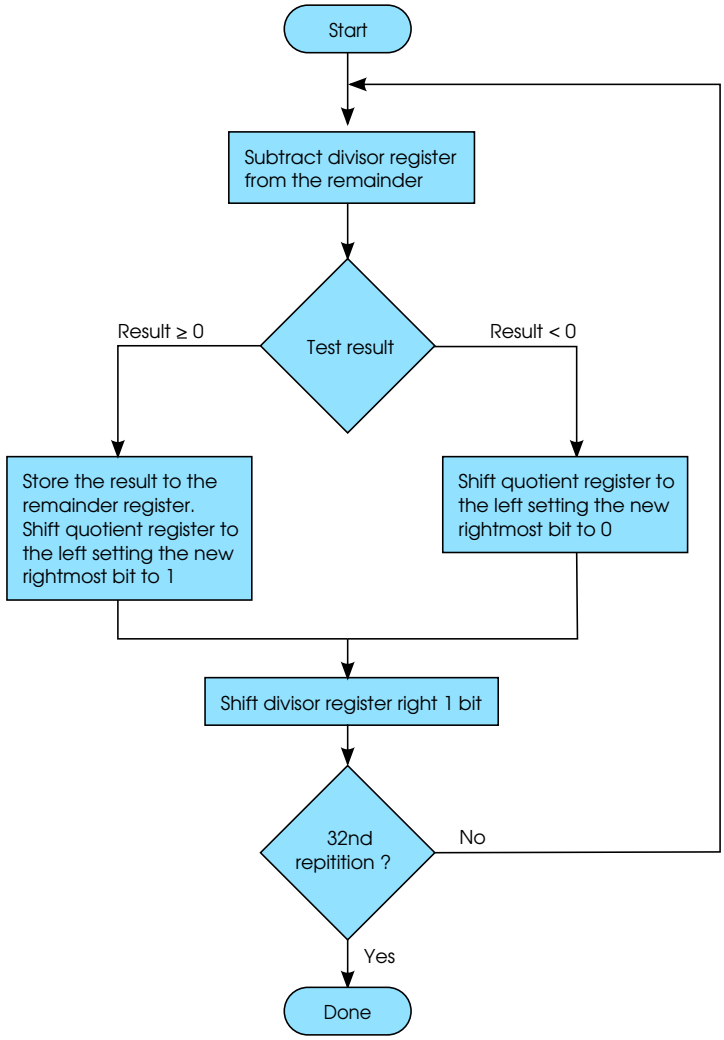

Figure A.2: Divide operation flowchart (Adapted from [PH07])

criteria is checked, which is 32 repetitions of the algorithm. This termination criteria does differ from the one presented in *Computer organization and design*. As the termination criteria there is 33 repetitions. However, in this implementation the MSB is the sign bit, which is always zero, and the operands will actually just be 31 bits wide. The reason for this will be explained in the implementation in section A.3. Actually the same argument can be used with the multiplication algorithm. However, it was chosen to keep it at 32 repetitions there so that the termination criteria for the multiplier and divider would be identical.

Computer organization and design also suggests optimizations for this algorithm. The divisor can be kept constant, and at 32 bits. In stead the remainder is shifted to the left. Which increases the value of the remainder in relation to the divisor. The remainder is still stored in a 64-bit shift register. Of these 64 bits only the top 32 bits is used in the subtraction operation. Hence, the width of the subtractor can be reduced to 32 bits.

The last optimization is eliminating the separate quotient register. At the beginning of the algorithm the quotient register contains no value of significance. During the algorithm the number of significant bits in this register increases as the same rate as the remainder is shifted to the left. This allows the use of the lower part of the remainder shift register as storage for the quotient.

A.3 Hardware implementation

This implementation of a combined multiplier and divider unit uses the previously presented algorithms and optimizations. These algorithms have been augmented with some handling of conditions not taken into account in the previous sections. The resultant hardware implementation will support signed, and handle division by zero gracefully. The structure will structure will be presented in three stages, each building upon the previous presented structure.

A.3.1 Unsigned operation

In figure A.3 the combined multiplier and division structure is shown. The lighter blue boxes are the input registers to the unit. The darker blue boxes are the internal components. This structure is very similar to the optimized hardware structure presented by in *Computer organization and design* [PH07]. The structure is used slightly different for multiplication and division. Which operation is performed is decided by the *Op* input.

When both valid inputs are asserted the computation will begin. At this time the values on the data and operation input register will no longer be updated, and therefore will be frozen to the current input values.

At the start of multiplication the *Data a* value is stored in the lower part of the bidirectional shift register, and the top value is set to zero. The bidirectional shift register will be the multiplier register. And as the multiplier is shifted out, it will gradually become the product register.

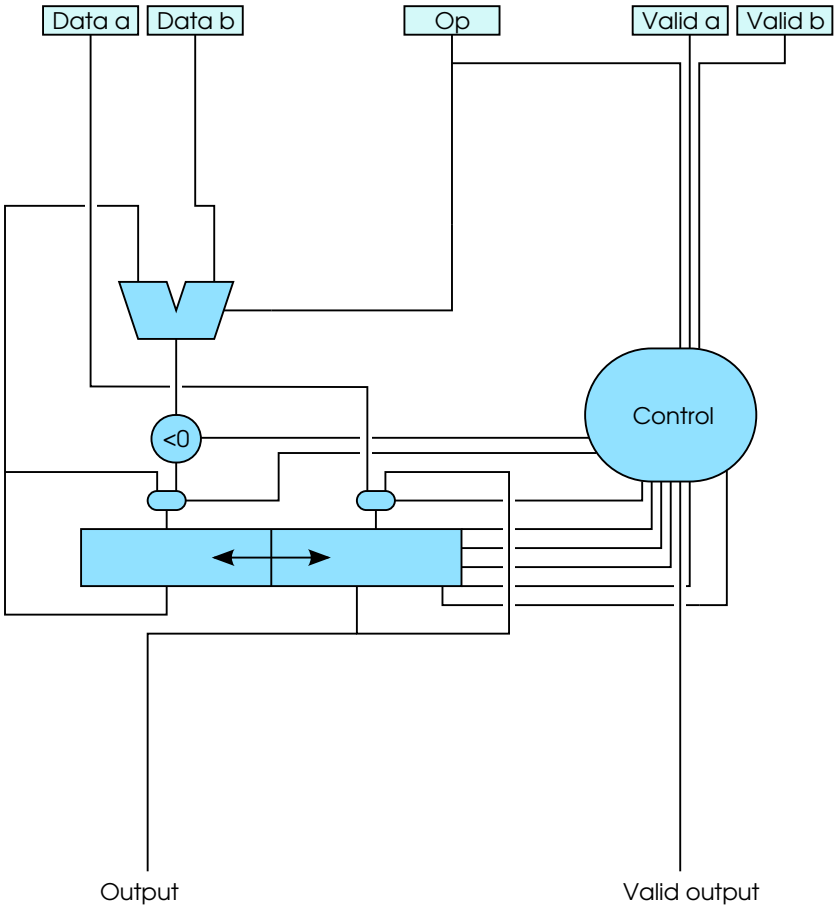

Figure A.3: Unsigned combined multiplier and divider

This bidirectional shift register will be set to shift right, by the *Control* unit. The *Control* unit will decide based upon the value shifted out of this register, whether the value in the top part of the register should be fed back into the register or if the output of the ALU should be used. The ALU can perform both addition and subtraction. During unsigned multiplication only addition will be used.

The *Control* unit has an internal counter, which it uses to decide the repetition number of the algorithm. When the termination criteria has been met, the product is present in the entire shift register. However, as the architecture has a fixed operand width only the bottom part of the product is put on the *Output* port. At this point the *Valid output* is also asserted.

Division uses the structure differently. *Data a* is stored in the lower part of the bidirectional shift register, as with multiplication. During division this is the remainder register. As the remainder is shifted to the left, it will gradually become the quotient register.

During division the divisor stored in *Data b* will be subtracted from the output of the top part of the remainder register, by the ALU. The output of the ALU is monitored by the *Control* unit for negative values. If the result is non-negative a one will be shifted into the combined remainder and quotient register. When the result of the subtraction is negative the output of the remainder register is fed directly back into the bidirectional shift register, and a zero will be shifted into the quotient register.

The *Control* unit uses the same internal counter as with multiplication to terminate operations. When operations are terminated the lower part of the bidirectional register, which contains the quotient, is put on the *Output* port. The *Valid output* is asserted, as at the end of multiplication.

A.3.2 Signed operation

The components used for unsigned operation, shown in blue in figure A.4 are extended with the components for signed operation, shown in yellow. In *Computer organization and design* [PH07] it is suggested that the easiest way to implement signed operation is by converting the input operands to positive numbers, and remembering their signs.

However, calculating the absolute value of two's complement numbers requires an adder and an inverter. This would increase the size of the combined multiplication and division unit. It turns out that the numbers can be converted by reusing the ALU and by manipulation the ALU operation performed during algorithm execution.

The *Data a* input, now being passed through the ALU on the way to the bottom half of the bidirectional shift register, is converted to a positive value. This is done by checking if *Data a* is negative, and feeding the result of that check into the operation port of the ALU. The ALU will perform a subtraction when this check is true. Zero is put on the other input to the ALU. This results in the following.

A	: The input value
a	: The absolute value of A

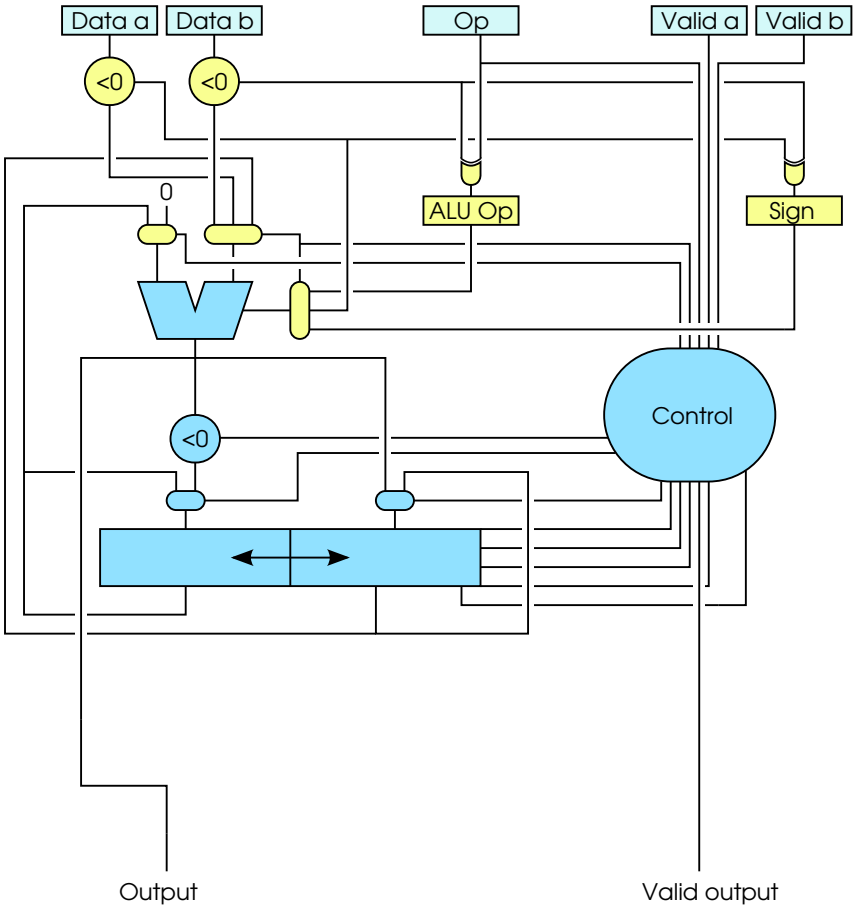

Figure A.4: Signed combined multiplier and divider

When A is positive:

$$\begin{aligned} 0 + A \quad & = 0 + +(a) \\ & = a \end{aligned}$$

When A is negative:

$$\begin{aligned} 0 - A \quad & = 0 - -(a) \\ & = 0 + (a) \\ & = a \end{aligned}$$

Hence, the variable ALU operation will result in all negative numbers being converted to their corresponding absolute value. While positive numbers remain unchanged.

Data b might also be negative. The previously presented scheme that reuses the ALU could be employed. However, this would add another clock cycle to the computation time required by the combined multiplication and division unit. as the ALU is busy converting the *Data a* input. Converting *Data b* to a positive value before computation starts is actually unnecessary, as this can be performed while the algorithms runs, by using a modified version of the previously presented scheme.

In the unsigned version the *Op* input would decide whether the ALU should perform addition or subtraction directly. This is possible as multiplication will always use addition, and division will always use subtraction. Given this it is possible to convert the value of *Data b* to be positive on the fly by inverting the operation the ALU performs when *Data b* is negative. The following is a generalization of the *Data a* conversion:

X	: The first alu input
B	: The second alu input
b	: The absolute value of the second alu input

Addition - When B is positive:

$$\begin{aligned} X + B \quad & \rightarrow X + +(b) \\ & = X + b \end{aligned}$$

Addition - When B is negative:

$$\begin{aligned} X + B \quad & \rightarrow X - -(b) \\ & = X + (b) \\ & = X + b \end{aligned}$$

Subtraction - When B is positive:

$$\begin{aligned} X - B \quad & \rightarrow X - +(b) \\ & = X - b \end{aligned}$$

Subtraction - When B is negative:

$$\begin{aligned} X - B \quad & \rightarrow X + -(b) \\ & = X - b \end{aligned}$$

In figure A.4 the operation inverting is performed by taking the exclusive or of the *Op* input and the sign bit of *Data b*. Exclusive or is a somewhat expensive gate to use. Though compared to an adder and an inverter, it is very cheap. Especially as a half adder element consist of a exclusive or gate and an and gate.

Converting the operands to positive numbers is not enough, as the result then would always be positive. The result of the multiplication or division must be converted to negative number when sign of the two operands differ. Therefore a second exclusive or gate is added in figure A.4.

The input to this gate is the sign bit of both operands. The result being asserted when the signs differ. This is stored in the *Sign* register. When the division or multiplication algorithm has terminated, the result is stored in the lower part of the bidirectional shift register. This result is put on the second input of the ALU. Zero is present on the first input. The contents of the *Sign* register is then used to perform the same conversion as with *Data a*. Although as the output of the bidirectional shift register is always positive, the reverse would happen. Which is that the positive value would be left positive or converted to a negative number dependent on the *Sign* register.

A.3.3 Division by zero

In figure A.5 the complete combined multiplier and divider structure is shown. The previous structure has been augmented with elements that should handle division by zero, shown in green.

Division by zero on CPUs often cause some sort of exception or error being signaled to the running program. The presented RC device has no exception handling system. Hence, this problem should be handled by other means. The solution chosen here, is that any division by zero attempt should yield and appropriate large value. Which corresponds to taking the limit of the function:

$$\begin{aligned} f(a,b) \quad & = \frac{a}{b} \\ lim_{b \to 0^+} f(a,b) \quad & = lim_{b \to 0^+} \frac{a}{b} \end{aligned}$$

Given that a is non-zero and positive:

$$= \infty$$

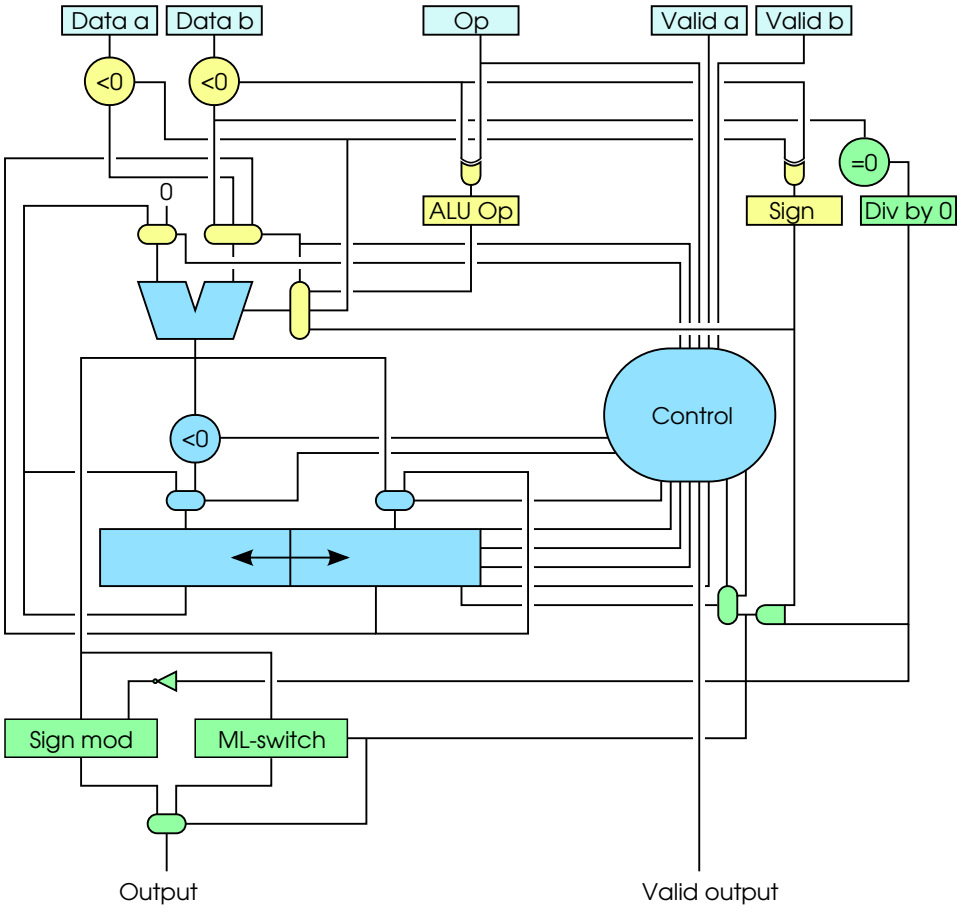

Figure A.5: Complete combined multiplier and divider

However, if *Data a* is negative it would be appropriate that the result should reflect his an yield the largest negative number that is representable. This could be achieved by a simple test performed by the control unit, and the driving the output to the predetermined values.

In the implementation a more area efficient, less power efficient, approach was used. This was done by considering the division algorithm illustrated in figure A.2. When the divisor is zero, in this algorithm, the result of the subtraction operation would always be positive. Hence, the quotient register would be filled with one bits, when the algorithm terminates. The quotient register is then converted based upon the sign bits of the input values. The result of these operations with an 8 bit data width:

$$\begin{aligned} Q &= \text{Quotient when the division algorithm terminates} \\ C &= \text{Output from the conversion} \\ R &= \text{The result expected} \\ Q &= 11111111 \end{aligned}$$

If sign bits cause no conversion

$$\begin{aligned} C &= Q \\ &= 11111111 \\ R &= 01111111 \end{aligned}$$

If sign bits cause conversion

$$\begin{aligned} C &= \text{not}(Q) + 00000001 \\ &= 00000000 + 00000001 \\ &= 00000001 \\ R &= 10000000 \end{aligned}$$

As shown the number of bits that must be changed to obtain the correct result is low. The core of the result can be left unchanged and only the LSB and MSB bits require manipulation. This manipulation is performed by the *Sign mod* and *ML-switch* blocks in figure A.5.

Sign mod should simply negate the MSB of the result when division by zero is performed and the result should be positive. To reduce the number of paths that would lead to the MUX connected to the *Output* this unit is connected by default, and the negation operation is performed by conditional by anding the inverted content of the *Div by zero* to the MSB bit.

ML-switch simply exchanges the LSB and MSB bits. This must be performed when the sign bits of the input value should yield a negative result and division by zero is performed. Hence it should be activated if content of both the *Sign* and *Div by 0* registers are asserted.

There is however a corner case related to division by zero and negative values. When the largest negative value the structure can handle as input is divided by

zero, the encoding of two's complement numbers causes some trouble. This is shown in the following conversion example:

$$\begin{aligned} I &= \text{The input value} \\ R &= \text{The absolute value} \\ I &= 10000000 \end{aligned}$$

Try to calculate the absolute value:

$$\begin{aligned} R &= not(I) + 00000001 \\ &= 01111111 + 00000001 \\ &= 10000000 \end{aligned}$$

This would by it self cause no trouble, if it were not for the fact that this value will be fed to the ALU at the very last step of the division algorithm. Zero will be subtracted in the ALU, which will not change the value. When the value then leaves the ALU it will be tested for signedness. When since the sign bit is set, the *Control* unit will detect a negative value, and shift a zero into the LSB of the quotient.

The assumption done previously with regards to the result of the algorithm when division by zero is performed will not hold in all cases. To mitigate this a MUX was added to the shifter input, under normal circumstances this input will be driven by the *Control* unit. However, during division by zero the input is the result of anding the contents of the *Sign* register and the *Div by 0* register. The effect of this is that ones will always be shifted into the quotient register, when dividing by zero and the other operand is negative.

There is one additional corner case here. Which is $\frac{0}{0}$. This corner case has been defined to cause the same operation as any other positive number. Hence no structural changes are needed. However, the RPU programmer should be aware of this behavior.

Appendix B

Register and configuration layout

The descriptions in this section assumes that the implemented device uses 16-bit datawords and 8-bit addresses.

B.1 System interface

B.1.1 Status register

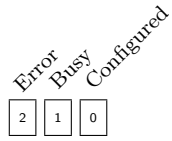

Figure B.1: Status register layout

Error	This bit indicates that an error has occurred during reconfiguration. This can be the result of loading a configuration of higher version than the device, or that the configuration is unscalable.
Busy	Indicates whether the system is ready for new reconfiguration commands, or if the reconfiguration system is currently busy.
Configured	This bit is set if there is a configuration currently running of the device.

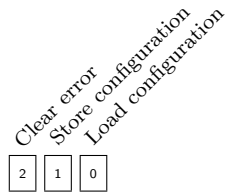

Figure B.2: Control register layout

B.1.2 Control register

Clear error Writing a logic one to this portion of the register clear the error flag currently indicated by the status register, and allow normal reconfiguration operation to resume.

Store configuration Writing a logic one to this register will instruct the reconfiguration system to store the current configuration to memory.

Load configuration Writing a logic on to this register will instruct the reconfiguration system to load a configuration into the device from memory.

B.1.3 Auxiliary control register

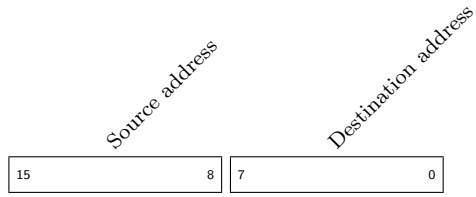

Figure B.3: Auxiliary control register layout

Source address Specifies the start address of the configuration that is to be loaded into device.

Destination address Specifies the storage address of the current running configuration.

B.2 Configuration

B.2.1 Header

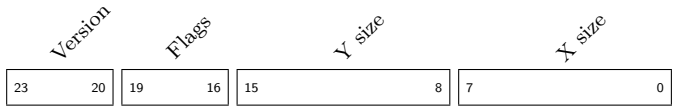

Figure B.4: Configuration header layout

Version Contains the version of the rest of the configuration stream. The content of this register shall be one less than the human readable version number.

Flags Contains the flags of the configuration. See figure B.5 for usage.

Y size The vertical size of the configuration.

X size The horizontal size of the configuration.

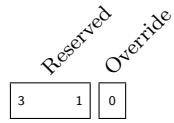

Figure B.5: Flags field

Reserved Reserved for future use.

Flags Setting this bit high, will disable any idle wave generation during scaling. This can be used when there are no external feedback loops in the configuration.

B.2.2 Configuration version 1

ALU input a Specifies the first operand to the ALU. See table B.1 for encoding.

ALU input b Specifies the second operand to the ALU. See table B.1 for encoding.

ALU op Specifies the operation the ALU shall perform. See table B.2 for encoding.

MD input a Specifies the first operand to the combined multiplication and division unit. See table B.3 for encoding.

MD input b Specifies the second operand to the combined multiplication and division unit. See table B.3 for encoding.

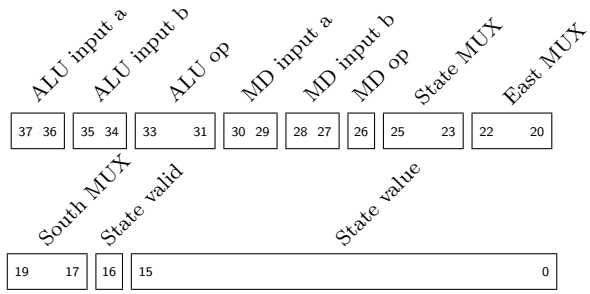

Figure B.6: Configuration version 1 layout

MD op Specifies the operation the combined multiplication and division unit should perform. If set low multiplication will be performed. If set high division will be performed.

State MUX Specifies the signal that should be connected to the state register input. See table B.4 for encoding.

South MUX Specifies the signal that should be connected to the south cell output. See table B.4 for encoding.

East MUX Specifies the signal that should be connected to the east cell output. See table B.4 for encoding.

State valid Specifies if the value in the state register is valid.

State value Contains the value the state register should contain when configuration has completed.

Value	Input
00	West
01	North
10	State
11	MD output

Table B.1: ALU Multiplexer encoding

Value	Operation
000	Add
001	Subtract
010	Shift logic right
011	Shift logic left
100	Logic OR
101	Logic NOR
110	Logic AND
111	Logic XOR

Table B.2: ALU opcode encoding

Value	Input
00	West
01	North
10	State
11	ALU output

Table B.3: MD Multiplexer encoding

Value	Input
000	West
001	North
010	State
011	ALU output
100	MD output
101-111	Reserved for future use

Table B.4: State and output Multiplexer encoding

B.2.3 Configuration version 2

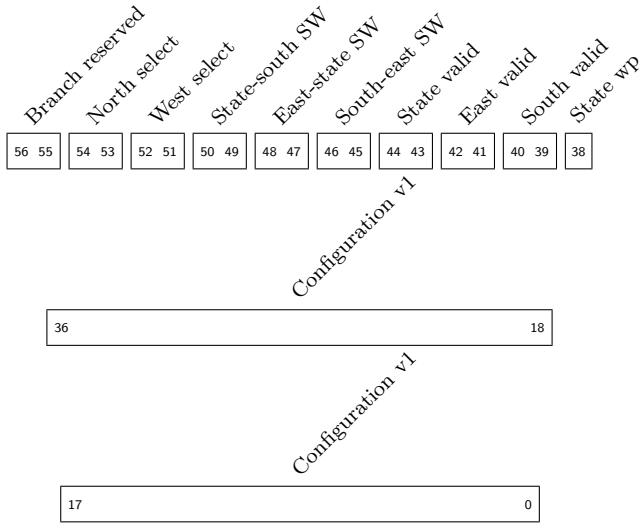

Figure B.7: Configuration version 2 layout

Branch reserved These bits in the configuration stream is reserved for branch based memory load. However, this has not been implemented, as the memory system is currently a black box.

North select Specifies the policy of witch input should be used as the north input in computations. See table B.5 for encoding.

West select Specifies the policy of witch input should be used as the west input in computations. See table B.5 for encoding.

State-south SW Specifies the conditions that will switch the input to the state register and the south output. See table B.6 for encoding.

East-state SW Specifies the conditions that will switch the east output and the input to the state register. See table B.6 for encoding.

South-east SW Specifies the conditions that will switch the south and east output. See table B.6 for encoding.

State valid Specifies the valid policy of the input to the state register. See table B.7 for encoding.

East valid Specifies the valid policy of the east output. See table

B.7 for encoding.

South valid Specifies the valid policy of the east output. See table B.7 for encoding.

State wp State register write policy. Decides when the content of the content of the state register should be updated. See table B.8 for encoding.

Configuration v1 This part of the configuration is identical to a version 1 configuration bit stream.

Value	Input
00	Default
01	Use other input when the default is not valid
10	Use other input when other input is valid
11	Reserved

Table B.5: Input select policy encoding

Value	Input
00	Default
01	Switch on ALU zero
10	Switch on ALU negative
11	Reserved

Table B.6: Switch policy encoding

Value	Input
00	Default
01	Conditional on ALU zero
10	Conditional on ALU negative
11	Never valid

Table B.7: Valid policy encoding

Value	Input
0	Write always
1	Write on valid

Table B.8: State write policy encoding

Appendix C

Assembler Grammar

< description >	→	< cell list >
< cell list >	→	< cell >
	|	< cell list >
< cell >	→	**cell** < dim list > {< statement list >}
< statement list >	→	< statement >
	|	< statement list >
< statement >	→	< initialization >
	|	< assignment >
	|	< branch >
	|	< valid >
	|	< write >
< initialization >	→	**init**{**state** =< number >;}
< assignment >	→	< target >=< source >< op >< source >;
	|	< target >=< source >;
< branch >	→	**if**(< branch expression >){< branch statement list >}
< branch expression >	→	< flags > (< flag source >)
	|	**not valid**(< inputs >)
	|	**valid**(< inputs >)
< branch statement list >	→	< branch statement >
	|	< branch statement list >< branch statement list >
< branch statement >	→	**switch**(< outputs >, < outputs >);
	|	< inputs >=< inputs >;
< valid >	→	**valid**(outputs) =< flags > (flags source);
	|	**valid**(outputs) = **never**;

< write > → **write**(**state**) =< write policy >;
< write policy > → **valid**
| **always**
< target > → < outputs >
| **aluout**
| **mulout**
< op > → **and**
| **or**
| **xor**
| **nor**
| **add**
| **sub**
| **div**
| **mul**
| **sll**
| **slr**
< source > → < inputs >
| **aluout**
| **mulout**
< inputs > → **north**
| **west**
< outputs > → **state**
| **east**
| **south**
< flags > → **negative**
| **zero**
< flag source > → **aluout**
< dim list > → **dim**
| **dim list**
< dim > → [< integer >]
< number > → **-** < integer >
| < integer >

Appendix D

Test overview

D.1 Version independent components

Test	Result
Automatic	Passed

Table D.1: Addersubtracter

Test	Result
Basic IO	Passed
Addition	Passed
Addition underflow	Passed
Addition overflow	Passed
Addition operand order	Passed
Addition signed	Passed
Subtraction	Passed
Subtraction underflow	Passed
Subtraction overflow	Passed
Subtraction operand order	Passed
Subtraction signed	Passed
Shift logic right	Passed
Shift logic right long	Passed
Shift logic left	Passed
Shift logic left long	Passed
Logic OR	Passed
Logic OR operand order	Passed
Logic NOR	Passed
Logic NOR operand order	Passed
Logic AND	Passed
Logic AND operand order	Passed
Logic XOR	Passed
Logic XOR operand order	Passed
Instruction mix	Passed
Zero flag	Passed
Negative flag	Passed
Valid flag	Passed
Automatic	Passed

Table D.2: ALU

Test	Result
Basic IO	Passed
Zero multiplicand	Passed
Zero multiplier	Passed
Basic multiplication	Passed
Multiplication operand order	Passed
Signed multiplication	Passed
Multiplication large operand	Passed
Multiplication overflow	Passed
Zero dividend	Passed
Basic division	Passed
Division operand order	Passed
Division with -MAX_INT (abs)	Passed
Division by zero	Passed
Instruction mix	Passed
Automatic	Passed

Table D.3: Multiplydivide

Test	Result
Zero	Passed
Contamination	Passed
Shorts	Passed
Automatic	Passed

Table D.4: MUX

Test	Result
Basic IO	Passed
Basic shifting	Passed
Shift out	Passed
Complex shifting	Passed
Long shift	Passed
Marginally long shift	Passed
Marginally sub-long shift	Passed
Shift right border	Passed
Shift left border	Passed
Automatic	Passed

Table D.5: Shifter

Test	Result
Basic IO	Passed
Write flag	Passed
Shift left	Passed
Shift right	Passed
Shift left priority	Passed
Carry in	Passed
Carry out	Passed
Write flag does not affect carry	Passed
Automatic	Passed

Table D.6: Shift register

D.2 Version 1 components

Test	Result
Configuration system connected	Passed
Reconfiguration	Passed
Basic pass-through configuration	Passed
Check configuration bus connection during computation	Passed
Back-to-back reconfiguration	Passed
Valid flag propagation	Passed
Port crosstalk	Passed
ALU	Passed
ALU valid	Passed
MD	Passed
MD valid	Passed
State register	Passed
Internal data flow	Passed
Internal data flow valid	Passed
Idle	Passed
Automatic	Passed

Table D.7: Cell

Test	Result
No stray signals during idle	Passed
Read initial configuration	Passed
Read configuration with writeback of the previous configuration	Passed
Write configuration without reading a new configuration	Passed
Drain configuration twice, should be indicated as an error	Passed
Errors are presistive until cleared	Passed
Errors should not affect current configuration	Passed
Loading oversized configuration, should be indicated as an error	Passed
Loading configuration that is too new, should be indicated as an error	Passed
Loading unscalabel configuration, should be indicated as an error	Passed
Automatic	Passed

Table D.8: Reconfiguration master

Test	Result
Basic IO	Passed
Disable horizontal pattern	Passed
Enable horizontal pattern	Passed
Disable vertical pattern	Passed
Enable vertical pattern	Passed
Address propagation correct	Passed
Back-to-back reconfiguration	Passed
Automatic	Passed

Table D.9: Reconfiguration reader

Test	Result
Basic IO	Passed
Disable horizontal pattern	Passed
Enable horizontal pattern	Passed
Disable vertical pattern	Passed
Enable vertical pattern	Passed
Address propagation correct	Passed
Back-to-back reconfiguration	Passed
Automatic	Passed

Table D.10: Reconfiguration writer

Test	Result
Parallel IO	Passed
Parallel write signal	Passed
Serial IO	Passed
Combined serial and parallel IO	Passed
Generic override of write prohibit	Passed
Late change on write flag	Failed
Automatic	Passed

Table D.11: Reconfiguration register

Test	Result
Automatic testing based upon simulator	Passed
Automatic testing of scaled configuration	Passed

Table D.12: Reconfigurable processing unit

D.3 Version 2 components

Test	Result
Configuration system connected	Passed
Reconfiguration	Passed
Basic pass-through configuration	Passed
Check configuration bus connection during computation	Passed
Back-to-back reconfiguration	Passed
Valid flag propagation	Passed
Port crosstalk	Passed
ALU	Passed
ALU valid	Passed
MD	Passed
MD valid	Passed
State register	Passed
Internal data flow	Passed
Internal data flow valid	Passed
Idle	Passed
Conditional validation	Passed
Conditional writing	Passed
Input selection priority other	Passed
Input selection priority local	Passed
Output switching	Passed
Late flag based write to state	Passed
Automatic	Passed

Table D.13: Cell

Test	Result
No stray signals during idle	Passed
Read v1 configuration	Passed
Read v1 configuration with writeback of the previous configuration	Passed
Write configuration without reading a new configuration	Passed
Drain configuration twice, should be indicated as an error	Passed
Errors are presistive until cleared	Passed
Errors should not affect current configuration	Passed
Loading oversized configuration, should be indicated as an error	Passed
Loading configuration that is too new, should be indicated as an error	Passed
Loading unscalabel configuration, should be indicated as an error	Passed
Read v2 configuration	Passed
Read v2 configuration with writeback of the previous configuration	Passed
Automatic	Passed

Table D.14: Reconfiguration master

Test	Result
V1 Basic IO	Passed
V1 Disable horizontal pattern	Passed
V1 Enable horizontal pattern	Passed
V1 Disable vertical pattern	Passed
V1 Enable vertical pattern	Passed
V2 Basic IO	Passed
V2 Disable horizontal pattern	Passed
V2 Enable horizontal pattern	Passed
V2 Disable vertical pattern	Passed
V2 Enable vertical pattern	Passed
V2 Dummy	Passed
Back-to-back reconfiguration	Passed
Address propagation correct	Passed
Automatic	Passed

Table D.15: Reconfiguration reader

Test	Result
V1 Basic IO	Passed
V1 Disable horizontal pattern	Passed
V1 Enable horizontal pattern	Passed
V1 Disable vertical pattern	Passed
V1 Enable vertical pattern	Passed
V2 Basic IO	Passed
V2 Disable horizontal pattern	Passed
V2 Enable horizontal pattern	Passed
V2 Disable vertical pattern	Passed
V2 Enable vertical pattern	Passed
Address propagation correct	Passed
Back-to-back reconfiguration	Passed
Automatic	Passed

Table D.16: Reconfiguration writer

Test	Result
Parallel IO	Passed
Parallel write signal	Passed
Serial IO	Passed
Combined serial and parallel IO	Passed
Generic override of write prohibit	Passed
Late change on write flag	Passed
Automatic	Passed

Table D.17: Reconfiguration register

Test	Result
Automatic testing based upon simulator	Passed
Automatic testing with v1 configuraitons	Passed
Automatic testing of scaled configuration	Passed

Table D.18: Reconfigurable processing unit

Appendix E

Code

All the code has been included in digital appendix. This includes:

- Implementation VHDL
- Testbenches
- Assembler source code
- Modified software simulator used for testing
- Various utilities