Performance driven FPGA design with an ASIC perspective

Andreas Ehliar

Linköping, 2009

Performance driven FPGA design with an ASIC perspective
Andreas Ehliar
Dissertations, No 1237

Copyright © 2008-2009 Andreas Ehliar (unless otherwise noted)
ISBN: 978-91-7393-702-3
ISSN: 0345-7524
Printed by LiU-Tryck, Linköping 2009

Front cover Pipeline of an FPGA optimized processor (See Chapter 7)
Back cover: Die photo of a DSP processor optimized for audio decoding (See Chapter 6)

URL for online version: `http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-16732` Errata lists will also be published at this location if necessary.

Abstract

FPGA devices are an important component in many modern devices. This means that it is important that VLSI designers have a thorough knowledge of how to optimize designs for FPGAs. While the design flows for ASICs and FPGAs are similar, there are many differences as well due to the limitations inherent in FPGA devices. To be able to use an FPGA efficiently it is important to be aware of both the strengths and weaknesses of FPGAs. If an FPGA design should be ported to an ASIC at a later stage it is also important to take this into account early in the design cycle so that the ASIC port will be efficient.

This thesis investigates how to optimize a design for an FPGA through a number of case studies of important SoC components. One of these case studies discusses high speed processors and the tradeoffs that are necessary when constructing very high speed processors in FPGAs. The processor has a maximum clock frequency of 357 MHz in a Xilinx Virtex-4 devices of the fastest speedgrade, which is significantly higher than Xilinx' own processor in the same FPGA.

Another case study investigates floating point datapaths and describes how a floating point adder and multiplier can be efficiently implemented in an FPGA.

The final case study investigates Network-on-Chip architectures and how these can be optimized for FPGAs. The main focus is on packet switched architectures, but a circuit switched architecture optimized for FPGAs is also investigated.

All of these case studies also contain information about potential pitfalls when porting designs optimized for an FPGA to an ASIC. The focus in this case is on systems where initial low volume production will be using FPGAs while still keeping the option open to port the design to an ASIC if the demand is high. This information will also be useful for designers who want to create IP cores that can be efficiently mapped to both FPGAs and ASICs.

Finally, a framework is also presented which allows for the creation of custom backend tools for the Xilinx design flow. The framework is already useful for some tasks, but the main reason for including it is to inspire researchers and developers to use this powerful ability in their own design tools.

Populärvetenskaplig Sammanfattning

En fältprogrammerbar grindmatris (FPGA) är ofta en viktig komponent i många moderna apparater. Detta innebär att det är viktigt att personer som arbetar med VLSI-design vet hur man optimerar kretsar för dessa. Designflödet för en FPGA och en applikationsspecifik krets (ASIC) är liknande, men det finns även många skillnader som bygger på de begränsningar som är inbyggda i en FPGA. För att kunna utnyttja en FPGA effektivt är det nödvändigt att känna till både dess svagheter och styrkor. Om en FPGA baserad design behöver konverteras till en ASIC i ett senare skede är det också viktigt att ta med detta i beräkningen i ett tidigt skede så att denna konvertering kan ske så effektivt så mycket.

Denna avhandling undersöker hur en design kan optimeras för en FPGA genom ett antal fallstudier av viktiga komponenter i ett system på chip (SoC). En av dessa fallstudier diskuterar en processor med hög klockfrekvens och de kompromisser som är nödvändiga när en sådan konstrueras för en FPGA. I en Virtex-4 med högsta hastighetsklass kan denna processor användas med en klockfrekvens av 357 MHz vilket är betydligt snabbare än Xilinx egen processor på samma FPGA.

En annan fallstudie undersöker datavägar för flyttal och beskriver hur en flyttalsadderare och multiplicerare kan implementeras på ett effektivt sätt i en FPGA.

Den sista fallstudien undersöker arkitekturer för nätverk på chip och

hur dessa kan optimeras för FPGAer. Huvudfokus i denna del är paketbaserade nätverk men ett kretskopplat nätverk optimerat för FPGAer undersöks också.

Alla fallstudier innehåller också information om eventuella fallgropar när kretsarna ska konverteras från en FPGA till en ASIC. I detta fall är fokus främst på system där småskalig produktion använder FPGAer där det är viktigt att hålla möjligheten öppen till en ASIC-konvertering om det visar sig att efterfrågan på produkten är hög. Detta avsnitt är även av intresse för utvecklare som vill skapa IP-kärnor som är effektiva i både FPGAer och i ASICs.

Slutligen så presenteras ett ramverk som kan användas för att skapa skräddarsydda backend-verktyg för det designflöde som Xilinx använder. Detta ramverk är redan användbart till vissa uppgifter men den största anledningen till att detta inkluderas är att inspirera andra forskare och utvecklare till att använda denna kraftfulla möjlighet i sina egna utvecklingsverktyg.

Abbreviations

- ASIC: Application Specific Integrated Circuit
- CLB: Configurable Logic Block
- DSP: Digital Signal Processing
- DSP48, DSP48E: A primitive optimized for DSP operations in some Xilinx FPGAs
- FD,FDR,FDE: Various flip-flop primitives in Xilinx FPGAs
- FIR: Finite Impulse Response
- FFT: Fast Fourier Transform
- FPGA: Field Programmable Gate Array
- HDL: Hardware Description Language
- IIR: Infinite Impulse Response
- IP: Intellectual Property
- kbit: Kilobit (1000 bits)
- kB: Kilobyte (1000 bytes)
- KiB: Kibibyte (1024 bytes)
- LUT: Look-Up Table

- LUT1, LUT2, . . . , LUT6: Lookup-tables with 1 to 6 inputs
- MAC: Multiply and Accumulate
- MDCT: Modified Discrete Cosine Transform
- NoC: Network on Chip
- NRE: Non Recurring Engineering
- OCN: On Chip Network
- PCB: Printed Circuit Board
- RTL: Register Transfer Level
- SRL16: A 16-bit shift register in Xilinx FPGAs
- VLSI: Very Large Scale Integration
- XDL: Xilinx Design Language

Acknowledgments

There are many people who have made this thesis possible. First of all, without the support of my supervisor, Prof. Dake Liu, this thesis would never have been written. Thanks for taking me on as your Ph.D. student!

I would also like to acknowledge the patience with my working hours that my fiancee, Helene Karlsson, has had during the last year. Thanks for your understanding!

I've also had the honor of co-authoring publications with Johan Eilert, Per Karlström, Daniel Wiklund, Mikael Olausson, and Di Wu.

Additionally, in no particular order[1] I would like to acknowledge the following:

- The community on the *comp.arch.fpga* newsgroup for serving as a great inspiration regarding FPGA optimizations.
- Göran Bilski from Xilinx for an interesting discussion about soft core processors.
- All present and former Ph.D. students at the division of Computer Engineering.
- Ylva Jernling for taking care of administrative tasks of the bureaucratic nature and Anders Nilsson (Sr) for taking care of administrative tasks of technical nature.
- Pat Mead from Altera for an interesting discussion about Altera's Hardcopy program.

[1]Ensured by entropy gathered from /dev/random.

- All the teaching staff at Datorteknik, especially Lennart Bengtsson who offered much valuable advice when I was given the responsibility of giving the lectures in basic switching theory.

Finally, my parents have always supported me in both good and bad times. Thank you.

Andreas Ehliar, 2009

Contributions

My main contributions are:

- An investigation of the design tradeoffs for the data path and control path of a 32-bit microprocessor with DSP extensions optimized for the Virtex-4 FPGA. The microprocessor is optimized for very high clock frequencies (around 70% higher than Xilinx' own Microblaze processor). Extra care was taken to keep the pipeline as short as possible while still retaining as much flexibility as possible at these frequencies. The processor should be very good for streaming signal processing tasks and adequate for general purpose tasks when compared with other FPGA optimized processors. Finally, it is also possible to port the processor to an ASIC with high performance.

- A network-on-chip architecture optimized for very high clock frequencies in FPGAs. The focus of this work was to take a simple packet switched NoC architecture and push the performance as high as possible in an FPGA. When published this was probably the fastest packet switched NoC for FPGAs and it is still very competitive when compared with all types of FPGA based NoCs. This NoC architecture has also been released as open source to allow other researchers to access a high performance NoC architecture for FPGAs and improve on it if desired.

- High performance floating point adder and multiplier with performance comparable to commercially available floating point modules for Xilinx FPGAs.

- A library for analysis and manipulation of netlists in the backend part of Xilinx' design flow. This library and some supporting utilities, most notably a logic analyzer core inserter, has also been released as open source to serve as an inspiration for other researchers interested in this subject.
- An investigation of how various kinds of FPGA optimizations will impact the performance and area of an ASIC port.

Preface

This thesis presents my research from October 2003 to January 2009. The following papers are included in the thesis:

Paper I: Using low precision floating point numbers to reduce memory cost for MP3 decoding

The first paper, written in collaboration with Johan Eilert, describes a DSP processor optimized for MP3 decoding. By using floating point arithmetic it is possible to lower the memory demands of MP3 decoding and also simplify firmware development. It was published at the International Workshop on Multimedia Signal Processing, 2004.

Contributions: The contributions in this paper from Johan Eilert and me are roughly equal.

Paper II: An FPGA based Open Source Network-on-chip Architecture

The second paper presents an open source packet switched NoC architecture optimized for Xilinx FPGAs. It was published at FPL 2007. The source code for this NoC is also available under an open source license to allow other researchers to build on this work.

Paper III: Thinking outside the flow: Creating customized backend tools for Xilinx based designs

The third paper presents the PyXDL tool which allows XDL files to be analyzed and edited from Python. It was published at FPGAWorld 2007. The PyXDL tool is available as open source.

Paper IV: A High Performance Microprocessor with DSP Extensions Optimized for the Virtex-4 FPGA

The fourth paper, written in collaboration with Per Karlström presents a high performance microprocessor which is heavily optimized for the Virtex-4 FPGA through both manual instantiation of FPGA primitives and floorplanning. It was published at Field Programmable Logic and Applications, 2008.

Contributions: I designed most of the architecture of the processor, Per Karlström helped me with reviewing the architecture of the processor and evaluated whether it was possible to add floating point units to the processor.

Paper V: High performance, low-latency field-programmable gate array-based floating-point adder and multiplier units in a Virtex 4

The fifth paper, written in collaboration with Per Karlström, studies floating point numbers and how to efficiently create a floating point adder and multiplier in an FPGA. It was published by IET Computers & Digital Techniques, Vol. 2, No. 4, 2008.

Contributions: Per Karlström is responsible for the IEEE compliant rounding modes and the test suite. The remaining contributions in this paper are roughly equal.

Paper VI: An ASIC Perspective on High Performance FPGA Design

The final paper is a study of how various FPGA optimizations will impact an ASIC port of an FPGA based design. It has been submitted for possible publication to the IEEE conference of Field Programmable Logic and Applications, 2009.

Licentiate Thesis

The content of this thesis is also heavily based on my licentiate thesis:

- *Aspects of System-on-Chip Design for FPGAs*, Andreas Ehliar, Linköping Studies in Science and Technology, Thesis No. 1371, Linköping, Sweden, June 2008

Other research interests

Besides the papers included in this thesis my research interests also includes hardware for video codecs and network processors.

Other Publications

- *Flexible route lookup using range search*, Andreas Ehliar, Dake Liu; Proc of the The Third IASTED International Conference on Communications and Computer Networks (CCN), 2005
- *High Performance, Low Latency FPGA based Floating Point Adder and Multiplier Units in a Virtex 4*, Karlström, P. Ehliar, A. Liu, D; 24th Norchip Conference, 2006.

Contents

Chapter 1
Introduction

Field programmable logic has developed from being small devices used mainly as glue logic to capable devices which are able to replace ASICs in many applications. Today, FPGAs are used in areas as diverse as flat panel televisions, network routers, space probes and cars. FPGAs are also popular in universities and other educational settings as their configurability make them an ideal platform when teaching digital design since students can actually implement and test their designs instead of merely simulating them. In fact, the availability of cheap FPGA boards mean that even amateurs can get into the area of digital design.

As a measure of the success that FPGAs enjoy, there are circa 7000 ASIC design starts per year whereas the number of FPGA design starts are roughly 100000 [1]. However, most of the FPGA design starts are likely to be for fairly low volume products as the unit price of FPGAs make them unattractive for high volume production. Similarly, most of the ASIC design starts are probably only intended for high volume products due to the high setup cost and low unit cost of ASICs. Even so, the ASIC designs are likely to be prototyped in FPGAs. And if a low volume FPGA product is successful it may have to be converted to an ASIC.

One of the motivations behind this thesis is to investigate a scenario where an FPGA based product has been so successful that it makes sense to convert it into an ASIC. However, there are many ways that an ASIC and FPGA design can be optimized and not every ASIC optimization can be used in an FPGA and vice versa. If the FPGA design was not designed with an ASIC in mind from the beginning, it may be hard to create such a port. This thesis will classify and investigate various FPGA optimizations to determine whether they make sense to use in a product that may have to be ported to an ASIC. This part of the thesis should also be of interest to engineers who are tasked with creating IP cores for FPGAs if the IP cores may have to be used in ASICs.

Another motivation is simply the fact that the large success of FPGAs of course also means that there is a large need for information about how to optimize designs for these devices. Or, to put it another way, a desire to advance the state of the art in creating designs that are optimized for FPGAs. This effort has focused on areas where we believed that the current state of the art could be substantially improved or substantially better documented.

A more personal motivation is the fact that relatively little research on FPGA optimized design is happening in Sweden. After all, it is more likely that a freshly graduated student from a university will be involved in VLSI design for FPGAs rather than ASICs. My hope is that this thesis can serve as an inspiration for these students and perhaps even inspire other researchers to look further into this interesting field.

The results in this thesis should be of interest for engineers tasked with the creation of FPGA based stand alone systems, accelerators, and soft processor cores.

1.1 Scope of this Thesis

This thesis is mainly based on case studies where important SoC components were optimized for FPGAs. The main case studies are:

- Microprocessors
- Floating point datapath components
- Networks-on-Chip

These were selected as they are representative of a variety of interesting and varied architectural choices where we believed that we could improve the state of the art. For example, when we began the microprocessor research project there were no credible DSP processors optimized for FPGAs. The NoC situation was similar in that most NoC research had been done on ASICs and very few NoCs had been optimized for FPGAs in any way. The floating point datapath is slightly different as there were already a few floating point adder and multiplier with good performance available. However, all of these were proprietary cores without any documentation of how the high performance was reached.

These case studies are also interesting because they cover a fairly wide area of interesting optimization problems. Microprocessors consists of many small but latency critical datapaths. In contrast, when floating point components are used to create datapath based architectures, high throughput is required, but the latency is usually not as important. NoCs are interesting because the datapaths in a NoC are intended mainly to transport data as fast as possible instead of transforming data.

The opportunities and pitfalls when porting a design which has been heavily optimized for an FPGA is also discussed for all of these case studies.

Finally, a framework is presented which allows a designer to create backend tools for the Xilinx design flow, either to analyze or modify a design after it has been placed and routed.

1.2 Organization

The first part of this thesis contains important background information about FPGAs, FPGA optimizations, design flow, and methods. This part also contains a comparison of the performance and area cost for different components in both FPGAs and ASICs.

Part II contains an investigation of two microprocessors (one FPGA friendly processor and one FPGA optimized processor). This part also contains a description of the floating point adder and multiplier. Part III contains both a brief overview of Networks-on-Chip and a description and comparison of FPGA optimized packet switched, circuit switched, and statically scheduled NoCs. Part IV describes a way to create custom tools to analyze and manipulate already created designs which will be interesting for engineers wanting to create their own backend tools. Part V contains conclusions and also a discussion about possible future work. This section also contains a list of all ASIC porting guidelines that are scattered through the thesis. Finally, Part VI contains the publications that are relevant for this thesis[1].

[1]The electronic version of this thesis does not contain Part VI.

Part I

Background

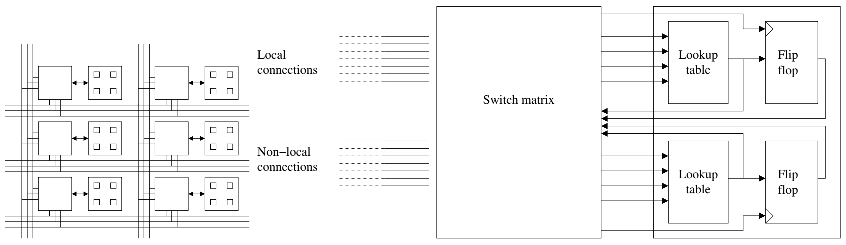
Chapter 2
Introduction to FPGAs

An FPGA is a device that is optimized for configurability. As long as the FPGA is large enough, the FPGA is able to mimic the functionality of any digital design. When using an FPGA it is common to use a HDL like VHDL or Verilog to describe the functionality of the FPGA. Specialized software tools are used to translate the HDL source code into a configuration bitstream for the FPGA that instructs the many configurable elements in the FPGA how to behave.

Traditionally, an FPGA consisted of two main parts: routing and configurable logic blocks (CLB). A CLB typically contains a small amount of logic that can be configured to perform boolean operations on the inputs to the CLB block. The logic can be constructed by using a small memory that is used as a lookup table. This is often referred to as a LUT.

The logic in the CLB block is connected to a small number of flip-flops in the CLB block. The CLBs are also connected to switch matrices that in turn are connected to each other using a network of wires. A schematic view of a traditional FPGA is shown in Figure 2.1.

In reality, todays FPGAs are much more complex devices and a number of optimizations have been done to improve the performance of important design components. For example, in Xilinx FPGAs, a CLB has been further divided into slices. A slice in most Xilinx devices for example, consists of two LUTs and two flip-flops. There is also special logic in the slice to simplify common operations like combining two LUTs into a



(a) FPGA overview (b) CLB and switch matrix

Figure 2.1: Schematic view of an FPGA

larger LUT and creating efficient adders.

2.1 Special Blocks

The basic architecture in Figure 2.1 is not very optimal when a memory is needed. To improve the performance of memory dense designs, modern FPGAs have embedded memory blocks capable of operating at high speed. In a Virtex-4 FPGA, an embedded memory, referred to as a block RAM, contains 512 words of 36 bits each. (It is also possible to configure half of the LUTs in the CLBs as a small memory containing 16 bits, this is referred to as a distributed RAM.) In contrast, a Stratix-3 from Altera have embedded memory blocks of different sizes. There are many blocks that contains 256 36-bit words and a few blocks with 2048 72-bit words.

To improve the performance of arithmetic operations like addition and subtraction, there are special connections available that allows a LUT to function as an efficient full adder. This is referred to as a carry chain. A carry chain is also connected to adjacent slices to allow for larger adders to be created.

To improve the performance of multiplication, hard wired multiplier blocks are also available in most FPGAs, sometimes combined with other logic like an accumulator. In a Virtex-4, a block consisting of a multiplier and an accumulator is called a DSP48 block. The multiplier is 18×18 bits and the accumulator contains 48 bits. There are also special connections available to easily connect several DSP48 blocks to each other that can be used to build efficient FIR filters or larger multipliers for example.

In some FPGAs there are also more specialized blocks like processor cores, Ethernet controllers, and high speed serial links.

2.2 Xilinx FPGA Design Flow

A typical FPGA design flow consists of the following steps (in more advanced flows some of these steps may be combined):

- **Synthesis:** Translate RTL code into LUTs, flip-flops, memories, etc.
- **Mapping:** Map LUTs and flip-flops into slices
- **Place and route:** First decide where all slices, memory blocks, etc should be placed in the FPGA and then route all signals that connects these components
- **Bitfile generation:** Convert the netlist produced by the place and route step into a bitstream that can be used to configure the FPGA
- **FPGA Configuration:** Download the bitstream into the FPGA

There are also other steps that are optional but can be used in some cases. A static timing analyzer, for example, can be used to determine the critical path of a certain design. It can also be used to make sure that a design is meeting the timing constraints, but this is seldom necessary as the place and route tool will usually print a warning if the timing constraints are not met.

There are special tools available to inspect and modify the design. A floorplanning tool allows a designer to investigate the placement of all components in a design and change the placement if necessary. An FPGA editing tool can be used to view and edit the exact configuration of a CLB and other components in terms of logic equations for LUTs, flip-flop configuration, etc. It will also show how signals are routed in the FPGA and can also change the routing if necessary.

2.3 Optimizing a Design for FPGAs

Optimizing an algorithm to an FPGA will use the same general ideas as optimizing for ASICs. The basic idea is to use as much parallelization as required to achieve the required performance. However, the details are not quite the same as described below.

2.3.1 High-Level Optimization

Adding pipeline-stages, if possible, is a simple way to increase the performance in both FPGAs and ASICs. It is usually especially area efficient in FPGAs, since most FPGA designs are not flip-flop limited, which means that there are a lot of flip-flops available and an unused flip-flop is a wasted flip-flop. Although a general technique, some designs cannot easily tolerate extra pipeline stages (e.g. microprocessors) and other methods are required in those cases.

Another way to improve the performance of an FPGA is by utilizing all capabilities of the embedded memories. In ASICs, dual port memories are more expensive than single port memories. Therefore it makes sense to avoid dual port memories in many situations. However, in FPGAs, the basic memory block primitive is usually dual-ported by default. Therefore it makes sense to use the memories in dual-ported mode if it will simplify an algorithm. Similarly, each memory block in an FPGA has a fixed size. Therefore it can make sense to decrease logic usage at a cost of increased memory usage as long as the memory usage for that part of the design will still fit into a certain block RAM.

Similarly, the multipliers in an FPGA have a fixed size (e.g. 18×18 bits, in a Virtex-4). When compared to an ASIC where it is easy to just generate a multiplier of another size, it is worthwhile to make sure that the algorithm doesn't need larger multipliers than provided in the FPGA. This works the other way around as well. Coming up with a way to reduce a multiplier from 16×16 bits to a mere 13×13 bits at the cost of additional logic is not going to help in terms of resource utilization (although it may improve the timing slightly).

2.3.2 Low-level Logic Optimizations

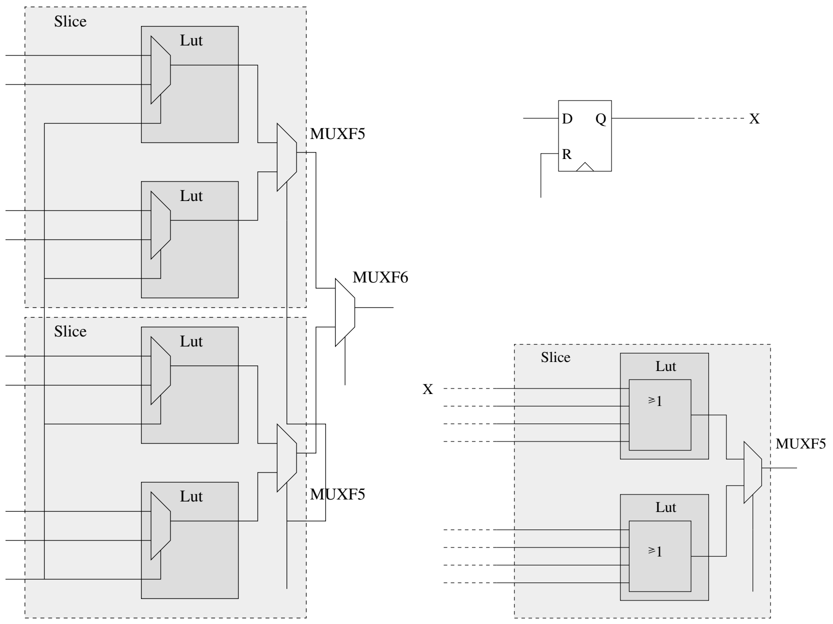
In many cases there is no need to go further than the optimizations mentioned in the previous section. However, if the performance that was reached by the previous optimizations was not satisfactory, it is possible to fine-tune the architecture for a certain FPGA. Some examples of how to do this are:

- Modify the critical path to take advantage of the LUT structure. For example, if an 8-to-1 multiplexer is required it will probably be synthesized as shown in Figure 2.2(a) when synthesized to a Virtex-4, utilizing a total of 4 LUTs distributed over two slices and taking advantage of the built-in MUXF5 and MUXF6 primitives. However, if it is possible to rearrange the logic so that the inputs to the mux are zero in case the input is not going to be selected, the mux can be rearranged using a combination of or gates and muxes. In Figure 2.2(b), the zero is arranged by using the reset input of a flip-flop directly connected to the mux.

  Other ways in which the design can be fine tuned is to make sure that the algorithms are mapped to the FPGA in such a way that adders can be efficiently combined with other components such as muxes while keeping the number of logic levels low.

- In a Virtex-4 some LUTs can be configured as small shift registers. This makes it very efficient to add small delay lines and FIFOs to a design.

- Bit serial arithmetics can be a great way to maximize the throughput of a design by minimizing the logic delays at a cost of increased complexity. To be worthwhile, a large degree of parallelism must be available in the application. Bit (or digit) serial algorithms can also be a very useful way to minimize the area cost of modules that are required in a system but have low performance requirements, such as for example a real time clock.



(a) Using four LUTs configured as 2-to-1 muxes (b) Using two LUTs configured as or gates

Figure 2.2: Example of low level logic optimization: 8-to-1 mux

2.3.3 Placement Optimizations

If the required performance is not reached through either high or low level logic optimizations it is usually possible to gain a little more performance by floorplanning. There are two kinds of floorplanning available for an FPGA flow. The easiest is to tell the backend tools to place certain modules in certain regions of the FPGA. This is rather coarse grained but can be a good way to ensure that the timing characteristics of the design will not vary too much over multiple place and route runs. The other way is to manually (either in the HDL source code or through the use of a graphical user interface), describe how the FPGA primitives should be placed. For example, if the critical path is long (several levels of LUTs), it makes sense to make sure that all parts of it are closely packed, preferably inside a single CLB due to the fast routing available inside a CLB. If the design consists of a complicated data path, the entire data path could be designed using RLOC attributes to ensure that the data path will always be placed in a good way.

The advantage of floorplanning has been investigated in [2], and was found to be able to improve the performance from 30% to 50%. However, since this was published in 2000, a lot of development has happened in regards to automatic place and route. Today, the performance increase that can be gained from floorplanning is closer to 10% or so and it is often enough to floorplan only the critical parts of the design [3]. It should also be noted that it is very easy to reduce the performance of a design by a slight mistake in the floorplanning.

Finally, if it is still not possible to meet timing even though floorplanning has been explored, it might be possible to gain a little more performance by manually routing some critical paths. The author is not aware of any investigation into how much this will improve the performance, but the general consensus seems to be that the performance gains are not worth the source code maintenance nightmare that manual routing leads to.

2.3.4 Optimizing for Reconfigurability

The ability to reconfigure an FPGA can be a powerful feature, especially for the FPGA families where parts of the FPGA can be reconfigured dynamically without impacting the operations of other parts of the FPGA. This can be a very powerful ability in a system that has to handle a wide variety of tasks under the assumption that it doesn't have to handle all kinds of tasks simultaneously. In that case it may be possible to use reconfiguration, similarly to how an operating system for a computer is using virtual memory. That is, swap in hardware accelerators for the current workload and swap out unused logic. This can lead to significant unit cost reductions as a smaller FPGA can be used without any loss of functionality.

While this ability is powerful, it is only supported for a few FPGAs and the support from the design tools is rather limited. But the configurability of an FPGA can still be useful, even if it is not possible to reconfigure the FPGA dynamically. One example is to use a special FPGA bitstream for diagnostic testing purposes (e.g. testing the PCB that the FPGA is located on). While such functionality could be included in the main design it may be better from a performance and area perspective to use a dedicated FPGA configuration for this purpose.

2.4 Speed Grades, Supply Voltage, and Temperature

Due to differences in manufacturing, the actual performance of a certain FPGA family can vary by a significant amount between various specimens. Faster devices are marked with a higher speed grade than slower devices and can be sold at a premium by the FPGA manufacturers. There is no exact definition of what a speedgrade means, but according to the author's experience of Xilinx' devices, going up one speedgrade means that the maximum clock frequency will increase around 15% depending on the design and the FPGA. An example of the impact of the speedgrade on two designs are shown in Table 2.1. (The unit that is tested is a small microcontroller with a bus interface, serial port and parallel port.) While an upgrade in speed grade is an easy way to improve the performance of a design, it is not cheap. For example, a XC4VLX80-10-FFG1148 had a cost of $1103 in quantities of one unit on the 15th of October 2008 on NuHorizon's webshop. The same device in speed grade 11 had a cost of $1358 and speed grade 12 a cost of $1901. It is clearly a good idea to use the slowest speedgrade possible.

Another factor that is seldom mentioned in in FPGA related publications is the supply voltage and temperature. By default, the static timing analysis tools uses the values for the worst corner (highest temperature and lowest voltage). For some applications this is not necessary. If good voltage regulation is available, which guarantees that the supply voltage

Design	Device	Speedgrade	F_{max} [MHz]
Small	Virtex-4	10	210
Microcontroller		11	246
		12	277

Table 2.1: Impact of speedgrade on a sample FPGA design

	85 °C	65 °C	45 °C	25 °C	0 °C
1.14V	323.2	324.1	325.3	326.4	329.6
1.18V	334.0	335.1	336.2	337.4	340.8
1.22V	344.5	345.7	346.9	347.9	351.4
1.26V	354.5	355.6	356.8	357.8	361.4

Table 2.2: Timing analysis using different values for supply voltage and temperature

will not approach the worst case, we can specify a higher minimum voltage to the timing analyzer. Similarly, if good cooling is available, we can specify that the FPGA will not exceed a certain temperature.

In Table 2.2, we can see the impact of these changes on a microprocessor design in a Virtex-4 (speedgrade 12). In the upper left corner the worst case with minimum supply voltage and maximum temperature is shown. The design will work at 323.2 MHz in all temperature and voltage situations that the FPGA is specified for. On the other hand, if an extremely good power and cooling solution is used, we could clock the design at 361.4 MHz with absolutely no margin for error. This is a difference of over 10% without having to change anything in the design! It can therefore be worthwhile to think about these values when synthesizing a design for a certain application. Many real life designs will not need to use the worst case values. However, results in publications are rarely, if ever, based on other than worst case values. Therefore Table 2.2 is the only place in this thesis where results are reported that are not based on worst case temperature and supply voltage conditions.

Chapter 3
Methods and Assumptions

Normally, the design flow for an FPGA based system will go through the following design steps:

1. Idea
2. Design specification
3. HDL code development
4. Verification of HDL code
5. Synthesis/Place and route/bitstream generation
6. Manufacturing
7. (Debug of post manufacturing problems if necessary)

This is of course a simplified view. In practice, the process of writing a design specification is a science in and of itself. Likewise with HDL code implementation and verification, not to mention manufacturing. There are usually some overlap between the phases as well, especially between the verification and development phase.

The method used for the majority of designs described in this thesis is based on the method described above. The most prominent idea in our method is the fact that a rigid design specification is an obstacle to a high performance VLSI design. There is therefore a considerable overlap between the design specification phase and the HDL code development phase. In fact, it is necessary to quickly identify areas that are likely to cause performance problems and prototype these to gain the knowledge that is necessary to continue with the design specification.

Another difference between a normal design flow and the design flow employed in this thesis (and many other research projects) is that a lot of effort was spent on low level optimizations with the intention of reaching the very highest performance. This is uncommon in the industry where performance that is "good enough" is generally accepted. To know where the low level optimizations are required it is necessary to study the output from the synthesis tool and the output from the place and route tool. If there is something clearly suboptimal in the final netlist it may be fixed either by rewriting the HDL code (possibly by instantiating low level FPGA primitives) or by manual floorplanning. This method is described as "construct by correction" in [4] (which also contain a good overview of the entire design process). Another description of the design flow (with a focus on ASIP development) can be found in [5].

3.1 General HDL Code Guidelines

This thesis assumes that FPGA friendly rules are used when writing the HDL code. Some of the more important guidelines are:

- Use clock enable signals instead of gating the clock
- Do not use latches
- Use only one clock domain if at all possible
- Do not use three-state drivers inside the chip

A thorough list of important guidelines for FPGA design can be found in for example [6]. It should also be noted that many of the guidelines for FPGA design are also useful for ASIC designs. For an in depth discussion of guidelines for VLSI design, see for example [4].

3.2 Finding F_{MAX} for FPGA Designs

While the parameters mentioned in Section 2.4 are easy to understand, there are other parameters that impact the maximum clock frequency. Perhaps the most important is the synthesis tools and the place and route tools. Depending on the tools that are used, different results will be obtained. It is probably a good idea to request evaluation versions of the various synthesis tools that are available from time to time to see if there is a reason to change tool. It should also be noted that it is not always a good idea to upgrade the tools. It is not uncommon to find that an older version will produce better results for a certain design than the upgraded version. The author has seen an older tool perform more than 10% better than a newer tool on a certain design. In some cases it is even possible that the best results will be achieved when combining tools from various versions.

All tools in the FPGA design flow have many options that will impact the maximum frequency, area, power usage, and sometimes even the correctness of the final design. Many of these choices can also be made on a module by module case or even on a line by line case in the HDL source code. Finding the optimal choices for a certain design is not an easy task. It is also not uncommon that the logical choice is not the best solution (e.g. sometimes a design will synthesize to a higher speed if it is optimized for area instead of speed).

3.2.1 Timing Constraints

Perhaps the most important of these options are the timing constraints given to the tools. The tools will typically not optimize a design further when it has reached the user specified timing constraints. If the timing constraint cannot be achieved, different tools behave in different ways. Xilinx' tools will spend a lot of time trying to meet a goal that cannot be achieved. It is also not uncommon that an impossible timing constraint will mean that the resulting circuit will be slower than if a hard but possible timing constraint was specified. Altera's tools on the other hand does not seem to be plagued by this particular problem though. If a very hard timing constraint is set, Altera's place and route tool will give a design with roughly the same F_{max} as can be found when sweeping the timing constraint over a wide region. (This behavior has been tested with ISE 10.1 and Quartus II 8.1. The same behavior has also been reported in [7].)

Another important thing to consider is clock jitter. As clock frequencies increase, jitter is becoming a significant issue that designers need to be aware of. It is possible to specify the jitter of the incoming clock signals in the timing constraints. The use of modules like DCMs and DLLs will also add to the jitter (this jitter value is usually added automatically by the backend tools). This is important since the jitter will probably account for a significant part of the clock period on a high speed design. However, since it seems to be very unusual to specify any sort of clock jitter when publishing maximum frequencies for FPGA designs, the number presented in this thesis will also ignore the effects of jitter[1]. The careful designer will therefore compensate for the lack of jitter when evaluating the maximum frequency of different solutions for use in his or her system.

3.2.2 Other Synthesis Options

Other issues that will have an impact on the maximum frequency of a certain design are the settings of the synthesis and backend tools. The following is a list of some of the more important options:

- Overall optimization levels: If the design can relatively easily meet the timing requirements there is no need to spend a lot of time on optimizations.
- Retiming: The tools are allowed to move flip flops to try to balance the pipeline for maximum speed
- Optimization goal: Area or speed

[1]The author has yet to see an FPGA related publication where the authors specifically state that they have used anything but 0 for the clock jitter.

- Should the hierarchy of the HDL design be kept or flattened to allow optimizations over module boundaries?
- Resource sharing: Allows resources to be shared if they are not used at the same time.

In this thesis many of these options have been tweaked to produce the best results for the case studies. As the HDL code itself gets more and more optimized it is common that many optimizations are turned off since they will interfere with the manual optimization that has already been done.

3.3 Possible Error Sources

In a work such as this there are a wide variety of possible error sources. Perhaps the most insidious source of error is in the form of bugs in the CAD tools. The author have encountered serious bugs of various kinds in many CAD tools during his time as a Ph.D. student. Some bugs are easy to detect by the fact that the tool simply crashes with a cryptic error message. Other bugs are harder to detect, such as when the wrong logic is synthesized without any warning or error message to indicate this.

This is not intended as criticism towards any vendor but rather as an observation of fact. Almost anyone who has used a program as complex as a CAD tool for a longer period of time will discover bugs in it. And anyone who has tried to develop a program as complex as a CAD tool knows how hard it is to completely eliminate all bugs. Overall, the vendors have been very responsive to bug reports as they are of course also interested in removing bugs.

3.3.1 Bugs in the CAD Tools

Sometimes a synthesis bug is easy to detect, for example, if the area of the design is significantly smaller than expected it is possible that a bug in the optimization phase has removed logic that is actually used in the design. Sometimes bugs introduced by the synthesis or backend tools will not have a dramatic effect on the area of a design and must be detected by actually using the design in an FPGA. To guard against this possibility, all major designs in this thesis have been tested on at least one FPGA. While minor bugs caused by the backend tools could still be present, they are unlikely to ruin the conclusions of this thesis as they would be present in fairly minor functionality of the designs that would only be triggered under special circumstances. It should also be noted that it is possible to simulate the synthesized netlist, which is yet another way to detect whether the synthesis tool has done something wrong. (Bugs in the backend tools are harder to detect.)

Another source of error that is even harder to detect is bugs in the static timing analysis where a certain path is reported as being faster than it actually is. This kind of error could mean that the maximum frequency of a design will not be as high as the value reported by the tool. This is harder to detect without testing the design on a wide variety of FPGAs (ideally FPGAs that are known to just barely pass timing tests for the speedgrade under test). Since this is clearly impractical, the only choice is to trust the values reported by the static timing analysis tool (unless the values that are reported are very suspicious).

Yet another form of possible problem is when the HDL simulator does not simulate the hardware correctly. The most likely way to find such bugs are to observe them during simulation. Another way is to observe that the FPGA does not behave as the simulation predicts (although this can also mean that the synthesis tool is doing something wrong).

This situation is even worse for ASIC based design flows as it is not practical to manufacture a small testdesign just to see if it works. In summary, we have little choice but to rely on the tools. Yet it is important to stay on guard and not trust the tools 100%, especially when they report odd or very odd results.

3.3.2 Guarding Against Bugs in the Designs

While tool bugs are very dangerous they are also quite rare. Another more common source of bugs is simply the designer himself[2]. The traditional way to guard against this is to write comprehensive testbenches and test suites. All major designs in this thesis have testbenches that are fairly comprehensive. Extra care has been taken to verify the most important details and the details that are thought most likely to contain bugs. For example, when writing the test suite for the arithmetic unit described in Section 7.1, care was taken to exercise all valid forwarding paths. However, only a few different values were tested. That is, not all possible combinations of input values were tested for addition and subtraction due to the huge amount of time this would take and the low likelihood that there would be a bug in the adder itself.

There are no known bugs in the current version of the designs described in this thesis, but it is possible that there are unknown bugs. However, since care has been taken to exercise the most important parts of the designs thoroughly, it is very likely that the remaining bugs will be minor issues that will have no or little effect on the conclusions drawn in this thesis.

Finally, it should also be noted that testbenches were not written for many of the simple test designs in Chapter 5. It was felt that the correctness of the source code of for example a simple adder could be ensured merely by inspecting the source code and by looking at the synthesis report, mapping report, and in some cases the actual logic that was synthesized. However, the more tricky designs described in Chapter 5, such as the MAC unit, do have testbenches.

3.3.3 A Possible Bias Towards Xilinx FPGAs

Due to the author's extensive experience with Xilinx FPGAs, much of this thesis has been written with Xilinx FPGAs in mind. All of the case

[2]At this point honesty compels the author to admit that he has been responsible for more than one bug in his life...

studies discussed in this thesis were optimized for Xilinx FPGAs, and often a particular Xilinx FPGA family as well. Care has been taken to avoid an unfair bias towards Xilinx in the parts that discuss other FPGA families but it is nevertheless possible that some bias may still be present and it is only fair to warn the reader about this.

There is also a clear bias towards SRAM based FPGAs in this text as FPGA families manufactured using flash and antifuse technologies are not typically designed for high performance.

3.3.4 Online Errata

As described above, there are many possible error sources. While care has been taken to minimize these, few works of this magnitude are ever completely free of minor errors. The reader is encouraged to visit either the author's homepage at `http://www.da.isy.liu.se/~ehliar/` or the page for the thesis at `http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-16732` to see if there are any erratas published. (The later URL is guaranteed by Linköpings University to be available for a very long period of time.) Likewise, if the reader encounters something that seems suspicious in the thesis, the author would very much like to know about this.

3.4 Method Summary

The method used in this thesis to optimize FPGA designs can be summarized as follows:

- Do not fix the design specification until a prototype has shown where the performance problems are located and a reasonable plan on how to deal with the performance problems has been finalized. To reach the highest performance it may be necessary to implement a prototype with much of the functionality required of the final system before the design specification can be finalized.

- Use synchronous design methods and avoid using techniques such as latches and clock gating
- Investigate if the synthesis tool have used suboptimal constructs. If so, rewrite the HDL code to infer or instantiate better logic.
- Investigate if floorplanning can help the performance as well.
- Vary synthesis and backend options to determine which options lead to the highest performance.
- Manage the timing constraints appropriately for the tool that is used for place and route (e.g. increase the timing constraints iteratively until it is no longer possible to meet timing when using Xilinx devices)
- Further, the timing constraint settings assume a clock with no jitter and the worst case parameters for temperature and supply voltage
- Be wary of bugs in both the design and the CAD tools. Always check that the reported area and performance are reasonable.

Chapter 4
ASIC vs FPGA

There are many similarities when designing a product for use with either an FPGA or an ASIC. There are also many differences in the capabilities of an FPGA and an ASIC. This chapter will concentrate on the most important differences.

4.1 Advantages of an ASIC Based System

The advantages of an ASIC can be divided into four major areas: Unit cost, performance, power consumption and flexibility.

4.1.1 Unit Cost

One of the biggest advantages which an ASIC based product enjoys over an FPGA based product is a significantly lower unit cost once a certain volume has been reached. Unfortunately the volume required to offset the high NRE costs of an ASIC is very high which means that many projects are never a candidate for ASICs. For example, in [8], the authors show an example where the total design cost for a standard cell based ASIC is \$5.5M whereas the design cost for the FPGA based product is \$165K. It is clear that the volume has to be quite high before an ASIC can be considered. It should also be noted that this comparison is for a 0.13 μm process. More modern technology nodes have even higher NRE costs and therefore even higher volumes are necessary before it makes sense to consider an ASIC.

4.1.2 Higher Performance

Another reason for using an ASIC is the higher performance which can be gained by using a modern ASIC process. During a comparison of over 20 different designs it was found that an ASIC design was on average 3.2 times faster than an FPGA manufactured on the same technology node [9]. This is slightly misleading though, as FPGAs are often manufactured using the latest technologies whereas an ASIC could be manufactured using an older technology for cost reasons. In this case the performance gap will be lower.

4.1.3 Power Consumption

An ASIC based system usually has significantly lower power consumption than a comparable FPGA based system. While some FPGAs are specifically targeting low power users, such as the new iCE65 from SiliconBlue, most FPGAs are not targeted specifically at low power users.

The main reason for this is of course the reconfigurability of the FPGA. There is a lot of logic in an FPGA which is used only for configuration. While the dynamic power consumption of the reconfiguration logic is practically 0, all of the configuration logic contributes to the leakage power.

Another reason why ASICs are better from a power consumption perspective is that it is easier to implement power reduction techniques like clock gating and power gating.

While it is possible to perform clock gating in an FPGAs, it is seldomly used in practice. One reason is that FPGAs have a limited number of signals optimized for clock distribution. While flip-flops in an FPGA can also be fed from a local connection, this will complicate static timing analysis. FPGA vendors strongly recommend users to avoid clock gating and to use the clock enable signal of the flip-flops instead.

While clock gating is possible but hard to do in an FPGA today, selective power gating is not possible in modern FPGAs. However, in Actel's Igloo [10] FPGA it is possible to freeze the entire FPGA by using a special *Flash*Freeze* pin. While Actel do not say exactly how this is implemented, it is reasonable to assume that some sort of power gating is involved. Spartan 3A FPGAs has a similar mode activated by a suspend pin which allows the device to retain its state while in a low power mode.

True selective power gating has also been investigated in a modified Spartan 3 architecture [11], but the authors state that there is not enough commercial value in such features yet due to the performance and area penalty of the power gating features.

4.1.4 Flexibility

The final main reason for using an ASIC instead of an FPGA is the flexibility you gain with an ASIC. An ASIC allows the designer to implement many circuits which are either impossible or impractical to create in the programmable logic of an FPGA. This includes for example A/D converters, D/A converters, high density SRAM and DRAM memories, non volatile memories, PLLs, multipliers, serializers/deserializers, and a wide variety of sensors.

Many FPGAs do contain some specialized blocks, but these blocks are selected to be quite general so that they are usable in a wide variety of contexts. This also means that the blocks are far from optimal for many users. In contrast, an ASIC designer can use a block which has been configured with optimal parameters for the application the designer is envisioning. This allows an ASIC designer to both save area and increase the performance.

The ultimate in flexibility is the ability of an ASIC designer to design either part of the circuit or the entire circuit using full custom methods. This allows the designer to create specialized blocks which have no parallel in FPGAs. For example, if a designer wanted to create an image processor with integrated image sensor, this would not be possible to do with the FPGAs currently available.

Full custom techniques are also able to reduce the power and area or increase the performance. For a more thorough discussion about this, see for example [12].

4.2 Advantages of an FPGA Based System

While there are many advantages to an ASIC, there are also many advantages to be had when using an FPGA.

4.2.1 Rapid Prototyping

As there is no manufacturing turn around time for an FPGA based system, a design can quickly be tested and evaluated even though parts of the design are not yet completed. In contrast, most companies would not be able to afford to manufacture a partially functioning ASIC just for testing purposes. This means that developers can start developing the firmware on a partially working prototype when using FPGAs instead of using a much slower simulation model.

A hybrid approach is to use an FPGA for prototyping and an ASIC for production. This is a good and easy solution in some cases, but in other cases it can be tricky. If the system will interface to external interfaces which has to run at high speed, the FPGA might not be able to run at this speed which means that some compromises have to be made. For example, while prototyping an ASIC with a PCI interface, the PCI bus might have to be underclocked as described in [13].

4.2.2 Setup Costs

The setup cost for using a low end FPGA is practically zero. Major FPGA vendors have a low end version of their design tool available for free download. The full version of the vendor tools are also available for a relatively low fee. It is also possible to buy a low end version of an HDL simulator from the FPGA vendors cheaply. There are also a large number of low cost prototype board available for various FPGAs. All of this means that anyone, even hobbyists, can start using an FPGA without having to buy any expensive tools. This is certainly not true for ASICs as the tool cost alone can be prohibitive in many cases.

The other reason for the low setup cost is that the use of an FPGA means that the mask costs associated with an ASIC are avoided which can be a significant saving for a modern technology.

4.2.3 Configurability

There are two main reasons why the configurability of an FPGA is important. The cost reason has already been briefly mentioned in Section 4.2.2. The other reason is that it is possible to deploy bug-fixes and/or upgrades to customers if a reconfigurable FPGA is used.

If a one time programmable FPGA, such as a member of Actel's antifuse based FPGA family, is used, this is of course not possible. It will still be possible to change the configuration of newly produced products without incurring the large NRE cost associated with an ASIC mask change though.

Another interesting possibility is the ability to reconfigure only parts of an FPGA while the FPGA is still running, so called partial dynamic reconfiguration. This capability is present in the Xilinx Virtex series from Virtex-II and up. This could for example mean that a video decoding application could have a wide variety of optimized decoding modules stored in flash memory. As soon as the user wants to play a specific video stream, a decoding module optimized for that particular video format is loaded into the FPGA.

The advantage is of course that a smaller FPGA could be used. The disadvantage is that the tool support for dynamic reconfiguration is limited at the moment. Simulating and verifying such a design is also considerably more difficult. Although a large number of research publications have studied partial reconfiguration it is seldomly used in real applications yet.

Finally, it should also be mentioned that by handling the configuration of an FPGA yourself you don't need to hand over your design files to an outside party. If an ASIC would be used instead of an FPGA, you will have to trust that the foundry will employ strict security measures to keep your design secret. This is not a big problem in most cases, but it could potentially be troublesome when very sensitive information is contained in the design files such as cryptographic keys.

4.3 Other Solutions

There are a few other solutions available which are worth mentioning even though they are mostly outside the scope of this thesis. If the price of a large FPGA is a concern, Xilinx has a product called Easypath [14]. The idea behind this product is that a certain design will only utilize a small amount of the available routing resources in an FPGA. If an unused part of the routing is not working correctly, this doesn't matter to that particular design. In fact, only the routing which is actually used has to be tested which will reduce the testing cost significantly. Xilinx guarantees that the FPGAs they sell under this program will work correctly for up to two customer specific bitstreams. The advantage is of course that the customer will get a substantial discount for a product that works identically to a fully tested FPGA. The disadvantage is that the configurability of the FPGA can no longer be fully used. While Xilinx guarantees that all LUTs have been tested, which allows some bugs to be fixed, it is no longer possible to reconfigure the FPGA with an arbitrary design with any guarantees of success.

Altera's alternative to Easypath is called Hardcopy [15]. This is a structured ASIC based product with a similar architecture to Altera's FPGA families. The advantage is that a Hardcopy based design will be faster than an FPGA based design while the unit cost will also be lower. Another advantage is that the footprint of the HardCopy device is the same as a regular FPGA. It is therefore easy to migrate a PCB from a Stratix FPGA to a Hardcopy device. The breakeven point where it makes sense to use a Hardcopy device varies from design to design, but it is probably somewhere between 1k units for a large design and 20k units for a small design [16]. The major disadvantage is of course the NRE cost. Since Altera is using a structured ASIC approach, custom masks have to be created for some metal layers. This also means that bugs which could have been fixable with a LUT change in a Xilinx Easypath device will not be fixable here.

Another interesting solution is eASIC's Nextreme product family [17]. This is a cross between a traditional gate array and an SRAM based FPGA. SRAM based lookup-tables are used for the logic while a custom created via layer is used to program the routing. This means the die area of a Nextreme solution when compared to a similar FPGA solution will be smaller, which leads to a reduction in cost. The major advantage of Nextreme when compared to a traditional gate array is that only one via layer has to be customized. In theory there is no NRE cost for low volume production since this via customization can be done by using eBeam technology. For high volume production it is more cost effective to create a custom mask for the via layer though. Another hybrid between an SRAM based FPGA and structured ASICs is Lattice's MACO blocks [18]. This is basically an FPGA which is simply combined with a structured ASIC. The FPGA works just as a regular SRAM based FPGA and the MACO blocks works just as a structured ASIC. Unfortunately, it seems that the significance of gate array will be reduced in the future as FPGAs take over their role [19].

4.4 ASIC and FPGA Tool Flow

A comparison of the design flow for an ASIC and an FPGA (exemplified using Xilinx tools and terminology) can be shown in Table 4.1. (The ASIC design flow has been adapted from [20]). There are some steps that are enclosed in parentheses on the FPGA side. These can be done but are not required. Simulating the post synthesis netlist or post place and route netlist could be done when it is suspected that a tool is present in the synthesis tool or the backend tools. Using a physical synthesis can be done if the tool supports it.

As can be seen, even if all optional parts of the FPGA flow is used, the ASIC flow is considerably more complicated than the FPGA flow and requires a considerable level of expertise fully utilize it.

ASIC flow	**Xilinx Design flow**
Specification	Specification
HDL Code	HDL Code
Behavioral Simulation	Behavioral Simulation
Preliminary RTL Floorplan	
Synthesis	Synthesis
Static Timing Analysis	
Floorplanning	(Floorplanning)
Physical Synthesis	(Physical Synthesis)
Scan Chain Ordering and Routing	
Verification	
Static Timing Analysis	
Gate level full timing simulation	(Post synthesis simulation)
Functional / Formal Verification	
Clock Tree synthesis	
Routing - Optimization	Place and Route
Parasitic Extraction	
Final Verification	
Static Timing Analysis	Static Timing Analysis
Gate Level Full Timing Simulation	(Post place and route simulation)
Functional / Formal Verification	
	Generate Configuration Bitstream
Design Rule Checks	Design Rule Checks
Layout Versus Schematic Checks	
Tapeout	Configure FPGA with bitstream

Table 4.1: Comparison of the ASIC and FPGA design flow (not including power optimizations)

Chapter 5
FPGA Optimizations and ASICs

In this chapter the performance of logic implemented in an FPGA and an ASIC will be compared. At first, this looks like a relatively easy task:

1. Select a design to test
2. Synthesize the design for an FPGA
3. Synthesize the design for an ASIC
4. Compare the performance of these designs

In practice, this is a decidedly non-trivial problem if a completely fair comparison of the capabilities of FPGAs and ASICs is desired. The best way to do this would be to compare an FPGA where the contents of all look-up tables are optimal and the routing is optimal to an ASIC design where the placement, sizing, and routing are all optimal. This is unfortunately an optimization problem of extreme complexity, even for small designs. For everything but the smallest toy design it is unsolvable.

A more realistic comparison would be to use powerful methods such as full custom ASIC design that can produce very efficient designs even though an optimal solution is not guaranteed. At the same time, the same design should be implemented on an FPGA where every LUT has been manually optimized and extensive floorplanning has been used to optimize timing. While this would certainly be an interesting research project, it would also require a large amount of time, making it impractical to do for anything but the smallest designs.

The approach used in this thesis is a more practical approach that intends to highlight a scenario where FPGAs are used for relatively low volume production and where the design is later on ported to an ASIC for high volume production, primarily for cost reduction. Although a lower total cost could be achieved by immediately designing for an ASIC, this is a risky move for several reasons. For example, if the market for a product is uncertain, it can be a good idea to avoid the high NRE costs of an ASIC since it is not certain that these costs can be recouped. Even if there actually is a huge market for a product it can still be a good idea to use an FPGA in the beginning due to its short time to market.

What this part of the thesis intends to highlight is the impact of various FPGA optimized constructs when the design is ported to an ASIC. The intention is that the reader should know what the impact is on an ASIC port when using different kinds of FPGA optimizations.

Finally, it is important to point out that the intention of this chapter is not to recommend a certain FPGA family or vendor. This is why relative performance numbers are used instead of absolute numbers. This chapter is rather intended to show the strengths and weaknesses of modern FPGAs when compared to ASICs. To a lesser degree it will also show that different FPGA families have different strengths which may be of interest when trying to optimize a system for a specific FPGA.

ASIC Porting Hint

Important design hints are marked like this. All guidelines are also present in Appendix A

5.1 Related Work

There is surprisingly little information available on converting designs optimized for FPGAs to ASICs. A brief introduction is given in for example [21], but few, if any, decent in-depth guides are publicly available. There is much more material available on how to port an ASIC design to an FPGA however [22] [23] [24]. These resources may still be be of interest when creating an FPGA design that will later be ported to an ASIC though. Finally, there are also a number of guides available that discusses how to port a design to a Structured ASIC [25] [26] [27].

One noteworthy publication that discusses the performance difference of ASICs and FPGAs is [9]. In this publication the area, power, and performance of over 20 designs are compared. However, the designs in this study were not optimized for a specific FPGA or ASIC process [28].

5.2 ASIC Port Method

The method investigated in this thesis for the ASIC ports is based on porting FPGA designs directly to an ASIC with a minimum of effort. To support RTL code with FPGA primitives instantiated, a small compatibility library has been written. This library has synthesizable models of FPGA primitives such as LUTs, flip-flops and even a limited version of the DSP48 block. This means that it is easy to retarget even an extremely FPGA optimized design to an ASIC. The primary advantage of this kind of porting method is that the time spent in verification should be minimized due to the minimal amount of logic changes necessary. (Assuming of course that the verification for the original FPGA product was thorough.)

Unless otherwise noted in the text, no floorplanning was done and the designs were synthesized without any hierarchy. This kind of low effort porting is compatible with the scenario outlined above, where the primary reason for an ASIC port is cost reduction instead of performance.

The toolchain used for the ASIC ports in this thesis is based on Synopsys Design Compiler (Version A-2007.12-SP5-1) and Cadence SOC Encounter (Version 05.20-s230_1). All performance numbers quoted in this thesis are based on the reports from the static timing analysis. Unless otherwise noted, the area for power rings and I/O pads is not included in any figures.

It could also be noted that this compatibility method could be used to port the synthesized FPGA netlist as well. This would leave stronger guarantees that the ASIC will have the same functionality as the FPGA (for example, if don't cares have been used incorrectly in the source code the behavior of the design could differ depending on the optimizations performed by the synthesis tool). Unfortunately this is probably not possible to do without violating the end user license agreement of the FPGA design tools so this will not be investigated further in this thesis.

5.3 Finding F_{max} for ASIC Designs

As has already been discussed in Section 3.2, there are many things that impact the maximum performance of an FPGA design. Most of these are also valid for ASIC designs. One of the largest differences when trying to find the maximum frequency of an FPGA design is that there are many ways to improve the performance of an ASIC design by trading area for frequency. The synthesis tool will basically use a slow but small unit when the frequency requirements are low and a large but fast unit when a higher maximum frequency is requested.

For most designs such large variations are not common as there are typically large parts of a design that do not require any extensive optimizations to meet timing. Only the critical paths have to be optimized by using expensive structures like logic duplication, advanced adder schemes, etc. As an experiment, the OpenRisc processor with 8 KiB data cache and 8 KiB instruction cache was synthesized to a 130nm process. When optimized for area, the processor had an area of 2.16 mm^2 and a maximum frequency of 36 MHz. When optimized for speed, the area was 2.20 mm^2 and the maximum frequency 178 MHz. In this case the choice of speed or area did not make a large difference in the area but a huge difference in the speed.

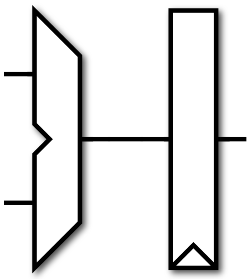
While it is possible to trade area for frequency in an FPGA as well, it is seldom possible to do so for primitive constructs like adders (under the assumption that reasonable sized adders are used).

5.4 Relative Cost Metrics

In this chapter, the relative cost of various design elements in terms of area and frequency are mentioned. The intention is that this chapter will show whether a certain construct is a good idea to use in a certain FPGA. Another important factor that will be discussed in this chapter is if it is a good idea to use a certain architecture in an ASIC.

The relative area and performance for an 32-bit adder was set to 1 and all other costs were derived from this. The ASIC area cost was measured by looking at the size of the block when synthesized, placed, and routed (excluding the power ring). No I/O pads were added to the ASIC designs. Figure 5.1 shows an example of what a report in this thesis can look like. Since the area and F_{max} of a 32-bit adder is used as a reference almost all values are 1 in this table. The exception is an ASIC optimized for area where the values are relative to an ASIC optimized for speed. In Figure 5.1, this means that when optimized for speed in an ASIC, the adder is roughly 9 times faster than the same adder optimized for area.

Determining the area of a circuit in an ASIC is relatively straightforward. Doing the same for an FPGA design is more difficult as the area usage of a certain FPGA design is a multidimensional value. That is, there is no obviously correct way to map the number of used LUTs, flip-flops, DSP blocks, memory blocks, IO pads, and other FPGA primitives to one single area number. One solution is to measure the silicon area of the various components in the FPGA and calculate the active silicon area for a certain design. This is basically the path that is used in [9]. This metric is certainly interesting from an academic point of view, especially when discussing FPGA architectures and how to make FPGAs



Device	Relative performance	Relative area cost
Spartan 3A	1	1
Virtex 4	1	1
Virtex 5	1	1
Cyclone III	1	1
Stratix III	1	1
ASIC (Speed)	1	1
ASIC (Area)	0.11	0.21

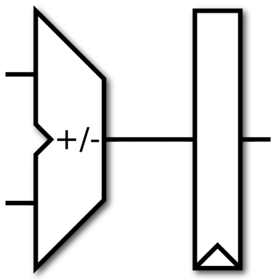
The performance of this circuit is used as a reference for all other performance comparisons in this chapter!

Figure 5.1: 32-bit adder

more silicon efficient.

Knowing the exact silicon area of a LUT or a flip-flop is not going to help a VLSI designer however. Therefore another metric will be used in this thesis. If we assume that the designer considers all components in the FPGA to be of equal monetary value it could be said that the area cost of for example all memory blocks and all slices have an equal area cost. (E.g. if there are 10000 slices and 20 memories in an FPGA the designer will value 1 memory and 500 slices equally. The total area cost of a design with for example 520 slices and 5 memories would in this example be $520 + 5 \cdot 500 = 3020$ pseudo slices.)

While it is unlikely that a designer will consider all components to be exactly equal in value, it is also likely that the designer will use an FPGA where the ratio of memory blocks or DSP blocks to slices is appropriate to his needs. However, the author acknowledges that this metric may be controversial to some readers and has therefore marked all area cost figures that includes converted memory or DSP blocks with a $*$ in the tables in this chapter.



Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

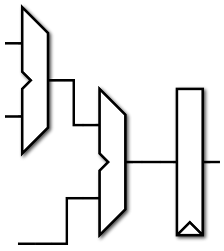
Device	Relative performance	Relative area cost
Spartan 3A	0.97	1
Virtex 4	0.98	1
Virtex 5	0.9	1
Cyclone III	0.82	2
Stratix III	0.81	1
ASIC (Speed)	0.89	1.9
ASIC (Area)	0.13	0.25

Figure 5.2: 32-bit adder/subtracter

5.5 Adders

An adder is a very commonly used component in many designs. It is also one of the components for which FPGAs have been optimized. The carry chains in FPGAs is usually one of the fastest parts of the FPGA if not the fastest. However, in an ASIC, the area cost of an adder can vary wildly depending on the timing constraints as can be seen in Figure 5.1. This was even more pronounced when creating an adder/subtracter as seen in Figure 5.2. Although the performance is still high, the area of the speed optimized circuit is almost twice as large as a plain adder. (Although experiments indicate that the area will quickly decrease if the absolutely highest performance is required).

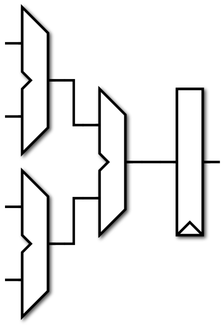
Another area that is interesting to investigate is the case where several adders are used after each other. Figure 5.3 and Figure 5.4 show the relative performance of a 32-bit adder with 3 and 4 operands. It should be noted that the area for the Xilinx families does not quite grow linearly due to some minor optimizations by the synthesis tool. Another interesting fact is that the relative area for a 3 input adder in a Stratix III is the same as the area for a 2 input adder. This is due to the slice architecture of the Stratix III which has a separate adder instead of implementing the adder partly in a LUT (there is also a separate carry chain for the LUT



Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

Device	Relative performance	Relative area cost
Spartan 3A	0.86	1.9
Virtex 4	0.83	1.9
Virtex 5	0.82	1.9
Cyclone III	0.77	2
Stratix III	0.89	1
ASIC (Speed)	0.74	1.3
ASIC (Area)	0.12	0.4

Figure 5.3: 32-bit 3 operand adder



Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

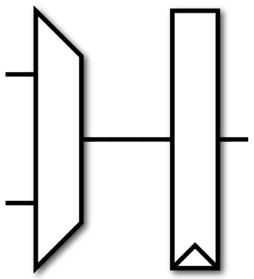
Device	Relative performance	Relative area cost
Spartan 3A	0.62	2.9
Virtex 4	0.58	2.9
Virtex 5	0.58	2.9
Cyclone III	0.77	3
Stratix III	0.82	2
ASIC (Speed)	0.69	1.7
ASIC (Area)	0.1	0.55

Figure 5.4: 32-bit 4 operand adder

that allows two adders to be implemented using the same amount of slices as only one adder). This architecture is also the explanation why the performance of the Stratix III based adders don't drop quite as much as for the other architectures.

ASIC Porting Hint

Adders with more than two inputs are typically more area efficient in ASICs than in FPGAs.

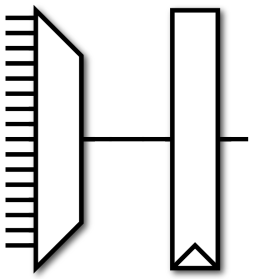


Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

Device	Relative performance	Relative area cost
Spartan 3A	2.3†	1
Virtex 4	2†	1
Virtex 5	1.7†	1
Cyclone III	2.2†	1
Stratix III	2.1†	1
ASIC (Speed)	1.8	0.12
ASIC (Area)	1.8	0.12

† Exceeds F_{max} for clock net.

Figure 5.5: 32-bit 2-to-1 mux



Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

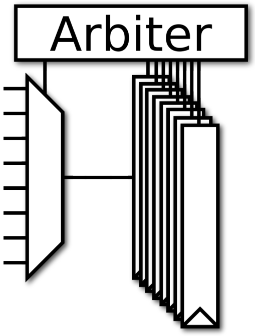
Device	Relative performance	Relative area cost
Spartan 3A	1.5	8
Virtex 4	1.2	8
Virtex 5	0.92	5
Cyclone III	1.3	10
Stratix III	1.7†	5
ASIC (Speed)	0.9	0.57
ASIC (Area)	0.31	0.48

† Exceeds F_{max} for clock net.

Figure 5.6: 32-bit 16-to-1 mux

5.6 Multiplexers

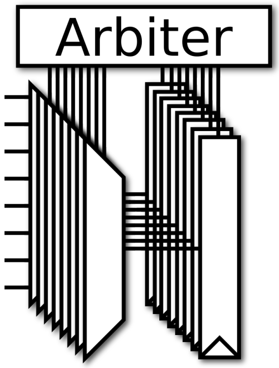
A very important part of many datapaths is the multiplexer (mux). An FPGA based solely on 4 input LUTs can implement a 2^n-to-1 mux using $2^n - 1$ LUTs. However, modern FPGAs usually have some sort of hard-wired muxes in the slices and CLBs to optimize the implementation of muxes. In this case it is possible to implement a 2^n-to-1 mux using only 2^{n-1} LUTs. In the Virtex-4 architecture, muxes of sizes up to 32-to-1 can



Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

Device	Relative performance	Relative area cost
Spartan 3A	0.85	9.4
Virtex 4	0.83	9.5
Virtex 5	0.78	9
Cyclone III	1.1	9.4
Stratix III	1.1	7.3
ASIC (Speed)	0.85	1.5
ASIC (Area)	0.25	1.2

Figure 5.7: 32-bit bus with 8 master ports and 8 slave ports



Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

Device	Relative performance	Relative area cost
Spartan 3A	0.77	74
Virtex 4	0.7	75
Virtex 5	0.6	59
Cyclone III	0.75	80
Stratix III	0.73	61
ASIC (Speed)	0.74	5.5
ASIC (Area)	0.24	3.4

Figure 5.8: 32-bit crossbar with 8 master ports and 8 slave ports (Implemented with muxes)

be optimized in this way.

Figure 5.5 - 5.6 show the relative performance and area of a 2-to-1 and 16-to-1 32-bit multiplexers. Note that the relative area of the ASIC multiplexers when compared to the FPGA multiplexers is very low.

It is clear that muxes are not likely to be a big problem when porting a design to an ASIC. However, it may be possible to improve the design of the ASIC port by inserting extra muxes in the design. A cheap way to do this could be to replace a mux based bus with a full crossbar. If the crossbar is using the same bus protocol as the bus, relatively little would have to be redesigned and reverified. This is shown in Figure 5.7 and Figure 5.8 which illustrates a bus and crossbar with 8 master ports and 8 slave ports. There are other ways that muxes could improve the performance of a design as well, but they are likely to be more complex to implement and verify (e.g. adding more result forwarding to a processor).

ASIC Porting Hint

While multiplexers are very costly in an FPGA, they are quite cheap in an ASIC. Optimizing the mux structure in the FPGA based design will have little impact on an ASIC port.

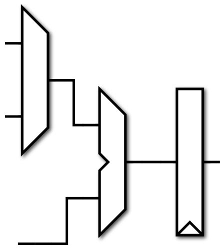
ASIC Porting Hint

When porting an FPGA optimized design to an ASIC it may be possible to increase the performance of the ASIC by adding muxes to strategic locations such as for example by replacing a bus with a crossbar.

5.7 Datapath Structures with Adders and Multiplexers

Just investigating standalone components is not very interesting. What is interesting is to look at a datapath that contains muxes in combination with other elements. The tradeoffs when creating a design with many muxes for an FPGA is not the same as when creating a design for an ASIC. In [29], Paul Metzgen describes the relative tradeoffs off different FPGA building blocks and comes to the conclusion that *"The Key to Optimizing Designs for an FPGA . . . is to Optimize the Multiplexers"*.

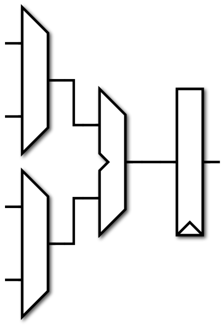
A simple example of how muxes can be optimized is shown in Figure 5.9. In this figure, a 2-to-1 mux is used for one of the operands to a 32 bit adder. A naive area cost estimate for this construct in an FPGA would take the cost of a 32 bit 2-to-1 mux and add it to the cost of a 32 bit adder. However, this doesn't take into account that the LUTs are not fully utilized in a 32-bit adder in all contemporary Xilinx devices. This means that it is possible to merge the mux into the LUTs used for the adder yielding the same LUT cost as for a regular 32-bit adder. The per-



Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

Device	Relative performance	Relative area cost
Spartan 3A	0.99	1
Virtex 4	0.97	1
Virtex 5	0.91	1
Cyclone III	0.81	2
Stratix III	0.75	1
ASIC (Speed)	0.69	1.1
ASIC (Area)	0.14	0.25

Figure 5.9: 32-bit adder with 2-to-1 mux on one operand



Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

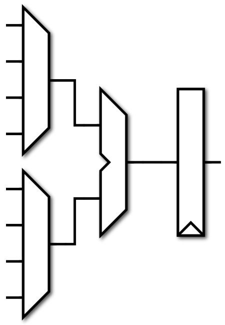
Device	Relative performance	Relative area cost
Spartan 3A	0.85	2
Virtex 4	0.83	2
Virtex 5	0.8	2
Cyclone III	0.79	3
Stratix III	0.79	2
ASIC (Speed)	0.67	1.2
ASIC (Area)	0.14	0.3

Figure 5.10: 32-bit adder with 2-to-1 mux on both operands

formance of this kind of solution will also be very good, almost the same as for just the 32-bit adder.

However, if two 2-to-1 muxes should be used, one for each operand, the area cost will double since it is not possible to put that mux into the same LUT as seen in Figure 5.10. Finally, if large muxes like 4-to-1 muxes are used for both operands of the adder, the area cost of the FPGA designs will go up significantly and the performance will drop considerably as can be seen in Figure 5.11

There are two final examples that are interesting to mention. The first is a rather special case which may be good to know about; the case where



Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

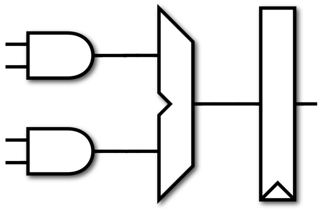
Device	Relative performance	Relative area cost
Spartan 3A	0.77	5
Virtex 4	0.69	5
Virtex 5	0.71	3
Cyclone III	0.77	5
Stratix III	0.88	3
ASIC (Speed)	0.63	1.8
ASIC (Area)	0.12	0.4

Figure 5.11: 32-bit adder with 4-to-1 mux on both operands

a two input and gate is used as the input for both adder operands as can be seen in Figure 5.12. Note that this has the same area cost in Xilinx FPGAs as a plain adder. This is because the and MULT_AND primitive can be used for one of the and gates. A similar structure was also used in the ALU in Microblaze where the result can be either *OpB* + *OpA*, *OpB* - *OpA*, *OpB*, or *OpA* [30]. Without using the MULT_AND primitive it would not be possible to get the last operation here (*OpA*)). Stratix III also allows this structure to be implemented using the same amount of resources as a plain adder because it has a dedicated adder inside each slice and doesn't need to use LUTs to create the adder itself. The LUTs in the slice can therefore be used separately to create these (and other) functions.

The other case is when an adder/subtracter is used instead of just an adder. Figure 5.13 and Figure 5.14 shows the properties of an adder/-subtracter without and with a 2-to-1 input mux on one of the inputs. The plain adder/subtracter can be implemented with the same area as an adder in all FPGAs. However, it is not possible to combine an adder/-subtracter with a mux in most FPGAs.

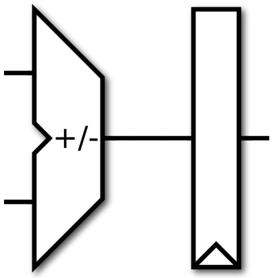
Overall, it is hard to say anything definitive about the cost of these components in an ASIC since there is a large gap between the area and frequency of the slowest and fastest variation. What can be seen in these



Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

Device	Relative performance	Relative area cost
Spartan 3A	0.99	1
Virtex 4	0.98	1
Virtex 5	0.78	1
Cyclone III	0.84	3
Stratix III	0.85	1
ASIC (Speed)	0.89	0.89
ASIC (Area)	0.11	0.27

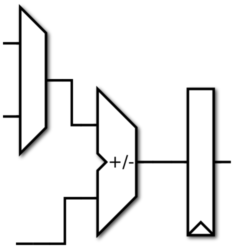
Figure 5.12: 32-bit adder with 2-input bitwise and on both operands



Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

Device	Relative performance	Relative area cost
Spartan 3A	1	1
Virtex 4	0.98	1
Virtex 5	0.89	1
Cyclone III	0.83	2
Stratix III	0.82	1
ASIC (Speed)	0.86	1.7
ASIC (Area)	0.14	0.25

Figure 5.13: 32-bit add/sub



Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

Device	Relative performance	Relative area cost
Spartan 3A	0.83	2
Virtex 4	0.84	2
Virtex 5	0.83	1
Cyclone III	0.82	2
Stratix III	0.89	2
ASIC (Speed)	0.62	2.4
ASIC (Area)	0.14	0.31

Note that it is important that this is implemented as a +/- b, where b is the output of the mux.

Figure 5.14: 32-bit add/sub with 2-to-1 mux for one operand

figures, especially the area optimized figures is that this particular kind of FPGA optimization is not very helpful for the ASIC performance or ASIC area. The area when optimized for area is (not surprisingly) about the same as when combining the area of the individual components.

ASIC Porting Hint

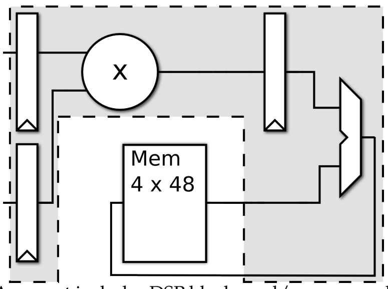
While a lot of performance and area can be gained in an FPGA by merging as much functionality into one LUT as possible, this will typically not decrease the area cost or increase the performance of an ASIC port.

5.8 Multipliers

The most common method for multiplication in FPGAs today is to use one of the built in multiplier blocks that are present in almost all modern FPGAs. While some FPGAs such as the Virtex-II and Spartan-3 have a standalone multiplier as a separate block, other FPGAs also integrate accumulators into the same block. The later are commonly called DSP blocks.

In many cases there are optional pipeline stages built into these multipliers and the maximum performance can only be reached if these pipeline stages are utilized. In the Spartan 3A for example, there are optional pipeline stages before and after the combinational logic of the multiplier whereas a DSP48 block in a Virtex-4 can contain up to 4 pipeline registers. (Although the fourth register will not increase the performance as it is only present to easily allow for construction of large multipliers without having to store intermediate results in flip-flops in the FPGA fabric [31]).

As an example of how to optimize a circuit for the DSP blocks we will study a typical MAC unit in a simple DSP processor when implemented on a Virtex-4 or 5. The MAC unit contains a 16×16 multiplier and four 48-bit accumulator registers. This could be implemented as seen in Figure 5.15. Unfortunately, the FPGA performance is not very good in this case since it is not possible to use the built-in accumulation register of the DSP48 block.

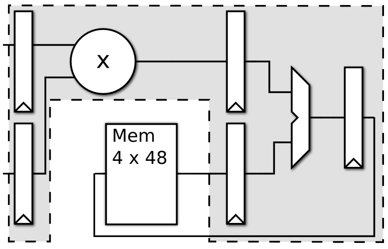


Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

Device	Relative performance	Relative area cost
Virtex 4	0.45	30*
Virtex 5	0.41	35*
ASIC (Speed)	0.42	5.3
ASIC (Area)	0.084	2.4

∗ Area cost includes DSP blocks and/or memory blocks as described in Section 5.4. *Parts in grey are mapped to the DSP48 block.*

Figure 5.15: MAC unit with 4 accumulator registers mapped to DSP48 block



Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

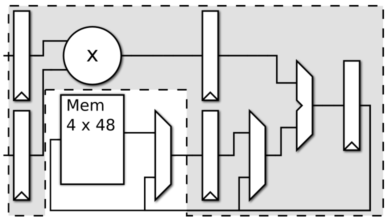
Device	Relative performance	Relative area cost
Virtex 4	1.2	31*
Virtex 5	0.82	35*
ASIC (Speed)	0.49	5.4
ASIC (Area)	0.11	2.7

∗ Area cost includes DSP blocks and/or memory blocks as described in Section 5.4. *Parts in grey are mapped to the DSP48 block.*

Figure 5.16: MAC unit with 4 accumulator registers mapped to DSP48 block with pipelining

The adder could also be pipelined, as shown in Figure 5.16. This allows a high performance to be reached, but the circuit is no longer identical to the circuit in Figure 5.15. In this case it is no longer possible to perform continuous accumulation to the same accumulation register due to the data dependency problems introduced by the pipelined adder.

Finally, a circuit that still allows for high speed operation while having the same capabilities as the circuit in Figure 5.15 is shown in Figure 5.17. In this circuit result forwarding is used to bypass the register file. The performance of this circuit is much higher than the naive implementation and clearly demonstrates how important it is to make sure



Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

Device	Relative performance	Relative area cost
Virtex 4	0.92	33*
Virtex 5	0.72	36*
ASIC (Speed)	0.49	4.9
ASIC (Area)	0.093	2.7

$*$ Area cost includes DSP blocks and/or memory blocks as described in Section 5.4. *Parts in grey are mapped to the DSP48 block.*

Figure 5.17: MAC unit with 4 accumulator registers mapped to DSP48 block with pipelining and forwarding

that the DSP blocks are utilized efficiently in an FPGA.

However, note that the performance and area cost in the ASIC doesn't differ very much from case to case. Since the multiplier structures are not fixed it is simply not necessary to adhere to a certain coding style. An unoptimized multiplier structure will therefore port well to an ASIC. However, if the multipliers have been optimized for the FPGA, an ASIC port will probably have performance problems due to the large relative performance difference between adders and multipliers in ASICs. This could mean that the datapaths that contain multipliers may have to be redesigned when ported to an ASIC.

It should also be noted that in all of these cases the RTL code was first written using generic addition/multiplication. The synthesis tool did not infer the desired logic however, so it was necessary to rewrite the RTL code to instantiate a DSP48 in the desired configuration. The Virtex-4 and Virtex-5 values are based on this rewritten design whereas the ASIC values are based on the initial generic design. The performance of the final two designs in the FPGAs were significantly improved by this rewrite.

ASIC
Porting
Hint

If a design is specifically optimized for the DSP blocks in an FPGA the ASIC port is likely to have performance problems. The datapath with multipliers may have to be completely rewritten to correct this.

There are a few alternative ways to design multipliers inside FPGAs. One way is simply to create a multiplier using the FPGA fabric itself. This approach is resource intensive and requires a decent amount of pipelining if high throughput is required. This is probably not a good idea unless targeting devices without any builtin multipliers. (While most modern FPGAs does have dedicated multipliers there are a few exceptions, like the LatticeSC.)

Another method that can be used when high performance is required but a hardware multiplier is not available is to use a look-up table based approach. While a direct look-up table for the function $f(a, b) = a \cdot b$ is only useful for very small bit-widths, a hybrid approach can be used where the look-up table implements the function $g(x) = x^2/4$. In this case $a \cdot b$ is calculated as $g(a + b) - g(a - b)$. This allows multiplication of modest bit-widths such as 8 bits to be efficiently calculated even in FPGAs without any support for hardware multiplication assuming enough memory resources can be dedicated to the look-up table.

A third method that can be used is a bit-serial approach. While commonly used in designs for old FPGAs it is still a valid design method for situations where low resources usage is more important than high throughput and low latency.

This is by no means an exhaustive survey of multiplication algorithms, there is a wide variety of alternative methods such as for example methods based on logarithmic number systems. There are also a wide variety of ways to optimize a multiplier if one of the operands is constant (see for example [32] or [33]).

5.9 Memories

When synthesizing a design for ASIC that contains memories it is very important that optimized memory blocks are used for large memories. As an experiment, an 8 KiB memory block was synthesized with standard cells for an ASIC process (and optimized for area). The area was almost 10 times as large as a custom made memory block of the same size.

There are a few disadvantages to using specialized memories as well. The design cannot be ported to a new process by merely resynthesizing the HDL code. Simulation is also more difficult because special simulation models have to be used for the memories. These disadvantages mean that it might not make sense to use memory compilers for small memories as the increase in design and verification time will outweigh the area/speed advantage of these memories.

A common way to handle customized memory blocks is to write wrappers around the memories so that it is possible to port the design to a new technology by changing the wrappers instead of having to rewrite the HDL code of the design itself.

ASIC Porting Hint

Create wrapper modules for memory modules so that only the wrappers have to be changed when porting the design to a new technology.

5.9.1 Dual Port Memories

Another very important factor to take into account when designing for both FPGAs and ASICs is the number of ports on the memory. The RAM blocks in an FPGA usually has two independent read/write ports, which means that many FPGA designs use more than one memory port even though similar functionality could be achieved using only one memory port.

It is hard to find publicly available information about the performance and size of custom memory blocks. One datasheet which is available without any restrictions shows that a 1 KiB dual port memory with 8-bit width is around 63% larger than a single port memory of the same size [34]. This datasheet is for a rather old process though (0.35 μm) and the author has seen dual port memories for newer processes that are more than twice as large as a single port memory.

Regardless of the exact proportions, it is certainly more expensive to use a dual port memory than a single port memory. There are many cases where a dual port memory is natural to use but not strictly necessary. A good example of this is a synchronous FIFO. While it is convenient with a dual port memory here, it is not strictly necessary as it is possible to use for example two single port memories and design the FIFO so that one memory is read while the other is written and vice versa. This is described in for example [35].

Finally, there are examples where it is not easily possible to avoid the use of dual port memories. In such cases the cost of redesigning the system to use single port memories has to be weighed against the area savings such a redesign will produce.

ASIC Porting Hint

Avoid large dual port memories if it is possible to do so without expensive redesigns.

5.9.2 Multiport Memories

There are some cases where more than two ports are required on a memory. An example of this would be a register file in a typical RISC processor. Such a register file usually has two read ports and one write port. In for example a Virtex4 this is implemented as shown in Figure 5.18. This memory is implemented by using two dual port distributed RAM memories where each memory has one read port and one write port. When writing to this memory block, the same value is written into both distributed RAMs. This gives the illusion of having a memory with two read ports and one write port even though in reality there are actually two separate memories.

The area cost of a few different register file memories with different number of ports are shown in Table 5.1. While the area grows relatively slowly in the ASIC, the area for the FPGA based register grows extremely fast when going from one to two write ports. The explanation for this is that the synthesis tool is no longer able to use the distributed memories. The relatively high area costs for the Cyclone III based architectures

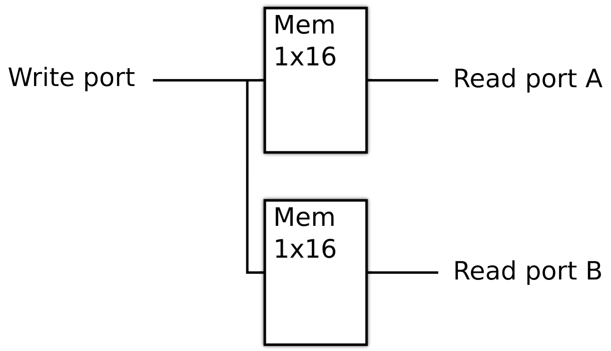


Figure 5.18: Small register file memory with multiple read ports implemented using duplication

Area cost values are relative to the values for the 32-bit adder in Figure 5.1.

Ports		Spartan 3A	Virtex 4	Virtex 5	Cyclone III	Stratix III	ASIC optimized for	
Read	Write						Area	Speed
1	1	0.50	0.50	0.25	9.8†	2.1	0.65	0.67
2	1	1.0	1.0	0.50	20†	2.1	0.78	0.83
2	2	9.5	9.5	13	11	4.5	0.95	1.1
4	2	14	14	15	16	6.1	1.2	1.4

† Area cost includes DSP blocks and/or memory blocks as described in Section 5.4.

Table 5.1: Relative area cost of an 8-bit 16 entry register file memory with different number of ports

with one write port are caused by the use ofM9K block RAMs as the Cyclone III does not have distributed memory.

ASIC Porting Hint

If many small register file memories with only one write port and few read-ports are used in a design, the area cost for an ASIC port will be relatively high compared to the area cost of the FPGA version. On the other hand, if more than one write port is required, the ASIC port will probably be much more area efficient.

Large multiport memories are probably not going to be used in many FPGA designs due to their extreme area cost. While such a memory should certainly be more area efficient in an ASIC than an FPGA, the area cost will be extremely high anyway. As an example, in a die-photo of an SPU in the cell processor the register file is roughly the same size as one of the large SRAM blocks [36]. However, the RAM block is a single port memory of 64 KiB whereas the register file has 6 read ports and 2 write ports and contains only 128×128 bits. The single port memory stores around 30 times as much data as the register file while using the same amount of die area!

There are also ways to fake a multiport memory that can be efficiently used in both ASICs and FPGAs. One way is to use a normal memory that has a clock signal which is running at a multiple of the regular system clock frequency [37]. Another way is to use some sort of caching scheme, although this implies that the memory is no longer a true multiport memory.

5.9.3 Read-Only Memories

Another type of memory which hasn't been considered yet is a read-only memory. Whether a memory is read-only or not is seldom an issue in FPGAs (although some Altera FPGAs will implement small ROMs more efficient than small RAMs), but the impact of using a ROM instead of RAM in an ASIC will be substantial. In for example the ATC35 process, a 1024×16 bit RAM is roughly 7 times larger than a ROM of the same size [34].

The disadvantage of using read-only-memory is of course the lost flexibility. Changes to the contents of the memory will now require an expensive mask change (although ROMs are typically designed so that only one mask has to be changed). It is therefore probably not a good idea to use a ROM-only solution for memories that are likely to contain bugs such as firmware or microcode memories. A compromise solution could be to wrap the ROM in a block that allows a small number of rows in the memory to be modified at runtime, such as the solution described by AMD in [38].

ASIC Porting Hint

Design Guideline: If some of the memories can be created with a ROM-compiler, the area savings in an ASIC port will be substantial.

5.9.4 Memory Initialization

An FPGA designer has the luxury of being able to initialize the RAM blocks in the design at configuration time. This means that it is common to use part of a RAM for read-only data such as for example bootloading and initialization code whereas the rest of the RAM can be used for runtime data.

In an ASIC, the contents of RAM memories are instead usually undefined at power-up. In the example above, the initial firmware/bootloading would either have to be part of a ROM or loaded into the RAM through other means at power-up.

ASIC Porting Hint

Avoid relying on initialization of RAM memories at configuration time in the FPGA version of a design.

5.9.5 Other Memory Issues

There are also some specialized memories that can be useful in some cases. One of these is the content addressable memory (CAM). Small CAM memories can be implemented in an FPGA although it is fairly expensive to do so as seen in Table 5.2. (It should be noted that significant performance improvements can be made by utilizing specialized CAM memories instead of the standard cell based approach used here.)

Even more expensive is the ternary CAM that is often used in routers. While it is certainly feasible to use small CAM memories in an FPGA, it is probably a good idea to avoid large CAM memories. In many cases, a CAM is not strictly required and can be replaced with other algorithms that are more area efficient in an FPGA such as algorithms based on searching for the data in a tree-like data structure.

Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

Device	Relative performance	Relative area cost
Spartan 3A	0.72	8.2
Virtex 4	0.64	8.2
Virtex 5	0.51	8.2
Cyclone III	0.85	11
Stratix III	1	8.2
ASIC (Speed)	0.92	2.4
ASIC (Area)	0.19	1.5

Table 5.2: 16 entry CAM memory with 16 bit wide data

ASIC Porting Hint

When porting a design to an ASIC, consider if specialized memories like CAM memories can give significant area savings or performance boosts.

Another issue is designs that are heavy users of the SRL16 primitive in Xilinx FPGAs. These small 16-bit shift registers are commonly used in delay lines or small FIFOs. As can be seen in Section 10.8, this can lead to a huge area increase in an ASIC when compared against a design that doesn't use SRL16 primitives.

There are a few other considerations to take into account when using memories in an ASIC process that are not necessary to take into account in an FPGA. For example it might be possible to generate memories that are optimized for speed or power consumption. For large memories it might be a good idea to use a memory with redundancy so that a tiny fabrication error in the memory will not cause it to fail. In an ASIC memory it is often also possible to use a write mask, which means that it is possible to write to only certain bits in a memory word without having to do a read-modify-write operation.

ASIC Porting Hint

Consider if special memory options that are unavailable in FPGAs, like write masks, can improve the design in any way.

5.10 Manually Instantiating FPGA Primitives

In some cases it might be necessary to manually instantiate adders and subtracters in an FPGA design. One reason could be that floorplanning requires a deterministic name, another reason could be that the synthesis tool is not able to infer a desired configuration automatically (although modern synthesis tools are usually pretty good at this). Regardless of the reason, such a design cannot be directly synthesized to an ASIC due to the dependencies on FPGA elements.

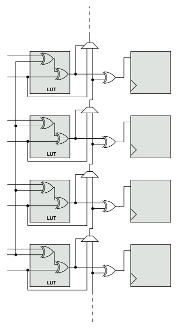
By using a compatibility library it is still possible to synthesize the design. The drawback to this approach is that the synthesizer will only see a bunch of combinational logic instead of a plus sign in the source code. This will mean that optimized adder structures such as the ones available in Synopsys DesignWare library will not be used.

An example of the performance and area cost of a 32-bit adder/subtracter implemented through instantiation of Xilinx primitives is shown in Figure 5.19. The performance of the ASIC port is not quite as good as the performance of the adder/subtracter shown in Figure 5.2, but the area is also lower. The synthesis tool is obviously able to do a pretty good job of optimizing this structure even though it doesn't know beforehand that it is an adder/subtracter.

A word of warning though, it is extremely important that the module that contains FPGA primitives is instantiated without hierarchy. Otherwise the synthesis tool will not be able to perform such optimizations. Experiments indicate that if hierarchy is disabled, the performance of an instantiated adder will be roughly the same as that of an area optimized adder.

ASIC Porting Hint

It is possible to port a design with instantiated FPGA primitives using a small compatibility library. For adders and subtracters, the performance will be adequate unless they are a part of the critical path in the ASIC. If this approach is used it is imperative that the modules with instantiated FPGA primitives are flattened during synthesis and before the optimization phase!



Both F_{max} and area cost values are relative to the values for the 32-bit adder in Figure 5.1.

Device	Relative performance	Relative area cost
Spartan 3A	0.97	1
Virtex 4	0.98	1
Virtex 5	0.9	1
ASIC (Speed)	0.76	1.4
ASIC (Area)	0.15	0.28

All 32 LUTs/flip-flops not shown.

Figure 5.19: 32-bit adder/subtracter using FPGA primitives

5.11 Manual Floorplanning and Routing

Manual floorplanning and manual routing is a labor intensive process that allows a designer to precisely control where certain parts of a design will end up in the FPGA. While this process can improve the timing of an FPGA design it should in theory have no impact on an ASIC. However, it is likely that other optimizations are necessary to efficiently support manual floorplanning, such as manually instantiating memories, LUTs and flip-flops to make sure that the names of these primitives in the netlist are not changed. By doing these changes to the source code it is possible that the synthesis output might also vary and the result of these changes must be evaluated separately.

ASIC
Porting
Hint

Manual floorplanning of an FPGA design will not have any impact on an ASIC port unless the design is modified to simplify floorplanning in the FPGA.

Area cost values are relative to the values for the 32-bit adder in Figure 5.1.

Pipeline Stages	Spartan 3A	Virtex 4	Virtex 5	Cyclone III	Stratix III	ASIC optimized for Area	ASIC optimized for Speed
1	260†	380†	450†	57	54	5.5	20
2	260†	380†	450†	58	77	6.1	13
3	260†	380†	450†	61	79	6.8	14
4	260†	380†	450†	60	78	7.0	13

† Area cost includes DSP blocks and/or memory blocks as described in Section 5.4.

(The constant coefficient multipliers are implemented in the fabric in the Cyclone III and Stratix III devices and don't use DSP blocks.)

Table 5.3: Relative area of an eight point 1D DCT pipeline

F_{max} values are relative to the values for the 32-bit adder in Figure 5.1.

Pipeline Stages	Spartan 3A	Virtex 4	Virtex 5	Cyclone III	Stratix III	ASIC optimized for Area	ASIC optimized for Speed
1	0.25	0.17	0.16	0.39	0.39	0.073	0.30
2	0.33	0.21	0.21	0.47	0.52	0.079	0.35
3	0.38	0.28	0.27	0.59	0.58	0.10	0.41
4	0.37	0.28	0.27	0.73	0.67	0.10	0.40

Table 5.4: Relative performance of an eight point 1D DCT pipeline

5.12 Pipelining

Pipelining is an important technique to improve the performance of a digital design. In FPGAs this can be done in an area efficient manner since there are usually a large amount of flip-flops available in an FPGA. When porting a design to an ASIC it is seldom a drawback to have a large number of pipeline stages as well if performance is the number one priority. Unfortunately flip-flops can be fairly expensive in terms of area in an ASIC which means that it may be a good idea to rewrite the design so that fewer pipeline stages are needed. However, it is not always true that an increased number of pipeline stages will always increase the area. Consider for example the case outlined in Table 5.3 and Table 5.4 where a pipeline for an eight point 1D DCT was implemented using different number of pipeline stages. (This module was not optimized for FPGAs in any way.) When going from one to two pipeline stages the maximum frequency increased by almost 15% even though the area of the new design is only 65% as large as the area of the design with a shorter pipeline. This is probably caused by area inefficient optimizations used by the synthesis tool as it is struggling to reach an unreachable performance goal. When adding extra pipeline stages after that, the performance increases until a plateau is reached at the third pipeline stage.

This example shows that pipelining a datapath is not guaranteed to increase the area although it is not uncommon that the area is increased, especially if the datapath is not a part of a critical path such as when adding delay registers to synchronize the values in one datapath with the values in another datapath.

ASIC Porting Hint

While pipelining an FPGA design will certainly not hurt the maximum frequency of an ASIC, the area of the ASIC will often be slightly larger than necessary, especially if the pipeline is not a part of the critical path in the ASIC. Designs that contains huge number of delay registers will be especially vulnerable to such area inefficiency.

5.13 Summary

There are a wide variety of issues that need to be taken into account when porting a design from an FPGA to an ASIC. This is especially true if the design has been optimized for a certain FPGA from the beginning without any thoughts of an ASIC port.

When porting a design the most tricky areas are likely to be the architecture around the memories and multipliers. If these have been optimized for a specific FPGAs an ASIC port is likely to be suboptimal. The other FPGA optimizations are not going to harm an ASIC port and are in some cases even beneficial.

There are also many other issues that have not been discussed here, like I/O, design for test, and power dissipation which it is important that the designer takes into account as well.

Part II

Data Paths and Processors

Chapter 6

An FPGA Friendly Processor for Audio Decoding

Abstract: *In this chapter a DSP processor specialized for audio decoding will be described. While not specifically optimized for FPGAs, the processor is still able to achieve a clock frequency of 201 MHz in a Virtex-4 and the performance of the processor when decoding an MP3 bitstream is comparable to a highly optimized commercial MP3 decoding library.*

In early FPGAs soft processors were seldom used due to their large area and the high cost of an FPGA compared to a microprocessor. Nowadays the situation is decidedly different and soft processors are regularly used in anything from the smallest to the largest FPGAs. Although most people are probably using the soft processor cores that are available from their FPGA vendor there are also a huge amount of processors available at for example OpenCores [39] (95 at the time of writing). There is certainly no lack of choice in the soft processor market.

6.1 Why Develop Yet Another FPGA Based Processor?

The main reason to create yet another soft core processor is because the majority of the processors that are available are not really optimized for FPGAs. While some processors on OpenCores probably have a decent performance in an FPGA, none are likely to match the performance of the FPGA optimized processors that are available from the FPGA vendors. Another issue is that there does not seem to exist a credible DSP processor that has been optimized for FPGAs.

At first, the idea of using a DSP processor for signal processing in an FPGA does not make sense. If the DSP algorithm is computationally intensive it is probably a much better idea to create custom hardware in the FPGA instead of using a processor for it. On the other hand many applications employs many different algorithms that are not indidually very computationally intensive.

Consider for example a video camera with a high definition image sensor and a microphone for audio recording. Two tasks that have to be carried out in this device is motion estimation for video and MDCT processing for audio. These are both signal processing tasks that can easily be accelerated in hardware. The difference is that over the course of one second, the number of calculations needed for the MDCT is negligible when compared to the number of calculations required for motion estimation. Therefore it makes sense to create a hardware based accelerator for the motion estimation whereas the MDCT for audio decoding could be done in software.

A processor with good digital signal processing performance is therefore a good idea for these kinds of tasks that don't require an accelerator but are still computationally intensive for a general purpose processor. And this processor also needs to be optimized for FPGAs as an ASIC based DSP is unlikely to have a high performance in an FPGA.

6.2 An Example of an FPGA Friendly Processor

The main idea behind the xi[1] processor described in this chapter is that it should use floating point arithmetic for most of the calculations that have to be performed when decoding an MP3 bitstream. This allows a high dynamic range to be achieved using fewer bits than fixed point arithmetic. This means that the amount of memory required for intermediate data is significantly reduced. (Dynamic range is very important for audio application and if high dynamic range is available the precision doesn't have to be extremely high, see [40] for more details.)

The architecture of the processor was not specifically optimized for FPGAs. However, one of the goals was that the architecture should be suitable both for FPGA and ASIC implementation to simplify prototyping.

The processor has also been manufactured in a 180 nm ASIC process (see photo on the back cover of the thesis), although the evaluation of this MPW chip is not yet complete and will be published in a later publication.

6.2.1 Processor Architecture

The processor core is a fairly standard pipelined RISC processor. There are 5 pipeline stages for integer operations and 8 pipeline stages for floating point operations. To increase the performance of MP3 decoding the processor has support for circular buffers and floating point MAC for filter acceleration and bitstream reading instructions for Huffman decoding acceleration.

There are 16 general purpose registers. Each register can hold either 16 bit integer data or 23 bit floating point data (16 bit mantissa, 6 bit exponent and one sign bit). The program memory is 24 bit wide and can contain up to 8192 instructions. The constant memory is 23 bits wide and

[1]The designers selected this name due to heavy indoctrination by the math department (no math lecture seems to be complete unless at least one ξ has been written on the whiteboard…)

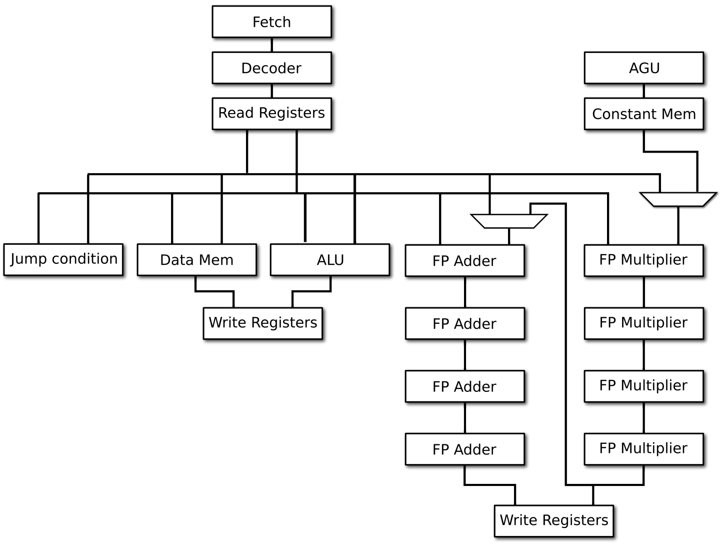


Figure 6.1: Pipeline of the xi audio decoding processor

1024 words large. It is primarily used for floating point constants. The data memory is 16 bits wide and 8192 words can be stored in it. Totally the chip has 343 kilobit memory.

It is not possible to read or store a 23 bit floating point value directly to the data memory due to the difference in size. This is handled by converting floating point values into a 16 bit format before they are stored to memory. The different word lengths were chosen by profiling an MP3 decoder, which was modified to support a number of different floating point formats.

6.2.2 Pipeline

The architecture of the processor is shown in Figure 6.1. As can be seen, the processor has a relatively long pipeline, especially the floating point units. Every pipeline step also does relatively little work, which means that it should be relatively simple for a synthesis tool to map this architecture to an FPGA. (Another reason for the long pipeline was to ensure that the performance in an ASIC would still be adequate if the power supply voltage was reduced in order to save power.)

There is no result forwarding used in this architecture, partly because the muxes required to implement this would increase the area, especially in FPGAs. Omitting result forwarding also simplified the verification. Another reason was that it was not clear whether the clock frequency of the processor would be high enough in an FPGA to decode MP3 bit-streams in real time if forwarding was available. Another feature that was not included due to the verification cost was hazard detection (it is up to the programmer to avoid hazards by careful scheduling of the code or by inserting NOP instructions).

6.2.3 Register File

In the processor there are a few special registers that control for example the address generator for the constant memory. It is tempting to place these registers in the regular register file. The advantage is that no special instructions are required to access these registers and it is also possible to manipulate these registers arbitrarily using most of the instructions in the instruction set. However, the disadvantage is that it is no longer possible to use the distributed memory available in for example Xilinx FPGAs to implement the register file. Because of this the decision was taken to implement special registers separately and use special instructions to access these registers.

6.2.4 Performance and Area

The performance of this processor is summarized in Table 6.1. This version has a total of 351 kbit memory. The most critical path is in the address generator to the data memory in the FPGA version and inside the multiplier for the ASIC version.

Table 6.2 shows a comparison of our MP3 decoder when compared to the commercial Spirit MP3 Decoder [41] on a variety of platforms. The numbers for peak MIPS is based on decoding a 48kHz 320 kbit/s MP3

Virtex 4 (speedgrade 12)	LUTs	2370
	Flip-flops	1048
	RAMB16	22
	DSP48	1
	F_{max}	201 MHz
130nm ASIC (Speed optimized)	Area	2.3 mm^2
	F_{max}	396 MHz

Table 6.1: Performance and area of audio processor

bitstream. In the case of the LiU firmware the value of 20 MIPS is reached by constructing a synthetic bitstream where all options and values in the bitstream have been chosen to trigger the worst case behavior of the decoder.

As can be seen, the performance of the xi MP3 decoder using our own firmware is efficient when compared to the Spirit MP3 Decoder [41] on single issue processors such as the ARM9. As for the memory usage, the use of floating point constants initially seems to drastically reduce the amount of constant memory used. However, this is not necessarily true as the Huffman decoding is implemented entirely as a program in our decoder. (It is not clear how the Spirit MP3 decoder stores the Huffman table but it is likely that it is stored in constant memory.) The program memory size for our decoder is not so impressive although optimizing the size of the program memory was never a goal for this particular project. There is a lot that could be improved here, see Section 6.2.6 for more information. Also note that our decoder is only able to decode MPEG1 - layer III bitstreams. A fully compliant decoder should also be able to decoder layer I and II. This would take some additional program memory.

When comparing the performance of our processor with the VLIW based AudioDE processor it is also clear that an impressive performance improvement can be gained by increasing the parallelism. There is however a rather steep increase in terms of memory area and probably core

Processor	Firmware	Peak/Average MIPS	Memory cost
xi	Custom† (limited accuracy)	20/14	35.1kB (20.5 kB program, 2.6 kB constants, 12 kB data)
xi	Custom† (full precision)	20/14‡	37.5kB (20.5 kB program, 2.7 kB constants, 14.3 kB data)‡ 6.8 kB constants, 13.3 kB data)
ARM9	Spirit	22/17.5	39.2kB (19.7 kB program, 7.2 kB constants, 12.3 kB data)
AudioDE	Spirit	5.5/5	54kB (27 kB program, 27 kB data+constants)

† Does not include Layer I or II decoding ‡ Estimated from simulation.

Table 6.2: MP3 decoder comparison (lower values are better)

area for this improvement.

Finally, we believe that it is a huge advantage to be able to use floating point numbers since it is quite easy to convert a high level model in ANSIC C or Matlab to our processor as most operations can be immediately ported. If a fixed point DSP was used, more development time has to be spent on making sure that the dynamic range of the fixed point calculations are good enough. Our approach should lead to shorter development time and faster time to market. This advantage is harder to measure quantitatively however.

6.2.5 What Went Right

Overall this project was a success. We have shown that it is possible to implement an efficient audio processor with a floating point unit. The ability to use floating point arithmetic significantly reduces the firmware development time (and consequently time to market) as there is no need to analyze the algorithms very deeply as the floating point numbers will handle scaling automatically for us.

The maximum frequency of the processor is high and the efficiency of the processor is also very high, as any MP3 bitstream can be decoded while running at a frequency of 20 MHz.

The simple architecture without result forwarding and hazard detection meant that the verification was easy as few bugs were present in the design. The simple architecture and deep pipeline also means that the architecture is very suitable to use in an FPGA. But it is important to note that this architecture is also suitable for an ASIC as the performance in an ASIC is also high.

6.2.6 What Could Be Improved

As has already been mentioned, this project was mostly aimed at demonstrating that low precision floating point arithmetic can efficiently be used for audio applications. Therefore a number of details were not investigated fully and could certainly be improved.

Program Size

The most important thing to improve is program memory size. One way to improve this could be to improve the instruction encoding. If the instruction word could be reduced by two bits this would mean an 8% saving in program memory size. This would be possible at the expense of reducing the size of immediate constants and would therefore mean a slight decrease in performance due to the need to load certain constants into registers before use.

Another way to decrease the program memory size is to use loop instructions instead of unrolling some code. This could probably save around a kilobyte of program code, especially in the windowing, which consists of large unrolled convolution loops.

However, the most major savings would come from optimizing the Huffman decoder. Right now all Huffman tables are implemented entirely in software using the read-bit-and-branch-conditionally instruction. This is very wasteful and it is likely that the size of the Huffman tables (6.7kB) could be reduced by about half by using a memory with custom bit width and a small Huffman decoding accelerator. This would also have the advantage of accelerating the performance of the MP3 decoder as well.

Processor Architecture

While the processor architecture was certainly good enough to implement an efficient decoder the architecture of the processor was not very programmer friendly. Since the pipeline contained no hazard detection it was often necessary to reschedule the code to avoid data hazards. This was especially true for the floating point instructions as four unrelated instructions have to be issued before the result of such an instruction is available for use. For integer operations it is necessary to issue two unrelated instructions before the result is available.

The lack of hazard detection presents an interesting challenge for the programmer who has to schedule the code so as to minimize the need for NOP instructions in the code[2]. In the current firmware about 10% of the program consists of such NOPs. (Making it possible to remove these NOPs would reduce the firmware size by about 2 kB.) When running the MP3 decoder, around 6.6% of all executed instructions are NOPs.

The lack of register forwarding can actually be seen as a feature in some cases since intermediate values can be stored in the pipeline instead of in the register file making it possible to perform a 16 point floating point DCT without having to store any intermediate results in memory using only a 16 entry register file. However, this makes it hard to implement interrupts since the entire state of the pipeline would have to be saved and restored if the MP3 firmware should work correctly.

6.2.7 Conclusions

Although the xi processor successfully demonstrated that floating point arithmetic is a good idea for audio processors the processor and firmware could be improved:

[2]The assembler and simulator will help the programmer with this by warning him when he is not using the pipeline correctly.

- In a processor without result forwarding it is usually possible to implement DSP tasks efficiently through careful instruction scheduling to avoid data dependencies. (Although this is not trivial in some cases.)
- When a data dependency could not be avoided the lack of automatic stalling also meant that NOPs had to be inserted, at a cost of increased program memory size.
- The utilization of the pipeline for intermediate storage of results meant that it is cumbersome to implement interrupts
- The integer pipeline and the floating point pipeline had different lengths, making it possible to create a situation where both pipelines are trying to write to the register file at the same time.

Of these, the total lack of result forwarding is probably the worst problem. While it is an interesting and challenging job to write and to schedule code for the xi processor it is not a good use of developer time.

Chapter 7
A Soft Microprocessor Optimized for the Virtex-4

Abstract: *In this chapter the tradeoffs that are required to design a microprocessor with a very high clock frequency in an FPGA is described. The design has been carefully optimized for the Virtex-4 FPGA architecture to ensure as much flexibility as possible without making any compromises regarding a high clock frequency. Even though the processor is more advanced than the FPGA friendly processor described in the previous chapter, the FPGA optimizations allows it to operate at a much higher clock frequency. With floorplanning, the processor can achieve a clock frequency of 357 MHz in a Virtex-4, which is considerably higher than other FPGA optimized processors.*

Although the processor described in the previous chapter was successful it did have a number of flaws that made it less attractive for a programmer, in particular the lack of result forwarding is a big problem when it comes to programmer productivity. The challenge that will be discussed in this chapter is how to create a processor (called xi2), which is truly optimized for an FPGA while at the same time trying to address the total lack of result forwarding in the processor described in the previous chapter.

The initial goal for this project was to be able to utilize the floating point units described in Chapter 8 in a processor on an FPGA[1] This means that the design should be able to operate at around 360 MHz in a Virtex-4 (speedgrade 12). To determine if this was remotely feasible, a number of small designs were synthesized to the Virtex-4 to estimate their maximum operating frequencies:

- 32-bit Arithmetic logic unit
- Result forwarding
- Address generator
- Pipeline Stall generation
- Shifter

These components were selected as they were suspected to be difficult to implement in hardware at high speed based on previous experience with Xilinx FPGAs. For every item in the list a couple of different designs were tried until an acceptable solution was found.

7.1 Arithmetic Logic Unit

The arithmetic logic unit (ALU) can be considered to consist of two separate parts, a logic unit and an adder/subtracter. The logic unit was not expected to be a part of the critical path. It was however unknown how many bits an adder/subtracter could contain before limiting the performance of the processor. The performance of an adder with different number of bits is shown in Table 7.1.

The biggest problem is that the results in the table assumes that a flip-flop is present on all inputs and outputs of the adder. In a processor this means that there is no way to utilize the result of an addition in the next clock cycle. In order to support forwarding of results it is necessary to modify this circuit. Ideally a mux would be present on each input to the

[1]The floating point components have not been integrated yet as we believe that the current version of the processor is very competitive, even without floating point support.

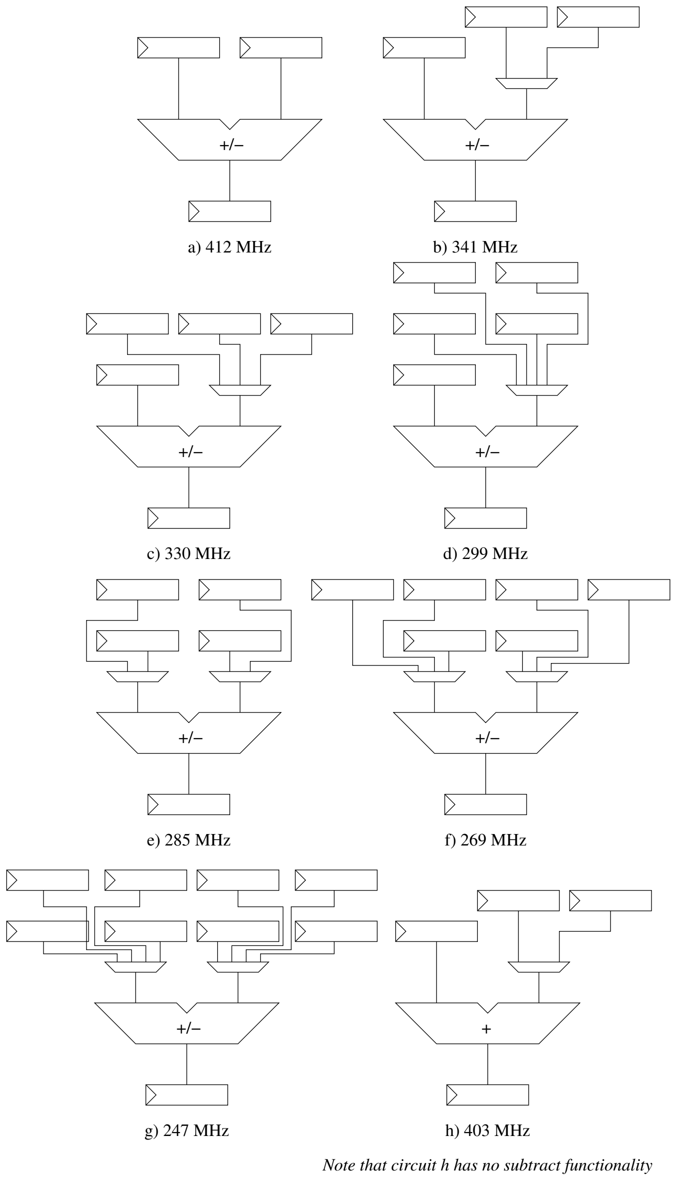


Figure 7.1: The maximum frequency of various adder/mux combinations in a Virtex-4 (fastest speedgrade)

Number of bits	Maximum frequency
16	529.6 MHz
24	467.5 MHz
32	412.2 MHz
40	377.8 MHz
48	338.2 MHz

Table 7.1: The performance of an adder/subtracter for various sizes

adder to select from a number of different results. Figure 7.1 shows the maximum performance attainable with various mux configurations.

As can be seen in Figure 7.1, the performance drops drastically compared to the 412.2 MHz shown in Table 7.1. This is especially true for an adder/subtracter with muxes on both inputs (*e*-*g*). This can be mitigated by removing the subtracter from the adder as seen in the last configuration (*h*) in Figure 7.1. Only configuration *a* and *h* can be implemented using one lookup-table per bit in the adder. The remaining configurations suffers from additional routing and logic delays.

Unfortunately configuration *h* doesn't directly support subtraction. This can be fixed by moving the inverter used for subtraction to the previous pipeline stage as seen in Figure 7.2. Additionally, by utilizing the Force0, Force1, InvA, and Swap signals, an arithmetic unit can be constructed that can handle most result forwarding operations without any penalty as described in Table 7.2. The only operation this configuration cannot handle is to forward the same result to both inputs since it is only possible to feed back the result to one input of the adder. (Although this is of little use when subtracting.)

Finally, the careful reader may have noticed that it is not possible to forward data from any other execution unit directly to the arithmetic unit. The lack of full forwarding is a weakness of this processor, but the implementation of partial forwarding allows a significant increase in clock frequency when compared to a processor with full forwarding. For example, if full forwarding was implemented it is likely that the con-

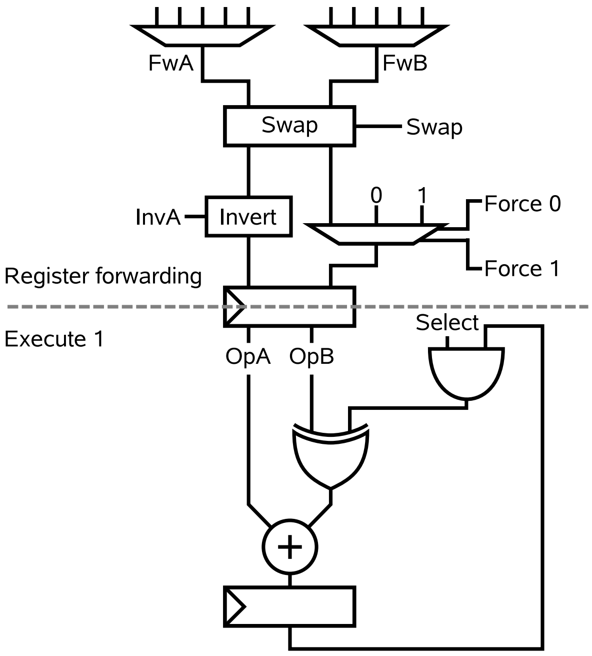


Figure 7.2: An arithmetic unit that can handle the most common result forwarding operations (Maximum frequency, 403 MHz in a Virtex-4 of the fastest speedgrade)

figuration in Figure 7.1*g* would have to be used, which is substantially slower than the current implementation. As a comparison, without any forwarding at all, the runtime cost of all NOPs that were inserted to avoid data dependency problems in the MP3 decoder in the previous chapter was around 6.6%. Since the F_{max} difference between full forwarding and partial forwarding is considerably larger than 6.6%, the tradeoff should be more than worth it in terms of performance. A more in-depth discussion of partial forwarding can be found in [42].

Instruction sequence	Forwarded operand	Control signals
add r2,r1,r0	-	-
add r2,r1,r2	OpB	Force0=1 Select=1
add r2,r2,r1	OpA	Swap=1 Force0=1 Select=1
sub r2,r1,r2	OpB	Force1=1 Select=1
sub r2,r2,r1	OpA	Swap=1 InvA=1 Select=1
sub r2,r2,r2	Both	Replace with *set r2,#0*
add r2,r2,r2	Both	Cannot forward directly

Table 7.2: Forwarding operands to the arithmetic unit (The order of the operands are *destination register, OpA, OpB).*

7.2 Result Forwarding

As has already been seen in the previous section, it can be tricky to implement forwarding in a processor when it is optimized for FPGAs. It simply isn't possible to use as many muxes as one would like since they won't fit into a critical path in combination with for example an adder. And as was seen in Section 5.6, the area cost for muxes is very large in an FPGA as well. The number of muxes should therefore be reduced for both area and performance reasons.

A common way to reduce the multiplexers in optimized soft core processors is to utilize the reset input of a pipeline register to set unused registers in all pipeline stages to 0. When this has been done only an or-gate is required to select the correct result from that particular pipeline stage. This is illustrated in Figure 7.3. While a mux is still required to select which pipeline stage to forward a result from, muxes are not required to select the execution unit in that particular pipeline stage.

It is not enough that the pipeline is able to support result forwarding. It is also necessary to detect that it is necessary to do so. One way to do this is to simply encode all forwarding information into the instruction word. This is the most simple solution in terms of hardware. Unfor-

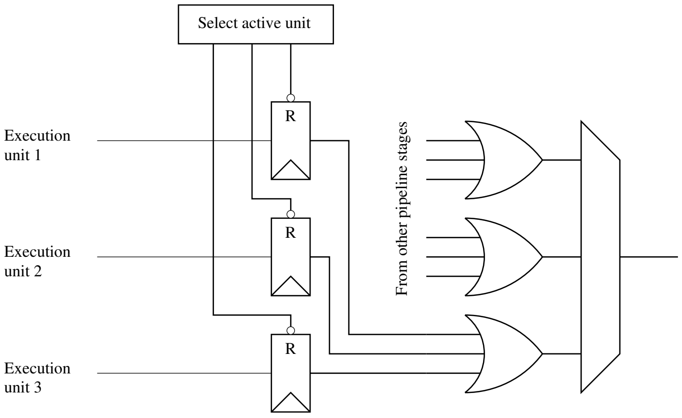


Figure 7.3: Result forwarding with reset/or structure

```
Original code            With manual forwarding
     sub r1,r9,r3             sub r1,r9,r3
     add r2,r1,r7             add r2,FWAU,r7
loop:                    loop:
     add r2,r2,#1             ; Ambiguous
     ...                      ...
     ...                      ...
     bne loop                 bne loop
```

Figure 7.4: Result forwarding controlled by software (The order of the operands are *destination register, OpA, OpB*)

tunately it is not an optimal solution. Consider the assembly code in Figure 7.4. On line 2 the result of the previous operation is forwarded by using FWAU as the source register. The first instruction of the loop should also use FWAU instead of r2 as the source register the first time the loop is run. However, the second time r2 should be read from the register file. In conclusion, while it is not impossible to use manual forwarding in a processor, to do so places severe restrictions on the programmer. Interrupts are also troublesome to implement as the control flow can now be interrupted at almost any time, disrupting the manual forwarding process.

To simplify things for the programmer, automatic result forwarding was implemented in favor of letting the programmer handle it. Figure 7.5 shows the first experiment as *a*. By putting the forwarding decision into the decode stage the result is available in the next pipeline stage. Unfortunately the performance of 337 MHz for this configuration when used in the processor was not quite satisfactory. (Mostly due to the complexity of quickly generating the control signals required for the forwarding in the arithmetic unit described in the previous section.)

Instead, some matching logic was moved to the pipeline stage before the decode stage. This means that the time available for reading from the program memory is shortened somewhat. But the performance of the program memory readout is still satisfactory as only a single LUT is inserted before the flip-flops. This is shown as *b* in the figure.

It should also be noted that the forwarding logic in the xi2 processor is slightly more complicated than described in this section as this pipeline stage also has to generate the Force0/Force1/Swap/InvA signals shown in Figure 7.2. It also has to handle data from the constant memory and immediate data from the instruction word.

7.3 Address Generator

A very common structure in DSP applications is the circular buffer. To efficiently handle for example FIR filters a DSP employs a number of techniques:

- Combined Multiply-Accumulate (MAC) unit
- Hardware loop support
- Circular addressing to feed the MAC unit with data

The MAC unit is relatively easy to support using the DSP48 blocks in the Virtex-4 FPGA. Hardware loop support is also relatively easy to

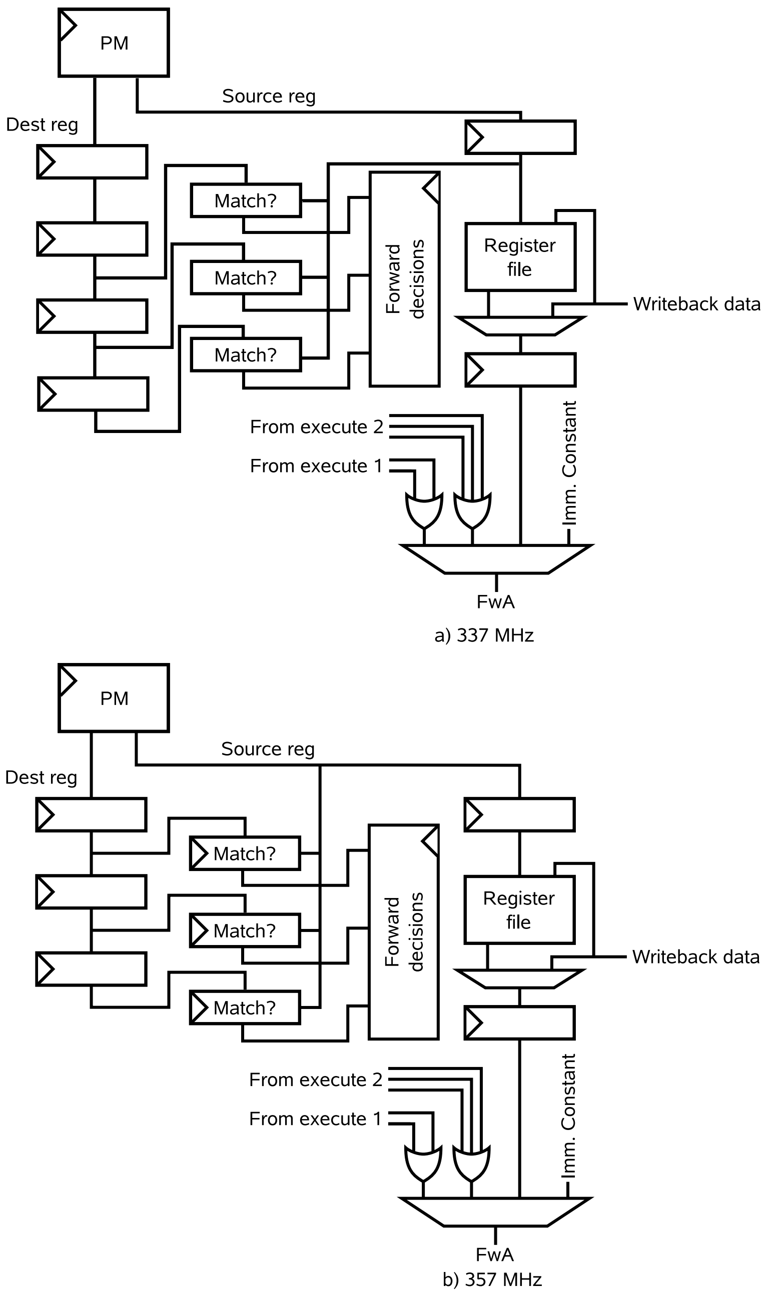


Figure 7.5: Performance for different forwarding configurations when used in the processor

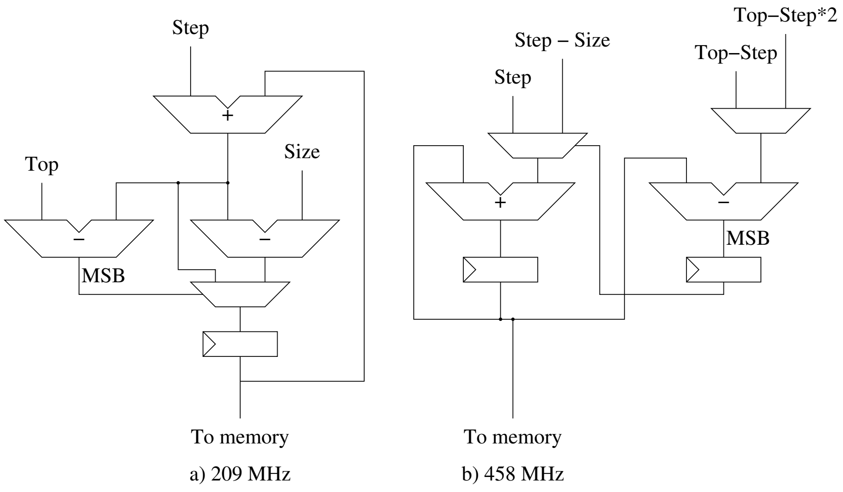


Figure 7.6: Address generators for modulo addressing

implement in the program counter module. However, the address generator is not as easily implemented. When written in C, the following assignment describes one step in a circular buffer[2]:

ADDR = (*ADDR* + *STEP* - *BOT*) % *SIZE* + *BOT*;

If *SIZE* is not a power of two, this addressing mode is clearly going to be expensive to implement. However, if the step size is small enough, it can be implemented inexpensively using the following C code:

ADDR += *STEP*; if(*ADDR* $>$ *TOP*) *ADDR* -= *SIZE*; if(*ADDR* $<$ *BOT*) *ADDR* += *SIZE*;

If it is certain that the buffer only has to be traversed in one direction, one of the conditions can be removed. A hardware implementation of this is shown in Figure 7.6 (*a*). The performance of this configuration is clearly not good enough for our purposes. However, if it is acceptable to use slightly different parameters configuration *b* can be used instead. The major disadvantage of configuration *b* is that the end parameter (*TOP*-2*STEP*) cannot be used for the first iteration and *TOP-STEP* has to be

[2]The addressing mode is sometimes referred to as modulo addressing due to the use of the modulo operator

used instead. (This could be handled automatically by the CPU when updating this register at a cost of some extra hardware.)

7.4 Pipeline Stall Generation

There are several ways to handle hazards in a program. The most flexible way is that the processor automatically detects the hazard and stalls the processor until the required data is available. The opposite solution is to ignore all hazards in hardware and trust the programmer to never create a situation the hardware cannot solve. The first version of the MIPS processor did not have any hardware to detect this situation for example (although later versions do) [43]. There are advantages and disadvantages to both solutions. A software based solution increases the code size because of forced *NOP* instructions. A hardware solution allows code density to be low while increasing the complexity of the hardware, possibly increasing the critical path, and complicates verification as there are many more situations to test.

A software based solution can also be advantageous because the pipeline registers themself can be used to store temporary results. Consider the following C code for example:

```
tmp = A - B; B = A + B; A = tmp;
```

In a regular processor this will need three registers. However, in a processor without hazard detection and result forwarding it can be done using only two registers:

```
add rA,rB,rA ; New value available after 2 cycles
sub rA,rB,rB ; This will use the _old_ value of rA
```

Effectively, this allows the programmer to utilize the pipeline registers in the processor as extra storage. This allows for example a 16 point DCT to be performed using only 16 registers without resorting to any temporary storage, a technique that was heavily utilized in the MP3 decoder described in the previous chapter. The drawback with this architecture is that it is difficult to implement interrupts as the current state of the pipeline registers must be saved as well.

Ideally, when implementing hazard checking in hardware, it is easiest if the pipeline can be stalled as early as possible. Initial experiments indicated that it would not be possible to stall the pipeline in the decode stage, which is the ideal place to place this. While it would be possible to stall an instruction in later pipeline stages this would complicate the pipeline and it was decided that an increase in program size could be accepted for now. This is a slight disappointment since the lack of automatic stalling was a rather large inconvenience when programming the xi processor described in the previous chapter. However, if the assembler is capable of automatically inserting NOPs as needed, the programmer will never notice that hazard checking is not implemented in the hardware except as an increase in program memory usage.

7.5 Shifter

The final component in the execute pipeline stage is the shifter. Figure 7.7 shows a number of different shift configurations. *a* and *b* shows a simple 32 bit shifter and the shifter in *c* can shift both left and right. It is no surprise that the version that can shift in both directions is slower. By pipelining version *c*, we arrive at version *d*, which is barely fast enough but lacks arithmetic right shift. Adding an arithmetic right shift to *d* results in the relatively slow *e*. As both arithmetic and logic shift is typically required in a processor another approach is necessary. *f* implements *e* in a different way. Instead of implicitly writing the >>> operation in Verilog, it is implemented by generating a mask that is always or:ed together with the result of the logic right shifter in the second pipeline stage. Another optimization is that a second unit determines in parallel if the shift will be longer than 32 bits. In that case the result register will be set to zero, regardless of the result of the shift (except for arithmetic right shift of a negative number, which is handled by the mask generation unit). Version *f* is used in the processor.

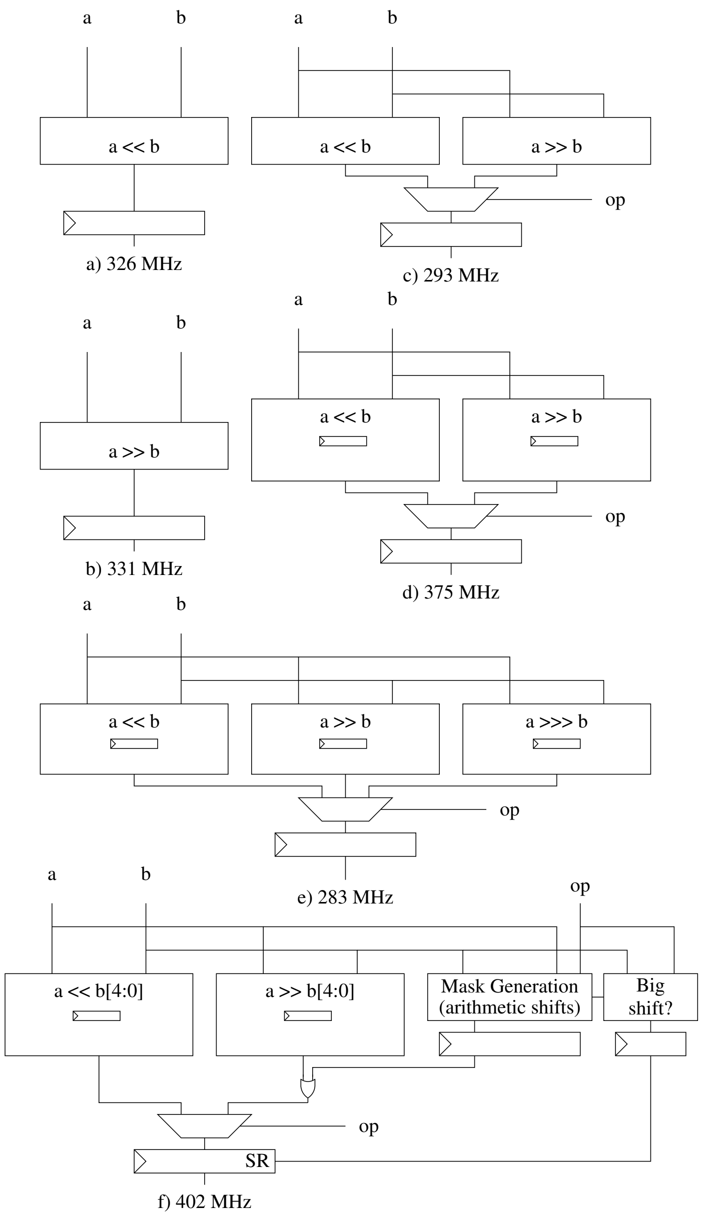


Figure 7.7: Various 32-bit shifter architectures

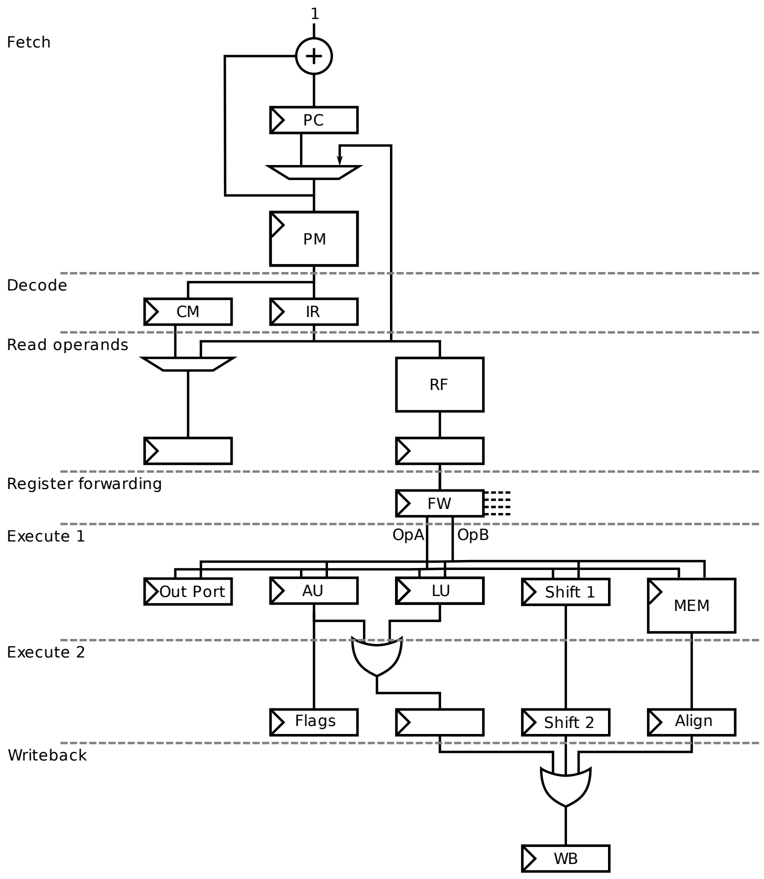


Figure 7.8: The overall architecture of the processor

7.6 Other Issues

A somewhat simplified view of the final pipeline is shown in Figure 7.8. There are seven pipeline stages (fetch, decode, read operands, register forwarding, execute 1, execute 2, and writeback). (Although it could arguably be said that the PC register is yet another pipeline register.) Also, the MAC unit is not shown in the pipeline figure (see below for more information).

7.6.1 Register File

The register file is 32-bit wide and contains 16 entries. It has two read ports and one write port and it is implemented using distributed RAM. The register file itself can probably be extended to 32 entries without any timing problems, but the forwarding logic described in Section 7.2 may have some performance problems in that case.

7.6.2 Input/Output

Input and output is handled through special instructions that can write and read to a number of I/O-ports. Only one output port and one input port is implemented, but it is also possible to update internal registers like the address generator registers using these instructions. The instruction word is 27 bits. In the performance numbers quoted in this thesis a 32-bit memory was used regardless of this, but if a larger program has to be used, the reduced number of bits in the instruction word means that 3 block RAMs with 9-bit wide memory ports could be placed in parallel instead of using 4 block RAMs.

7.6.3 Flag Generation

There are four status flags (zero, overflow, negative, and carry) and arithmetic and logic instructions are able to influence these flags. Of these flags the most tricky to generate is the zero flag since that flag depends on the entire result of an arithmetic or logical operation. To optimize this, the Z flag generation is partially done in the arithmetic unit and logic unit on the combinational outputs from the adder/logic unit. In the logic unit, all 32 bits are preprocessed down to 8 bits by using a logic or operation on 4 bits at a time. In the arithmetic unit there is not enough time available to do this on all bits after the addition, but the 20 lower bits are preprocessed into 5 bits in this way, which means that a total of 25 bits have to be considered in the next stage when generating the Z flag instead of 64 bits.

7.6.4 Branches

Delay slots are used for all branches. Apart from that, there are no penalties for a correctly predicted branch. If the branch is mispredicted there is a penalty of either three or four cycles depending on whether it is predicted as taken or not taken. Register indirect branches always have a penalty of four cycles.

There are only absolute branches available, but this is not a problem as the address space for the memories is only 16 bits wide and the target address will fit into the instruction word. Finally, there is a loop instruction that allows small loops to be implemented with a minimum of loop overhead.

7.6.5 Immediate Data

There is an eight bit immediate field that can be used instead of one of the operands from the register file. If the most significant of these bits are 0 the remaining seven bits are sign extended to a 32-bit value. If this bit is 1, the remaining seven bits are used to address a constant memory that contains 32 bit constants. It is therefore possible to use up to 128 arbitrary 32-bit constants without any penalty.

Finally, for those situations where 128 constants are not enough it is also possible to use a special *SETHI* instruction with a 24 bit immediate. These 24 bits will be concatenated with the eight bit immediate field the next time an instruction with immediate data is used. This will lead to a one instruction penalty when loading arbitrary 32-bit constants, which is similar to many other RISC processors where two instructions are required to load arbitrary 32-bit values into registers.

7.6.6 Memories and the MAC Unit

There are three different memory spaces available in the processor. The address space for all memories is 16 bit wide, although only 2 KiB large block RAMs have been used so far. What is not shown in the pipeline in Figure 7.8 is that the second port of the constant memory (*CM*) and data memory (*MEM*) is connected to the address generators described in Section 7.3. A special part of the instruction word instructs the pipeline to replace the OpA or OpB value in Figure 7.8 with a value read from either the constant memory or the data memory.

When this ability is used in conjunction with the MAC operation this allows a convolution to be efficiently performed. The MAC unit itself is also not shown in the pipeline diagram, but it is based on instantiated DSP48 blocks. It allows for 32x32 bit multiplication and 64 bit accumulation. The results are accessed by reading from special registers as described in Section 7.6.2. The MAC unit itself contains 6 pipeline stages and the first stage is located in the Execute 1 stage.

7.7 Performance

In the beginning of this project the author thought it unlikely that this processor would be able to compete with established commercial FPGA microprocessors like MicroBlaze, hence the initial focus on DSP processing. However, an initial design (lacking many features/instructions) written directly in Verilog could be synthesized to almost 400 MHz in a Virtex-4 (speedgrade 12). At this point we realized that it may be possible to create a more general purpose microprocessor that can operate at a much higher clock frequency than MicroBlaze. The focus shifted from a DSP processor to a microprocessor with DSP extensions at this point.

This processor has been optimized for a specific FPGA architecture. While it is possible to synthesize the design without any changes for the Virtex-5 instead of the Virtex-4, the performance when doing so is not higher than in Virtex-4. A higher performance could probably be reached if the processor was redesigned around the 6 input LUTs of the Virtex-5. In fact, Table 7.3 shows that the performance in a Virtex-5 is lower than in a Virtex-4.

Device	F_{max}	Area
xc4vlx80-12	334 ($357^{\dagger}$)	1682 LUTS, 1419 FF, 3 RAMB16, 4 DSP48
xc5vlx85-3	320	1433 LUTS, 1419 FF, 3 RAMB16, 4 DSP48E
ASIC (Direct port)	325	1.4 mm^2
ASIC (Rewritten MAC)	500	1.3 mm^2

† With floorplanning

Table 7.3: Performance and area of the xi2 processor

7.7.1 Porting the Processor to an ASIC

It is interesting to look at this processor when porting it to an ASIC. When using a compatibility library for every FPGA construct, including the DSP48 blocks, the area for the processor is 1.4 mm^2 and the maximum frequency is 325 MHz. In this case, the critical path is in the MAC unit. This is relatively discouraging as it is slower than the xi processor described in the previous chapter.

As the critical path is in the MAC unit it makes sense to investigate this further. The performance of the MAC unit itself when synthesized separately is shown in Table 7.4. Three different kind of configurations have been tried here. The first is the version that is based on the DSP48 block. This configuration is only available for the Virtex-4 and Virtex-5. The second configuration is based on a limited wrapper of the DSP48 block which allows the MAC to be ported to an ASIC without any changes to the source code. The final configuration is based on a rewritten MAC unit where the multiplier is implemented using a pipelined DesignWare multiplier. When using this configuration the maximum clock frequency of the processor is increased to 500 MHz and the area reduced to 1.3 mm^2. Rewriting the MAC unit was clearly a good idea. In this case the critical path is mainly caused by the path from the program memory to the constant memory. Changing the memory to a version that is optimized for speed instead of area will probably improve this (at a cost of increased area/power), but we have not been able to test this yet due to a lack of appropriate memory models.

Configuration	Device	F_{max}	Area
DSP48	xc4vlx80-12	500	4×DSP48
DSP48	xc5vlx85-3	550	4×DSP48
DSP48 based	130nm ASIC (speed)	424	0.19 mm^2
DSP48 based	130nm ASIC (area)	83	0.10 mm^2
Rewritten	130nm ASIC (speed)	743	0.13 mm^2
Rewritten	130nm ASIC (area)	63	0.058 mm^2

Table 7.4: The performance of the MAC unit in different architectures

Another memory related problem with this design in an ASIC is that dual port memories have been used for both the constant memory and the data memory. For the constant memory it is easy to separate it into two memories, one for immediate constants used in the program and one for constants that should be used for convolution operations. For the data memory, one of the ports are used to read data during convolution operation and the other port is used for normal read/write access to the memory. This is a harder issue to fix that may require a redesign of the pipeline. One possibility could be to use a dual port memory for a small part of the address space and single port memories for the remaining address space. The drawback would be that the programmer would then have to place circular buffers in the correct memory region.

7.8 Comparison with Related Work

An early processor optimized for FPGAs is described in [44] which describes a RISC based CPU mapped to a Xilinx XC4005XL FPGA. While the FPGA is old, this is still a very interesting publication that highlights many issues that are still true today.

However, soft processor cores were not popular until larger FPGA devices appeared. Currently, the major FPGA vendors have their own solutions in the form of MicroBlaze [45], Nios II [46], and Mico32 [47]. Readers interested in soft CPUs for Altera FPGAs are encouraged to read

James Ball's chapter in [48]. Another interesting Altera related publication discusses how to optimize an ALU for Altera FPGAs [49].

Finally, while not explicitly designed for FPGAs, Leon [50] and Open-Risc [51] can both easily be synthesized to FPGAs but the performance will not be very good when compared to FPGA optimized processors.

7.8.1 MicroBlaze

The most natural processor to compare this work with is the MicroBlaze from Xilinx, which has been optimized for Xilinx FPGAs. The maximum clock frequency of the MicroBlaze is going to be around 200 MHz [52] in a Virtex-4 of the fastest speedgrade. However, it should be noted that MicroBlaze is a much more complete processor than the processor described in this chapter. For example, MicroBlaze has better forwarding, cache support, and support for stalling the processor when a hazard occurs. At the same clock frequency, it is very likely that the performance of MicroBlaze will be higher than the processor described here. On the other hand, xi2 has a maximum clock frequency which is over 70% higher than MicroBlaze. We believe that xi2 will still have a comfortable performance advantage for many applications, especially those that involve DSP algorithms.

Unfortunately the source code of MicroBlaze is not publicly available so it is not possible to investigate how well it will perform in an ASIC.

7.8.2 OpenRisc

The OpenRisc or1200 processor is an open source 32-bit microprocessor that is available at the OpenCores website. It has similar features to the MicroBlaze. When a version of the or1200 processor with 8 KiB instruction and data cache + 4 KiB scratch pad memory was synthesized to a 130 nm process the maximum frequency was around 200 MHz. When synthesized to an FPGA, the performance was around 94 MHz.

It is clear that the or1200 processor is not optimized for FPGAs, but it is also interesting to note that the performance of our xi2 processor is significantly higher in an ASIC as well. Although in fairness to the OpenRisc processor it should be noted that the or1200 is certainly more general than our processor as well.

7.9 Future Work

While the processor architecture described in this publication seems very promising, it does have some quirks that makes it harder to use. Regardless of these quirks we believe that it would be worthwhile to continue the development of this processor and that the general architecture is a good one.

One thing that is currently missing is interrupt support. This would be relatively easy to add, especially if the interrupt delay is permitted to be slightly non-deterministic so that the CPU can never be interrupted in a delay slot. Another thing that could be added is some more instructions although this will probably require the instruction word to be extended beyond 27 bits. Most notably a division instruction of some sort is missing.

The issues mentioned in the previous paragraph are relatively minor but there are a few major shortcomings in the current architecture. The lack of caches is a serious problem if general purpose development should be done on this architecture. Handling cache misses is a non-trivial problem and it is not clear how this can be implemented in the best way on this processor.

Another major issue is the lack of a compiler. This means that it is very hard to benchmark the processor using realistic benchmarks. Right now we believe the processor to be good based on the fact that it should be better than the xi processor in many ways (more forwarding, 32-bit integer operations instead of 16-bit, better AGUs, higher clock frequency, and branch prediction). Since we consider the xi processor to be a success, it follows that xi2 should be even better. However, without running real benchmarks on the processor we cannot prove this.

Another interesting research direction would be to continue the work started in the previous chapter and try to design a processor that has very high performance in both FPGAs and ASICs with a minimum of customizations for each architecture.

7.10 Conclusions

We believe that the processor described in this chapter is a very promising architecture. An improved version of this processor with a cache and a compiler may be a serious alternative to other FPGA optimized processors, especially for DSP tasks where data dependencies can usually be avoided by careful code scheduling.

The high F_{max} of this design could be reached by carefully investigating the critical paths in all parts of the processor during the entire design flow. By tailoring the architecture around these paths it was possible to reach 357 MHz in a Virtex-4 (speedgrade 12). However, the final solution represents a compromise between frequency and flexibility. One of the tradeoffs is that the pipeline of the processor is visible to the programmer. For example, data hazards have to be managed in software since the processor cannot detect this and stall the processor. A good toolchain should be able to compensate for this, but this will also mean that it will be hard to retain binary compatibility if the processor is improved.

When the processor is ported to a 130 nm ASIC process the performance of the processor is 500 MHz and is mostly limited by the performance of the memory blocks. To reach this performance in the ASIC port it was necessary to rewrite the MAC unit to avoid the limitations enforced by the DSP48 blocks in the FPGA version.

Chapter 8
Floating point modules

Abstract: *This chapter describes how floating point components can be optimized for FPGAs. The focus in this chapter is to create a high speed floating point adder and multiplier with relatively low latency. The final solution has a maximum frequency which is higher than previous publications at a comparable pipeline depth although this comes at a price of a larger design area. The floating point components can be easily ported to an ASIC with only minor modifications.*

All calculations in DSP systems can be performed using fixed point numbers. However, in many cases the dynamic range of the data makes it difficult to design a solution using only fixed point. Either the fixed point numbers will be very wide and hence waste resources or scaling has to be employed which will increase the design time.

To avoid these problems it is possible to use floating point numbers instead. If floating point numbers are used, the result of an operation will automatically be scaled and it is therefore possible to represent a much larger dynamic range than a fixed point number of the same width. The use of floating point numbers can also reduce the width of memories, which can decrease the amount of on-chip memories as described in Chapter 6.

This chapter concentrates on floating point addition/subtraction and multiplication as these are the most commonly used operators.

8.1 Related Work

Floating point arithmetic has of course been intensively studied, but there are not so many recent publications for FPGAs. A paper which studies a tradeoff between area, pipeline depth and performance is [53]. However, their design is probably too general to utilize the full potential of the FPGA.

A very impressive floating point FFT core is presented in [54]. The core can operate at 400 MHz in a Virtex-4 and the inputs and outputs are single precision IEEE 754 compliant. However, numbers are not normalized inside the core and the latency is high.

Xilinx has IP cores for both double and single precision floating point numbers [55]. Nallatech also has some floating point modules available [56]. Neither has chosen to publish any details about their implementation though.

Another interesting project is FPLibrary at the Arenaire project [57]. Although the modules are not extremely fast, the source code is available and the project has many function blocks available besides addition and multiplication.

8.2 Designing Floating Point Modules

A floating point multiplier is quite easy to implement. The mantissas are multiplied and the exponents are added. Depending on the result of the multiplication, the result might have to be right shifted one step and the exponent adjusted. A typical pipeline for floating point multiplication is shown in Figure 8.1.

Floating point addition is more complicated than multiplication. At first the exponents are compared and the mantissas are aligned so that the exponent for both mantissas are equal. After the addition the result

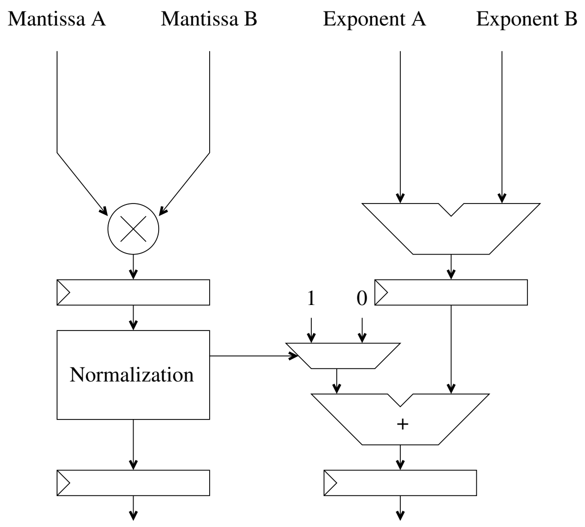


Figure 8.1: Typical floating point multiplier

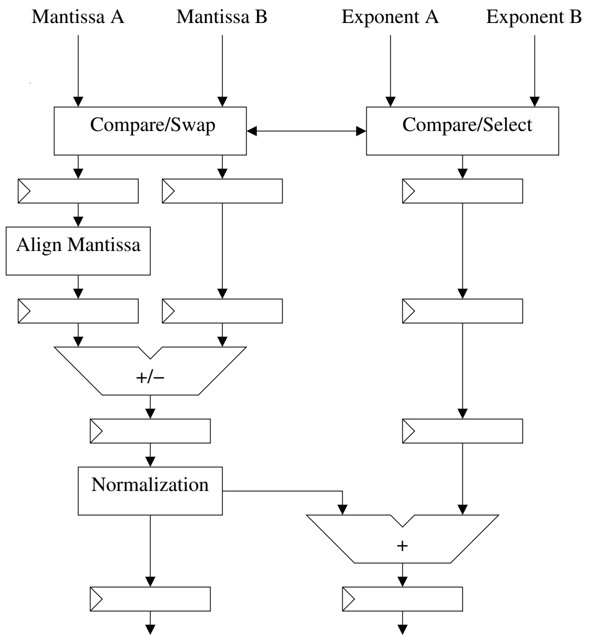


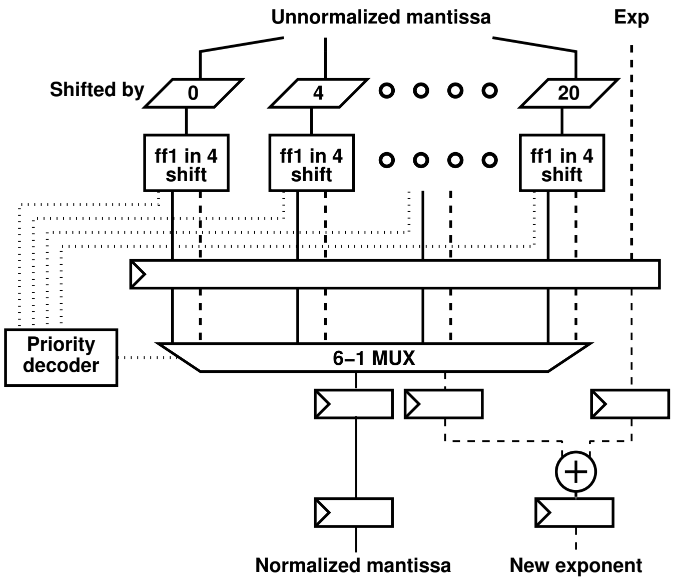
Figure 8.2: Typical floating point adder

must be normalized and the exponent updated. If two positive numbers or two negative numbers are added, the normalization is just as easy as in the case of the multiplier. At worst, the mantissa has to be right shifted one step. However, if a negative and a positive number are added, the mantissa might have to be left shifted an arbitrary number of times (this is often referred to as cancellation). Finally, the exponent must also be updated depending on how many times the mantissa was shifted. A typical pipeline for floating point addition is shown in Figure 8.2. By comparing the magnitude as the first step we make sure that the smallest number is always sent to the “Align Mantissa” module. This also assures that the result of a subtraction in the third pipeline stage will always be positive. Finally, in the exponent part of the pipeline, it is only necessary to retain the exponent of the largest number.

8.3 Unoptimized Floating Point Hardware

As a first performance test, the floating point adder and multiplier from [58] was synthesized to a Virtex-4 (speedgrade 12). These modules are closely based on the architecture in Figure 8.1 and 8.2. The major difference is that the multiplier consists of four pipeline stages instead of 2. The source code of these modules were written in VHDL and were not optimized for FPGA usage. The maximum frequency of the multiplier and adder is 207 MHz and 190 MHz respectively when synthesized with ISE 9.2. The mantissa is 16 bits and an implicit 1 is used instead of an explicit 1[1]. The exponent is 6 bits and a sign bit is used to represent the signedness of the number.

[1]If an implicit 1 is used, the first bit in a floating point number is assumed to be set to 1. This is always the case with a normalized floating point number. If an explicit 1 is used, the first 1 is stored in the mantissa which is necessary if unnormalized floating point numbers will be used.



©2006 IEEE. Reprinted from Norchip Conference, 2006. 24th, *High Performance, Low Latency FPGA based Floating Point Adder and Multiplier Units in a Virtex 4*, Karlström, P. Ehliar, A. Liu, D.

Figure 8.3: Parallelized normalizer

8.4 Optimizing the Multiplier

The critical path of the multiplier in the previous section is in the DSP48 multiplier block. The problem is that the synthesizer doesn't instantiate a pipelined multiplier correctly. By manually instantiating the appropriate DSP48 component, the floating point multiplier can operate at over 400 MHz without any further FPGA optimizations. This example demonstrates that it is easy to get good performance out of a floating point multiplier in an FPGA.

8.5 Optimizing the Adder

As for the floating point adder, the main bottleneck in the floating point adder mentioned above is the normalizer when synthesized for an FPGA. The normalizer must quickly both determine how many steps the mantissa should be shifted and shift the mantissa. This is quite complicated to do efficiently in an FPGA. One way to optimize this part is by using parallelism. A normalize unit is constructed which can only normalize a number with up to four leading zeros. Several of these units are then placed in parallel. A priority decoder is used to determine the first unit with less than four leading zeros. A final mux selects the correct mantissa and an adjustment factor for the exponent. This is illustrated in Figure 8.3 for a mantissa width of 23 bits.

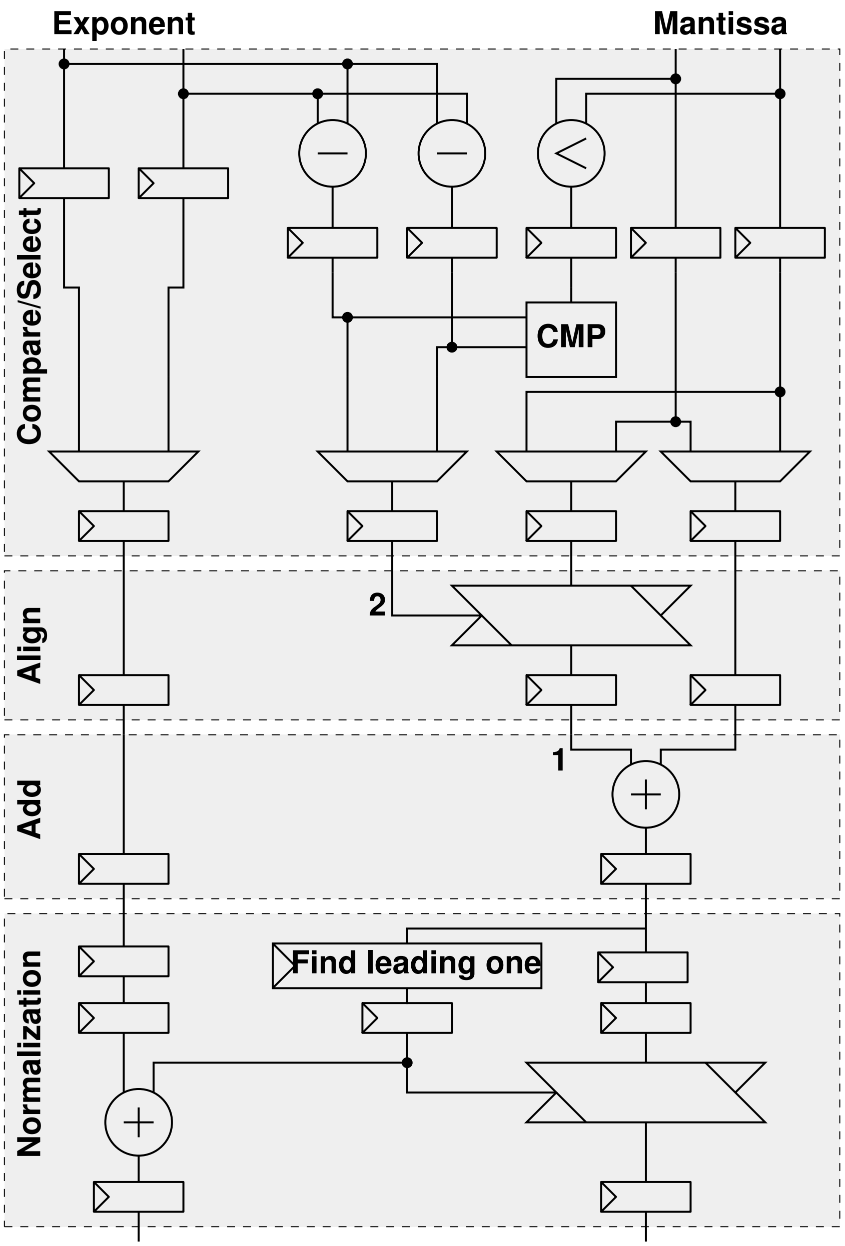
By incorporating this three stage normalizer into our floating point adder and extending the compare/select pipeline stage into two stages, we have constructed a floating point adder capable of operating at 361 MHz. In this case the width of the mantissa is 23 bits (plus the implicit 1) and the exponent 8 bits. This can be seen in Figure 8.4.

Other optimizations include a special signal to zero out the mantissa of the smallest number (marked with 1 in the figure) if the control signal to the shifter (marked with 2) is too large. This means that the shifter itself only has to consider the five least significant bits. The cost of the special "set to zero" signal is small as an otherwise unused LUT input in the adder is used for it as shown in Figure 8.5.

Given these optimizations, the floating point adder with 7 pipeline stages is able to support a throughput of 361 MHz in a Virtex-4 (speedgrade 12).

8.6 Comparison with Related Work

A comparison with the related work shows that our performance is similar to commercial IP cores. In Table 8.1, we compare our modules (DA in the table) to the floating point modules generated by CoreGen in ISE. As can be seen, our processor has a higher frequency but also higher area usage. Our architecture is therefore very interesting to architectures where it is necessary that the latency is kept low and where it is acceptable to use a higher area. However, if the Xilinx components are configured for maximum throughput it will be faster than our solution, but the latency



©2006 IEEE. Reprinted from Norchip Conference, 2006. 24th, *High Performance, Low Latency FPGA based Floating Point Adder and Multiplier Units in a Virtex 4*, Karlström, P. Ehliar, A. Liu, D.

Figure 8.4: Optimized floating point adder

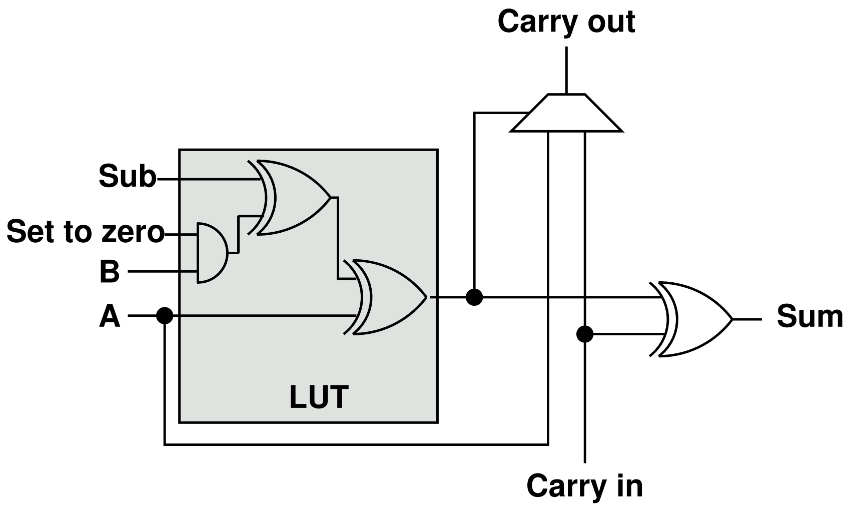


Figure 8.5: Adder/subtracter with set to zero functionality

Design	F_{max}	LUTs	Flip-flops	DSP48
Our FP Adder	370	894	643	0
Our FP Mul	372	140	281	4
Xilinx FP Adder	320	547	404	0
Xilinx FP Mul	341	122	290	4

Table 8.1: Performance comparison of two different floating point implementations with 8 pipeline stages in a Virtex-4 speedgrade -12

will also be much higher than our solution. (Up to 16 pipeline stages in the adder and 11 pipeline stages in the multiplier.)

8.7 ASIC Considerations

In an ASIC, the performance of these floating point components are not very good when ported using the compatibility library described in Section 5.2. The critical path in the ASIC version of the adder is in the addi-

Technology	Component	F_{max}	Area
130nm ASIC (speed)	FP adder	796	0.050 mm^2
130nm ASIC (area)	FP adder	146	0.037 mm^2
130nm ASIC (speed)	FP mul	978	0.072 mm^2
130nm ASIC (area)	FP mul	83	0.042 mm^2

Table 8.2: Performance of the floating point components when ported to an ASIC.

tion stage which may be caused by the fact that this part is implemented using instantiated FPGA primitives as seen in Figure 8.5. Once this part is replaced with code that infers the same functionality the performance is increased from around 400 MHz to 796 MHz.

For the multiplier, the critical path is not surprisingly in the multiplier itself. When using DSP48 based version ported to the ASIC, the maximum performance is 421 MHz. When replacing this version with a DesignWare based multiplier the performance is increased significantly. The performance and area of the ASIC port is summarized in Table 8.2.

8.8 Conclusions

Optimizing a floating point components for a specific FPGA is an interesting problem with many opportunities to trade area for frequency and vice versa. It is typically not a problem to create a high performance floating point multiplier in an FPGA, but a floating point adder is a real challenge, especially the normalization stage.

By liberal use of instantiated FPGA primitives it was possible to reach a very high performance, even higher than Xilinx' floating point adder with the same pipeline length. The price we pay for the performance is a higher area which means that our floating point adder is a good choice when few but fast adders are required. Our adder would probably be a good choice for a soft microprocessor whereas Xilinx' adder would be a good choice when a datapath with high throughput but modest latency requirement is needed.

The designs will not have very good performance in an ASIC if they are ported directly without any modifications at all, but the performance increases dramatically after minor modifications to the design even though most of the design still consists of many instantiated FPGA primitives.

Part III

On-Chip Networks

Chapter 9
On-chip Interconnects

Abstract: *This chapter is intended to serve as a brief introduction to on-chip interconnections. Readers who are already familiar with this concept may wish to skip this chapter.*

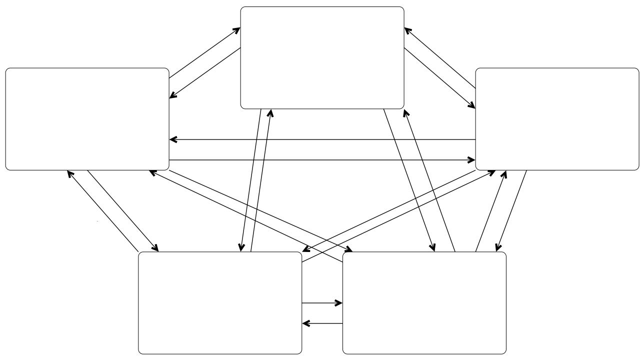
9.1 Buses

A bus is a simple way to connect different parts of a computer at low cost. This was recognized early on as even the very first computers utilized buses, such as for example the electromechanical Z3 [59].

The advantages of a bus is clear when looking at Figure 9.1. Instead of 20 dedicated connections between each of the five components there is only one shared bus to which all components are connected. This will significantly reduce the complexity and cost of the system under the assumption that the traffic between the components do not overload the bus.

One assumption here is that the total number of messages that will be sent during a certain time period does not exceed the capacity of the bus during the same time period. In many cases, this will also mean that messages must be buffered for a while before they can be sent over the bus.

Traditionally, buses were implemented using three-state drivers to


(a) Connecting five components with dedicated connections

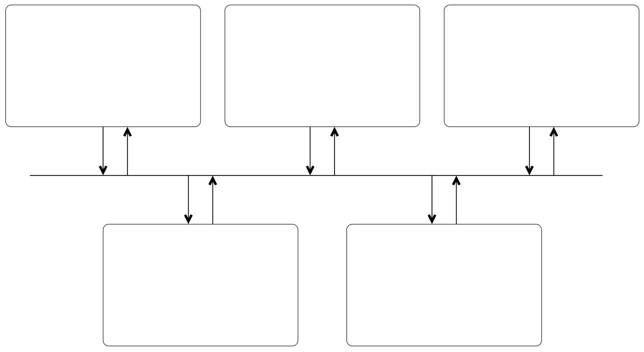
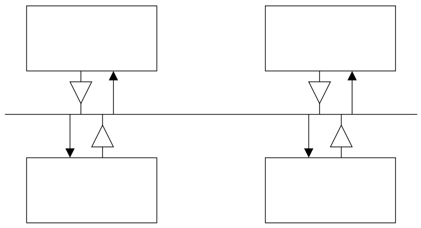

(b) Connecting five components with a shared bus

Figure 9.1: Dedicated connections vs a shared bus

save area but this is very rarely used for on-chip buses any longer due to the increased verification cost and slow performance of such buses [4]. Instead, on-chip buses can be implemented by using muxes as shown in Figure 9.2. (Unless otherwise noted, a bus in this thesis refers to a bus implemented using multiplexers.)

9.1.1 Bus Performance

A variety of parameters affect the performance of a bus. The most important parameters are the operating frequency, the width of the bus and the choice of bus protocol.


(a) Bus based on three-state signals

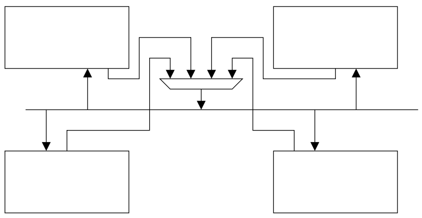

(b) Bus based on muxes

Figure 9.2: Buses implemented using three-state signaling and muxes

Other things that will indirectly impact the performance is the number of components connected to the bus. If an on-chip bus is constructed that has many components connected to it, it will not be able to operate as fast as a bus with few components due to physical constraints (e.g. wire delays).

9.1.2 Bus Protocols

A simple bus protocol has only two types of transaction, single address read and single address write. Typically it is possible for a bus slave to delay a transaction if it cannot respond immediately, for example if a read transaction to external memory would take more than one clock cycle to complete.

In many cases a bus master may want to read more than one word at a time. This is sometimes handled by signaling how many words to read in the beginning of the transaction. This is how the PLB bus in IBM's CoreConnect architecture operates. Another way is to merely signal the intention to read multiple words which will allow a slave to speculatively read ahead in memory. This is how the PCI bus works.

If a bus is used by many bus masters and there are some high latency devices connected to the bus, a single bus master could lock the bus for a hundreds of cycles when reading from a slow device. (For example, trying to read an I/O register in a PCI device connected to a PCI bridge. One way to avoid this problem is to allow a slave to tell a bus master to retry the transaction later on when the requested value may be available. This allows the bus to be used by another bus master while the value requested by the first bus master is fetched.

A drawback with bus retries is that the bus can be saturated with retry requests. A common way to avoid this is by using split transactions. The idea is similar to how retried transactions work, but instead of the bus master retrying the transaction, the slave will automatically send the requested value back to the bus master using a special kind of bus transaction.

Besides the techniques outlined above, most buses also have some sort of error control. This can both be used to signal a link layer error (e.g parity error or CRC error) or that the current transaction is invalid in some way (e.g writing to a read only register).

9.1.3 Arbitration

If there is more than one potential bus master connected to a bus it is important to make sure that only bus master is granted access to the bus simultaneously. A popular way to solve this is to use an arbiter to which all bus masters are connected. If a bus master needs to access the bus, it first requests access to the bus from the arbiter which will grant access to only bus master at a time. If more than one bus master request access at the same time, a variety of algorithms can be used based on for example priorities or fairness.

It is also possible to statically schedule all bus transactions at design time. This is useful in real time systems where failure to meet a deadline (because of for example a busy bus) can be catastrophic. The drawback with static scheduling is that it is hard to analyze and schedule a complex system with many components and several buses.

9.1.4 Buses and Bridges

If a single bus does not provide enough performance for a certain system the system may be partitioned so that it contains more than one bus.

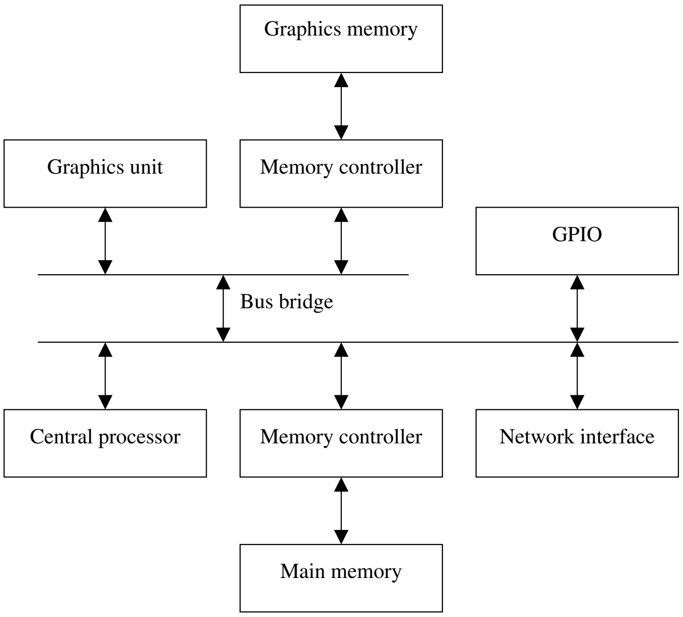


Figure 9.3: Partitioning a design into two buses

Consider for example the hypothetical system shown in Figure 9.3. The system is divided so that it contains two buses instead of one. One of the buses connects the graphics unit to the graphics memory and the other connects the CPU to the main memory and other peripherals. This includes a bus bridge that allows the CPU to access the graphics unit and its memory. The idea behind this division is that the CPU rarely needs to access graphics memory and the graphics unit rarely (or never) needs to access main memory. In this way, the accesses done by the graphics processor to show the screen are not noticed by the CPU except when accessing graphics memory and vice versa.

Since it is easier to create a fast bus if it is only connected to few components the system described in the previous paragraph will be easier to design and integrate than a system with similar performance utilizing only one bus.

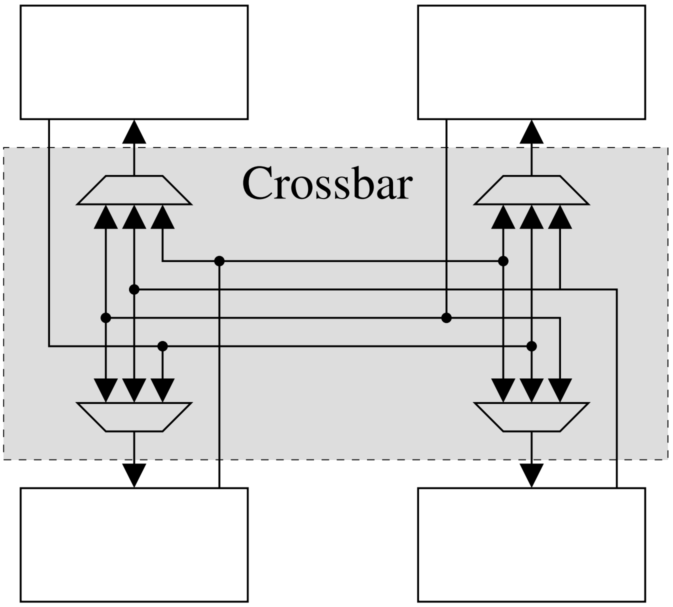


Figure 9.4: Crossbar implemented with muxes

9.1.5 Crossbars

In some situations it is not possible to divide a system as easily as the hypothetical system described above. In the worst case it would be common that all components communicate with all other components connected to the bus. If the bandwidth requirement is high enough in this case it will not be possible to use either a single bus or several buses.

In this situation the designer can use a crossbar instead of a bus. A crossbar is an interconnect component that will allow any input port to communicate with any output port. A crossbar can be constructed by using muxes, as seen in Figure 9.4.

The area cost for a crossbar is significantly higher than for a single bus. While the cost of a bus is roughly linear with the number of ports, the cost of a crossbar increases quadratically as more ports are added to it. Similarly to a simple bus, the latency of a crossbar will increase as more ports are added to it.

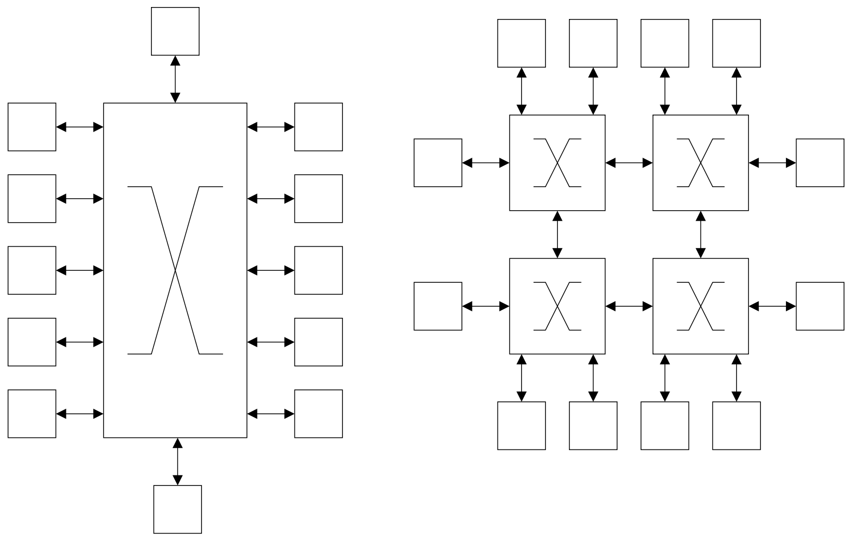


Figure 9.5: 12 modules connected using a crossbar and a distributed network

9.2 On Chip Networks

In an ideal world, a designer would be able to select a crossbar of a suitable size and connect all components to that. Unfortunately the cost of a large crossbar is prohibitive in most designs and compromises have to be made. One way to do this is basically an extension of Section 9.1.4. Instead of dividing a single bus into two separate buses, the system now uses many different crossbars and buses. The advantage is that it is easier to create a system with high throughput if individual crossbars are kept small. An illustration of this is seen in Figure 9.5. This is usually called Network-on-Chip (NoC) or sometimes On-Chip-Network (OCN).

9.2.1 Network Protocols

A network is substantially more complex than a bus and the protocols used on a network reflect that. Perhaps the most important decision to make when designing a network is to chose between a circuit switched and a packet switched network.

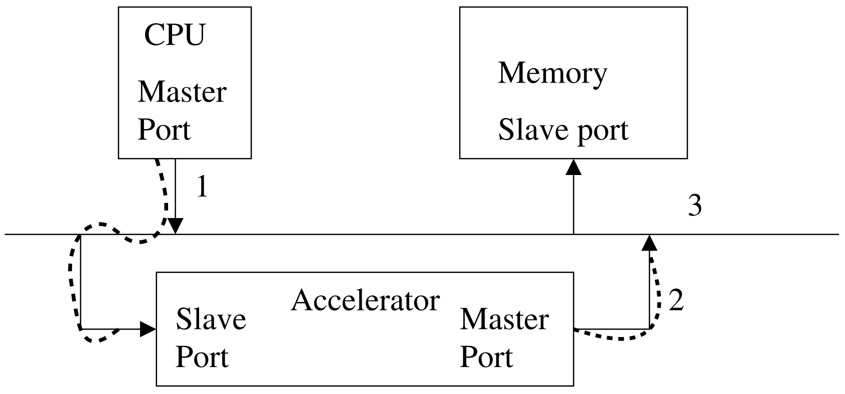
A circuit switched network is based on exclusive connections that are setup for a long period of time between different components. The archetypal example of circuit switching is the telephone system (although this is no longer completely true, especially as VoIP is getting more popular). The main advantage of such a network is that it is easy to implement. Once a connection is setup, each node along the route knows that the input from a certain port should always be sent directly to another port. The disadvantage is that a circuit switched network is often ineffective at using the available bandwidth.

Packet switching on the other hand is based on a system where a connection is setup for the duration of an incoming packet and tore down when the last part of that packet has been sent to the next switch. The main advantage is that it is much easier to utilize a communication link fully in a packet switched network because links are allocated to packets instead of to a connection. That way most allocations are short lived and the link can be reused for another connection immediately after a packet has been sent. The disadvantage is that it is necessary to buffer packets if a link is busy whereas a circuit switched NoC does not need any buffering once a connection has been established.

There are two ways to specify the destination address for a NoC packet. A NoC with source routing specifies the complete way that the packet will take in the NoC from the beginning. In a NoC with distributed routing only the destination address is sent to the NoC.

9.2.2 Deadlocks

A deadlock can be caused when there is a circular dependency on resources in a system. Consider a system where module X will first try to allocate resource on the first clock cycle a and then resource b in the second clock cycle while module Y will first try to allocate resource b and then resource a. Neither module will release a resource before it has allocated both resources. If X allocates a and Y allocates b, the system will



Step 1: CPU issues a read request to the accelerator
Step 2: Accelerator must read a value from memory to answer the read request
Step 3: Deadlock because the bus is already busy

Figure 9.6: Deadlock caused by a badly designed system

deadlock because neither module can release a resource before allocating the other resource.

An example of how a deadlock can appear in the context of a bus, Figure 9.6 shows a hypothetical system with a shared bus, a CPU acting as master, an accelerator acting as both slave and master and a memory acting as a slave. In the figure, the CPU tries to read from the accelerator, which causes the accelerator to try to access the bus. Since the bus is already used by the CPU, no further progress can be made. (The situation above could be solved if the accelerator could issue a "retry" to the CPU.)

Because a NoC is a distributed system there are many more opportunities for a deadlock to occur. By restricting the number of possible routes it is possible to create a NoC where it is impossible for a circular dependency to form. A well known method for this in a 2D mesh is X-Y routing. A complete discussion of routing algorithms is out of the scope of this thesis, the reader is instead referred to for example [60] for a in depth discussion.

However, simply using a deadlock free routing algorithm is not going to guarantee that the system is deadlock free. The example in Figure 9.6 shows a deadlock caused by the components connected to the bus and not the bus itself. Similarly, a NoC with deadlock free routing can still be part of a deadlock if the components connected to the bus are badly designed.

To avoid a deadlock in a NoC, a device connected to the NoC must be guaranteed to eventually accept an incoming packet for delivery. It doesn't have to accept it immediately, but the acceptance should not depend on being able to send a packet to the NoC.

9.2.3 Livelocks

A livelock situation is similar to a deadlock. If we modify the system described first in Section 9.2.2 so that a module automatically releases an allocated resource if it cannot allocate a required resource the following situation may occur:

- X allocates a, Y allocates b
- X fails to allocate b, Y fails to allocates a
- X releases a, Y releases b
- X allocates a, Y allocates b
- ...

In this case a livelock has occurred. Each module is continually doing something, but the system is not able to perform any real work.

Chapter 10
Network-on-Chip Architectures for FPGAs

Abstract: *In this chapter we will investigate the performance of Network on Chip architectures optimized for FPGAs. This is a challenging research problems because FPGAs are not very suitable for NoCs in the first place due to the high area cost of multiplexers. In this chapter the focus has been to create a highly optimized NoC architecture that is based on well understood principles. Both a circuit switched and a packet switched NoC were investigated although the packet switched NoC is probably a better fit for FPGAs than a circuit switched NoC. The maximum frequency of the packet switched NoC is 320 MHz in a Virtex-4 of the fastest speedgrade and the latency through an unloaded switch is 3 clock cycles. When compared with other publications this is a very good result.*

10.1 Introduction

At the Division of Computer Engineering we have a relatively long history of NoC research targeted to ASICs. The work described in this chapter is targeted at FPGAs instead while partially building on experiences gained from the SoCBUS [61] research project.

There are many challenges and opportunities in FPGA based NoC design. Many issues are identical or very similar to an ASIC based NoC, such as high level protocol and design partitioning. On the other hand, the architecture of an FPGA based NoC is limited by the FPGA whereas an ASIC based NoC can be optimized down to the layout of individual transistors in the most extreme case.

While early FPGAs were small enough to barely justify an on-chip bus, the largest FPGA today can fit a significant number of complex IP cores. One of the largest FPGAs available on the market today, the Virtex-4 LX200 has almost 180000 available 4 input look-up tables (LUTs). This can be compared to the resource usage of for example the Openrisc 1200 processor, which consumes around 5000 LUTs when synthesized to a Xilinx FPGA. Over 30 such processors or other IP cores of comparable complexity could fit into one such FPGA. It is only a matter of time before FPGAs of similar complexity becomes available at cost-effective prices. At that point, designs will need an efficient and scalable interconnection structure and many researchers think that Network-on-Chip research area will provide this structure.

When this case study was initiated, few FPGA based NoC seemed to exist that really pushed an FPGA to its limits. It therefore made sense to take a critical look at FPGAs to try to create an optimal match for NoC architecture and FPGA architecture. The goal was to optimize fairly simple architectures with well known behavior. In other words, this case study focuses on FPGA optimization techniques for NoCs instead of new and novel NoC architectures.

When the case study was initiated, statically scheduled NoCs were deliberately excluded from the study as they are quite easy to implement in an FPGA using for example the architecture in Figure 10.1. NoCs capable of handling dynamically changing traffic are more interesting except for specialized applications.

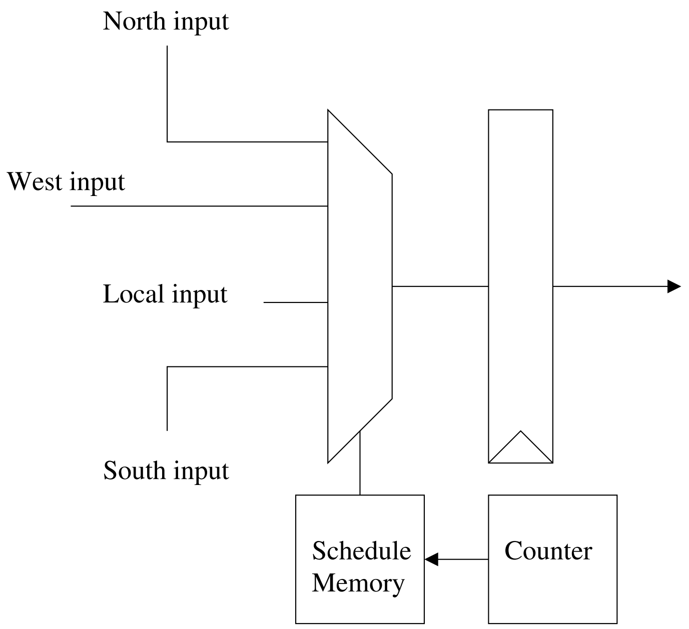


Figure 10.1: Minimalistic statically scheduled NoC switch

10.2 Buses and Crossbars in an FPGA

In Figure 10.2 a comparison is shown where the area and maximum frequency of a simple crossbar and a simple bus are shown for various number of ports. A Virtex-4, speedgrade 12 was used in this comparison. (Note that no modules are connected to this bus so Figure 10.2 shows the ideal case where the entire FPGA can be dedicated solely to the bus.)

It is no surprise that the maximum operating frequency of the components drop as more components are added, both for the bus and the crossbar. And of course, while not shown in the graph, the area of the crossbar grows extremely large as the number of ports are increased. It should be noted that neither the bus, nor the crossbar was pipelined in this comparison. Faster operation (at the expense of increased area) could be had by pipelining the bus/crossbar. This example is still valid for the majority of uses since many buses are not pipelined in practice.

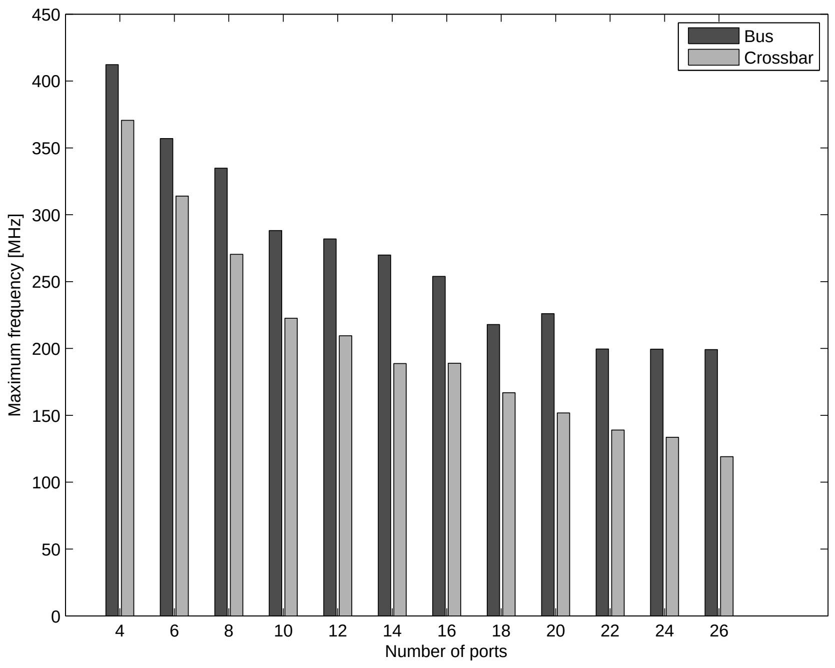


Figure 10.2: Maximum frequency for a bus and a crossbar with various number of ports

10.3 Typical IP Core Frequencies

To determine what frequency a NoC should operate at in order to be usable, we need to determine typical frequencies for the IP cores that will be connected to it. As an example of what to expect from typical IP cores, the maximum frequency of 40 cores when synthesized for a Virtex-4 were taken from the datasheets available at Xilinx IP Center [62]. This includes cores with a wide range of functionality including floating point, image coding, memory controller, and cryptographic cores. Extremely simple cores that are just wrappers around for example Block RAMs, DSP blocks, or distributed RAMs have been deliberately excluded in this figure as these cores are intended to be instantiated by a module that typically cannot achieve the same kind of operating frequencies as the

Maximum frequency	IP core	Maximum frequency	IP core
88	MD5	187	Floating point comparator
100	H.264 Encoder, Baseline	200	3GPP Turbo Encoder
107	XPS Ethernet Lite MAC	200	DDR SDRAM Controller
133	SHA-384, SHA-512	200	GFP
138	Modular Exponentiation Engine	200	MPEG-2 HDTV I & P Encoder
141	LIN Controller	200	MPEG-2 SDTV I & P Encoder
143	JPEG-C	200	SHA-1, SHA-256, MD5
148	JPEG-D	201	16550 UART w/ FIFO
148	JPEG-E	204	IPsec ESP Engine
162	IEEE 802.16e CTC Decoder	215	Tiny AES
165	Interleaver/Deinterleaver	225	3GPP Turbo Decoder
166	Floating point adder	225	H.264 Deblocker
166	Floating point divider	228	AES Fast Encryptor/Decryptor
166	MPEG-2 HDTV/SDTV Decoder	238	DDR SDRAM Controller
167	Floating point square root	250	AES-CCM
167	SDRAM Controller	255	Standard AES Encrypt/Decrypt
173	Integer to Floating point	256	PRNG
175	LZRW3 Data Compression	267	DDR2 SDRAM Controller
183	Floating point multiplier	292	IEEE 802.16e CTC Encoder
184	Floating point to integer	322	3GPP2 Turbo Encoder

Table 10.1: Maximum frequency of various Virtex-4 based cores

primitives them self. The numbers are summarized in Table 10.1. (Note that this include cores synthesized for both speedgrade -10, -11, and -12 Virtex-4 devices.)

While this data is by no means a complete survey of typical IP core frequencies it should at least give an indication of what to can expect from typical IP cores on a Virtex-4 based device. The data indicates that it is currently uncommon with a frequency of more than 250 MHz. Based on these numbers a NoC is therefore usable by most cores if it can operate at between 200 and 250 MHz. When compared with Figure 10.2, this means that a crossbar based solution will start to have problems when more than 10 typical IP cores are connected to it.

10.4 Choosing a NoC Configuration

As mentioned in Section 9.2.1, there are several protocols that can be used over a NoC. To determine their relative merits in an FPGA three different types of NoC switches for an FPGA were implemented:

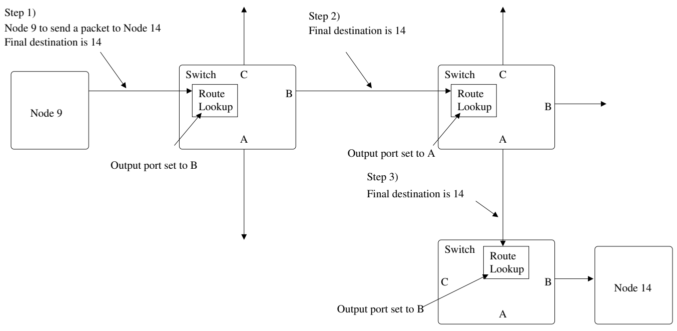
- Circuit switched
- Packet switched
- Minimalistic (no congestion/flow control)

The minimalistic NoC is included to get an idea of the maximum performance it is possible to get from a NoC optimized for an FPGA. It does not contain any sort of congestion control, if two words destined for the same output port arrive at a switch simultaneously one word will be ignored. While not very useful except for specialized applications, the clock frequency and area of this switch should be hard to improve on if distributed routing is used.

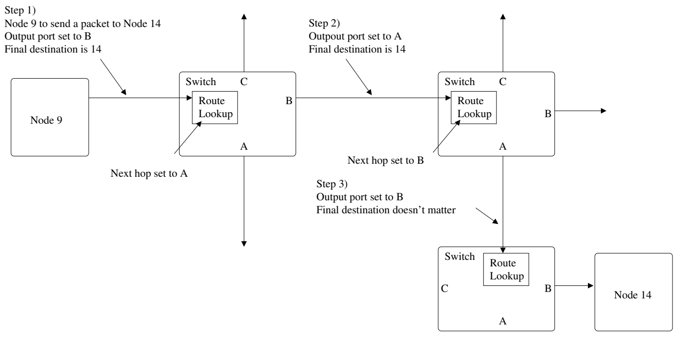
10.4.1 Hybrid Routing Mechanism

As mentioned earlier, there are two major kinds of routing mechanisms that can be used in a NoC, source routing and distributed routing. Source routing has the advantage that the routing decision in each switch is very simple. Distributed routing has the advantage that only the destination address is needed to determine the output port in a switch. Typically the route lookup in distributed routing will be part of the critical path of a switch. To retain most of the advantages of distributed routing while minimizing the drawback, a hybrid between source and distributed routing is used. A switch determines the output port for the next switch instead of the current switch. The output port of the current switch is directly available as a one-hot coded input signal. Figure 10.3 illustrates this compared to distributed routing.

In the proposed NoC, five signals are used to signal the final destination address. As a further optimization for the 2D mesh case, there is no



(a) A switch performs route lookup for the current output port



(b) A switch performs route lookup for the next switch

Figure 10.3: Route lookup as performed in the three NoCs

possibility for a message to be sent back to the same direction that it came from. Given this limitation and a maximum of 32 destination nodes, the route lookup tables can in theory handle any kind of topology with up to 32 destination nodes and any kind of deterministic routing algorithm. In practice a routing algorithm and topology that is deadlock free, such as the well known X-Y routing algorithm on a 2D mesh, should be used. (If the 32 destination nodes turns out to be a limitation it should be easy to extend this NoC to support more than 32 destination nodes if some sort of hierarchical addressing scheme is acceptable.)

In addition to the routing mechanism described above, similar signaling are used for all NoCs as shown in Table 10.2. The following sections

Table 10.2: Data and control signals in a unidirectional NoC link

Signals used in all three NoCs			
Signal name	Direction	Width	Description
Strobe	Sender to receiver	1	Qualifies a valid transaction
Data	Sender to receiver	36	Used as data signals
Last	Sender to receiver	1	Last data in transaction
Dest	Sender to receiver	5	Address of destination node
Route	Sender to receiver	3-4	Destination port on the switch (one hot coded)
Only in packet switched NoC			
Signal name	Direction	Width	Description
Ready	Receiver to sender	1	The remote node is ready to receive data
Only in circuit switched NoC			
Signal name	Direction	Width	Description
Nack	Receiver to sender	1	A connection setup was not successful
Ack	Receiver to sender	1	Acknowledges a successful connection

will describe each type of network in detail.

10.4.2 Packet Switched

The most complex part of this switch is the input part, which is shown in Figure 10.4. The FIFO is based on SRL16 primitives that allows a very compact 16 entry FIFO to be constructed. As the SRL16 has relatively slow outputs, a register is placed immediately after the SRL16. This means that the input part has a latency of two cycles in case the FIFO is empty and the output port is available.

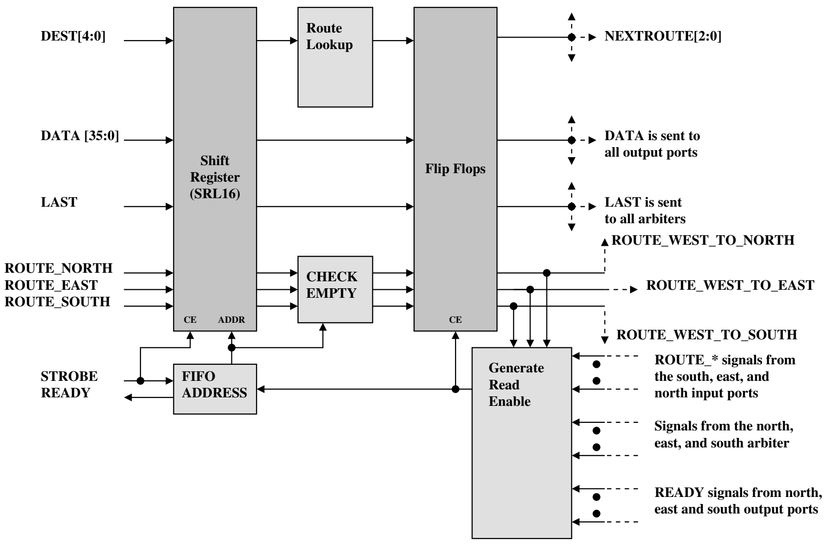
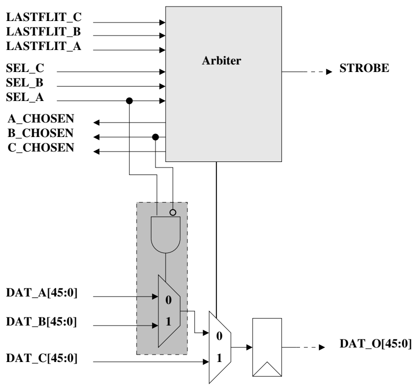


Figure 10.4: A detailed view of an input port of the packet switched NoC switch

The block named "check empty" makes sure that no spurious *ROUTE_** signals are sent to the arbiter if the FIFO is empty. By doing this, the arbiter will be simplified as compared to having both the *ROUTE_** signals and separate signals for *WEST_EMPTY*, *NORTH_EMPTY*, etc. In particular, it is easier to identify the case where only one input port needs to send a packet to the output port and send the packet immediately without any arbitration delay.

The block that generates the read enable signal to the input FIFO has to consider a large number of signals and it is therefore crucial to implement that block efficiently and place it so the routing delay is minimized. Through RLOC directives most of that logic can be placed into one CLB in order to minimize the routing delay. Finally, the *READY* signal is adjusted for pipeline latency so that the FIFO will not overflow if the sender does not stop sending as soon as *READY* goes low.



©2007 IEEE. Reprinted from Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on , *An FPGA Based Open Source Network-on-Chip Architecture*, Ehliar, A. Liu, D.

Figure 10.5: A view of the output part of the packet switched NoC switch

Figure 10.5 shows a detailed view of an output port of a 4-port switch. Each output port can only select from one of three input ports since there should be no need to route a packet back to where it came from in most topologies. If more than one input port needs to send a packet to the same output port, an arbiter in the output port uses round robin to select the port that may send. In the four port NoC switch, the output port is essentially a 3-to-1 mux controlled by the arbiter (or a 4-to-1 mux in the case of a five port switch). The *DAT_** signals are formed by combining the destination address with the *NEXTROUTE_** signals and the payload signal. It should also be noted that the part inside the dotted rectangle can be implemented inside one LUT for each wire in the *DAT_** signals which will reduce the delay for this part somewhat.

The latency of the switch when the input FIFO is empty and the output port is available is 3 clock cycles. If more than one input port has data for a certain output port, the latency is increased to 4 clock cycles due to an arbitration delay of one clock cycle for the packet that wins the arbitration.

There are two main critical paths in this switch. One path is caused by the read enable signal that is sent to the input FIFO. The other is from the

FIFO to the route look-up due to the slow output of the SRL16 elements.

10.4.3 Circuit Switched NoC

The circuit switched NoC has a similar design to the SoCBUS [61] network on chip architecture. The main difference between the circuit switched and the packet switched NoC is that there are no FIFOs in the input nodes. If the output port is occupied, a negative acknowledgment is instead sent back to the transmitter. In this case the transmitter has to reissue the connection request at a later time. Correspondingly, an acknowledgment is sent once the packet has reached the destination.

The overall design is similar to packet switched version with the exception of the input module. In the circuit switched switch there is no input FIFO as mentioned earlier. The arbiter is also different from the arbiter in the packet switched version. In particular, it has to arbitrate immediately if two or more connections arrive simultaneously to one output port. It does this by using a fixed priority for each input port.

The critical path of the circuit switched switch is the arbiter, which has to decide immediately if a circuit setup request should be accepted or rejected.

10.4.4 Minimal NoC

The main reason for including this architecture is to provide an upper bound on the achievable performance of an FPGA based NoC. Due to its low complexity it should be hard to create a NoC with distributed routing that can run at a higher frequency without making severe compromises on area and latency.

This NoC architecture does not use arbitration at all. If two words arrive at one output port at the same time, one of them is discarded. This means that such a network would have to be statically scheduled or use some other means of guaranteeing that messages do not collide if a lost or damaged message is unacceptable.

Table 10.3: Frequency/area of the different NoC architectures in different FPGAs

FPGA	Packet switched (4 port)	Packet switched (5 port)	Circuit switched (4 port)	Circuit switched (5 port)	No congestion control (4 port)
xc4vlx80-12	321 MHz	284 MHz	341 MHz	315 MHz	506 MHz
xc4vlx80-10	232 MHz	206 MHz	230 MHz	225 MHz	374 MHz
xc2vp30-7	267 MHz	235 MHz	284 MHz	264 MHz	390 MHz
xc2v6000-4	176 MHz	160 MHz	190 MHz	183 MHz	241 MHz
Resource utilization (In Virtex-4)					
LUTs	784	1070	633	828	396
Flip flops	448	572	452	595	368
Latency (cycles)	3	3	2	2	2

10.4.5 Comparing the NoC Architectures

The performance and area of the three different NoCs are shown in Table 10.3. The resource utilization of individual modules of the NoC switches can be found in Table 10.4. LUTs that were only used for route-thru are also included in these numbers. ISE 10.1 was used for synthesis and place and route. Note that the performance numbers in Table 10.3 is only for a single switch. The performance of the NoC will also be affected by the distance between the switches, but this will not be a huge problem since flip-flops are used on the both the inputs and outputs of the NoC switches. An experiments on a Virtex-4 SX35 has shown that a NoC with 12 nodes and 4 switches is not limited by the distance between the switches even though the switches were placed in different corners of the FPGA.

The clock frequencies of the packet switched and circuit switched net-

Switch type	Module type	LUTs	Flip-flops
5 ports, packet switched	Arbiter	29	8
	Input FIFO	92	64
	Output Mux	98	47
4 ports, packet switched	Arbiter	20	6
	Input FIFO	68	55
	Output Mux	96	47
4 ports, circuit switched	Arbiter	29	5
	Input Module	20	59
	Output Mux	98	47
5 ports, circuit switched	Arbiter	41	6
	Input Module	20	64
	Output Mux	101	47
4 ports, No congestion control	Input module	2	45
	Output mux	92	46

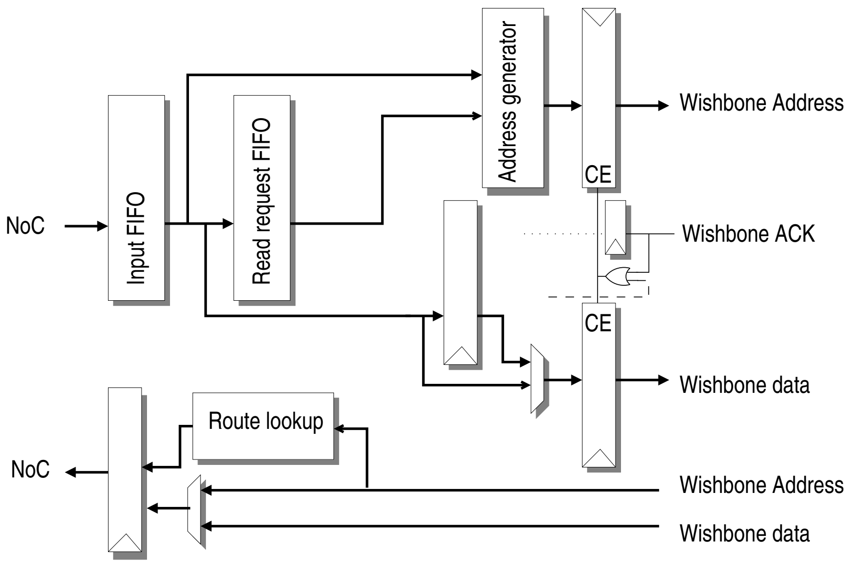
Table 10.4: The resource utilization of the individual parts of the switches

works are both high, although there is a relatively large gap to the upper limit established by the NoC without congestion control. A more efficient flow control mechanism would certainly be a welcome addition to these NoCs although inventing such an architecture is probably non-trivial.

Due to the small difference in performance between the circuit switched and packet switched network the packet switched network is probably the best fit for Xilinx FPGAs.

10.5 Wishbone to NoC Bridge

In addition to the NoC switches described above, a bridge between the Wishbone [63] bus and the packet switched NoC has been developed. The general architecture of the bridge is shown in Figure 10.6. A write requests issued to the bridge from the Wishbone bus is handled using



©2007 IEEE. Reprinted from Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on , *An FPGA Based Open Source Network-on-Chip Architecture*, Ehliar, A. Liu, D.

Figure 10.6: Simplified view of the data flow of the Wishbone to NoC bridge.

posted writes. That is, the write request is immediately acknowledged to the Wishbone master even though there will be a delay of several cycles before the write is guaranteed to reach its destination. A read request is handled by issuing retries for all read requests on the Wishbone bus until the requested value has been returned over the NoC. (This is very similar to how a PCI bridge works.)

To avoid deadlocks, read requests have lower priority than write requests and read replies. That is, the bridge will immediately service a write request in the input FIFO. The bridge will also service a read reply as soon as possible (by waiting for the originator of the read request to retry the read). On the other hand, if a read request comes in from the NoC downlink it cannot be serviced until the NoC uplink is available. This means that read requests has to be queued in a separate queue if the NoC uplink is not available (which will happen if the FIFO in the NoC switch the bridge is connected to is full). At the moment, the designer has to make sure that the read request queue is large enough to hold all possible incoming read requests that could be issued to a certain Wishbone bridge over the NoC.

A big problem with the Wishbone bus in the context of bus bridges is that the bus has been designed with a combinatorial bus in mind. While Wishbone does provide a couple of signals for burst handling, the only length indication for a linear burst is the fact that at least one more word is requested. To mitigate this, the bridge has an input signal that is used for reads to indicate the number of words to read, but this is no longer strictly Wishbone compliant.

Another area where the bridge is not fully Wishbone compatible is the error handling. It would be relatively easy to add support for the *ERR* signal to read requests/replies. Unfortunately, writes cannot be implemented using posted write requests if the *ERR* signal in Wishbone should be handled correctly. The easiest way to add error handling would be to add a status register that can be read by a master processor so that such errors can be detected by the operating system.

In our opinion, the complexity of the bridge is not a good sign. An interesting future research topic would be how to design a simple bus protocol which can both serve a bus at high performance while at the same time being easy to connect to a NoC.

10.6 Related Work

The ASIC community has been very active in the NoC research area. An early paper discussing the advantages of a NoC in an ASIC was written by Dally and Towles [64]. Other well known NoC architectures for ASICs include the Aethereal research project [65] and xpipes [66].

The FPGA community has not been quite as active but recently a number of publications have appeared. There seems to be a lot of interest in NoCs as a way to connect dynamically reconfigurable IP cores or even using dynamic reconfiguration on the NoC itself [67].

Bartic et al describes a packet switched network on a Virtex-II Pro FPGA [68]. Another packet switched network is described by Nachiket et al and compared to a statically scheduled networked on a Virtex-II 6000 [69]. There are also circuit switched FPGA based networks such as PNoC [70] which has also been studied in the context of an application and compared to a system with a shared bus.

Recently, a number of high speed NoCs have been presented such as MoCReS [71]. This is a packet switched NoC with support for both virtual channels and different clock domains with a reported performance of up to 357 MHz in a Virtex-4 LX 100. MoCReS seems to primarily utilize a BlockRAM per link which makes a much more expensive solution than the solution presented in this paper. The latency of a single switch is also not reported in the paper. Another recent publication is [72], in which the author describes a NoC intended for distributed and safety-critical real time systems which is pseudo-statically scheduled and makes heavy use of Cyclone II's 4kbit embedded memories.

10.7 Availability

The source code for the packet switched NoC can be downloaded at `http://www.da.isy.liu.se/research/soc/fpganoc/`. The Wishbone to NoC bridge is also available for download. Hopefully this will allow NoC researchers interested in FPGAs to easily compare their NoC against another NoC with good performance in an FPGA.

10.8 ASIC Ports

By using the compatibility library described in Section 5.2 it was possible to port the NoC to a 130nm ASIC process. Table 10.5 shows the performance and area of various NoC configurations and compares it to the performance of the FPGA versions.

Technology	Configuration	F_{max}	Area
xc4vlx80-12	4 port packet switched	320	784 LUT, 448 FF
130nm ASIC (speed optimized)	4 port packet switched	705	$0.20mm^2$
130nm ASIC (area optimized)	4 port packet switched	89	$0.18mm^2$
xc4vlx80-12	5 port packet switched	284	1070 LUT, 572 FF
130nm ASIC (speed optimized)	5 port packet switched	599	$0.26mm^2$
130nm ASIC (area optimized)	5 port packet switched	84	$0.24mm^2$

Table 10.5: Performance of packet switched NoC in different technologies

It is interesting that the critical path is still in the read enable signal to the FIFOs in the input ports even in the ASIC version of the packet switched switch. However, as can be seen in the table there is little possibility to improve the ASIC timing by trading area for frequency. This is not surprising as the switch consists mostly of muxes and flip-flops and there is little the synthesizer can do about these.

The difference between the packet switched NoC switch and circuit switched NoC switch is substantial in the ASIC port. This is because the packet switched switch is using many SRL16 primitives. While this primitive is very cost effective in an FPGA as it allows a LUT to be used as a 16 bit shift register, it is likely to be expensive to port this to an ASIC. In fact, since a circuit switched NoC is so much cheaper it is actually possible to use a much more complex network with more nodes in it if circuit switching is used instead of packet switching. Doubling the number of switches in the network is not a problem area wise. In fact, it is possible to both double the number of switches and the width of the links and still use less area than the packet switched network in an ASIC

Technology	Configuration	F_{max}	Area
xc4vlx80-12	4 port circuit switched	341	633 LUT, 452 FF
130nm ASIC (speed optimized)	4 port circuit switched	948	$0.025mm^2$
130nm ASIC (area optimized)	4 port circuit switched	259	$0.023mm^2$
xc4vlx80-12	5 port circuit switched	315	828 LUT, 595 FF
130nm ASIC (speed optimized)	5 port circuit switched	846	$0.038mm^2$
130nm ASIC (area optimized)	5 port circuit switched	232	$0.032mm^2$

Table 10.6: Performance of circuit switched NoC in different technologies

when using these components.

10.9 Conclusions

It is possible to create high speed NoC switches on a Xilinx FPGA that are both fast and relatively small. By manually instantiating FPGA primitives it is possible to achieve the level of control which is needed to reach the highest performance. Floorplanning is not a requirement to reach this performance, but investigating the output from the placer was necessary to understand how the design could be further optimized at many times during the development.

In our experience, circuit switched and packet switched NoCs will have roughly the same operating frequency and area in Xilinx devices and the developer is therefore free to chose which to use depending on his or her needs. However, if the design might eventually be ported to an ASIC, the packet switched NoC will be much more expensive in terms of area than the circuit switched NoC. In fact, in terms of area, a packet switched NoC is more than 5 times as expensive as a circuit switched NoC. The maximum frequency of a circuit switched node is also slightly higher than for a packet switched node which is another advantage of the circuit switched network in an ASIC.

However, it is possible that once Network-on-Chip reaches mainstream acceptance in the ASIC community it will be possible to buy hard IP blocks with a NoC switch created using full custom methods. Under this assumption it will probably become natural to replace FPGA based NoC switches with optimized ASIC versions in the same way that block RAMs are replaced with custom memory blocks when porting a design to an ASIC.

However, it is not so easy to interface a NoC to a normal bus. An interesting future research area would be to design a simple bus protocol that has a high performance on a regular bus while still being easy to interface to a high speed NoC.

Part IV

Custom FPGA Backend Tools

Chapter 11
FPGA Backend Tools

Abstract: *Sometimes a designer encounters a situation where the FPGA vendor's tool is not quite good enough. Of course, in most cases the existing tools are adequate, even if they are not optimal by any means. However, sometimes there are situations where a designer would really like a little more control over the backend part of the design flow. This chapter is intended to serve as an inspiration for those who would like to write their own backend tools for the Xilinx design flow.*

11.1 Introduction

XDL is a file format which contains a text version of Xilinx' proprietary NCD file format. (The NCD file format is used for netlists created by both the mapper and the place and route tool.) The *xdl* command can be used to convert between XDL and NCD. The XDL file format is no longer documented by Xilinx, but earlier version of ISE contained some information about it [73].

Due to the simplicity of the file format it is quite easy to parse it in a custom program or script. Unfortunately it is difficult to understand the part which deals with routing as those parts require knowledge about the FPGA which is difficult to obtain.

It is also possible to modify a netlist in XDL format to include or change the functionality. A common usage for this scenario is debugging. ChipScope [74] is a logic analyzer developed by Xilinx which can be inserted into an FPGA without having to resynthesize/place and route the entire design.

A typical design flow for a backend tool utilizing the XDL file format is shown in Figure 11.1. It is important to note that merely modifying the XDL file is not enough. It is also necessary to modify the PCF constraints file to avoid unspecified timing paths.

11.2 Related Work

A number of different ways to modify a design after synthesis has been implemented. As already mentioned, Xilinx has their own tool which allow a logic analyzer to be inserted [74].

At one point, Xilinx also distributed jbits [75], which allows a user to manipulate a design in Java. Sadly, jbits has been discontinued and doesn't support any design newer than a Virtex-II.

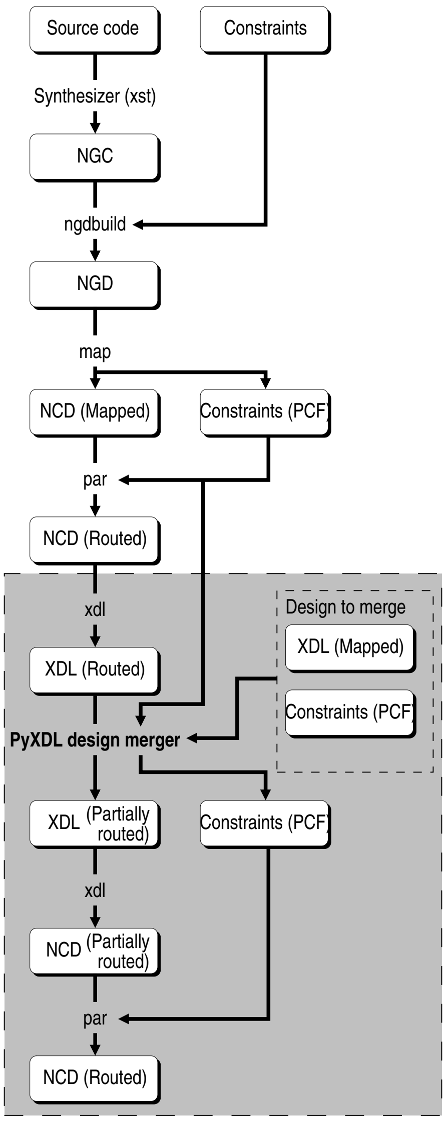
Of course, it is also possible to use the fpga_editor to modify or inspect a design. Unfortunately, this is a limited method as there is no general purpose script language in the FPGA editor.

A final tool of interest is abits [76], which allows an Atmel bitstream to be manipulated.

11.3 PyXDL

PyXDL is a library designed by us for reading and writing XDL files. While somewhat limited at the moment, it has three demonstration programs:

- Design viewer
- Resource usage viewer
- Logic analyzer inserter



©2007 FPGAWorld.com. Reprinted from 4th annual FPGAworld Conference, *Thinking outside the flow: Creating customized backend tools for Xilinx based designs*, Ehliar, A. Liu, D.

Figure 11.1: Typical design flow when utilizing the XDL file format

Of these, the logic analyzer inserter is probably the most useful. It inserts a logic analyzer core into an already placed and routed design and allows most signals to be probed. (Some signals cannot easily be probed e.g. carry chains.). After insertion of the logic analyzer, the core can be controlled via a serial port.

For those who are interested, PyXDL is available under the GPL at `http://www.da.isy.liu.se/~ehliar/pyxdl/` together with the sample applications listed above.

11.4 Future Work

While the current version of PyXDL and its demo applications is limited, the concept is very interesting. An obvious improvement would be to extend the logic analyzer. Right now it is hard coded for a certain channel width and memory depth. It is desirable that these values are configurable at runtime instead.

Another improvement could be to add other types of instrumentation to the logic analyzer core. Statistics gathering (possibly with some interpreter for typical buses) would be easy to implement.

Some more interesting uses relates to partial reconfiguration. With XDL it is easy to replace part of a design with another part. A most interesting use of XDL/Partial reconfiguration would be if a tool could be created which automatically divided a large design into several parts which are loaded into the FPGA on an as-needed basis using partial reconfiguration.

Part V

Conclusions and Future Work

Chapter 12
Conclusions

Optimizing a design for a certain platform will always include trade-offs between many parameters such as performance, area, flexibility, and development time. The case studies in this thesis was developed with the intention of aiming for very high performance without sacrificing too much flexibility and area. In all of these case studies, careful high and low level optimizations allowed the designs to reach very high clock frequencies.

12.1 Successful Case Studies

The first case study shows that a very high clock frequency can be achieved in an FPGA based soft processor without resorting to a huge number of pipeline stages. By optimizing the execution units, in particular the arithmetic unit, it is possible to forward a result calculated in the arithmetic unit immediately to itself without making any sacrifices regarding the maximum clock frequency. The processor can operate at a frequency of 357 MHz in a Virtex-4 of the fastest speed grade which is significantly higher than other soft processors for FPGAs. To achieve this frequency, both high and low-level optimizations had to be used, including manual instantiation of LUTs and manual floorplanning of the critical parts of the processor.

The next case study studied floating point adders and multipliers and showed how these can be optimized for the Virtex-4. By parallelizing the normalizer we could achieve a clock frequency of 370 MHz in the floating point adder with a latency of 8 clock cycles for a complete addition. This is faster than previous publications at the same latency in terms of clock cycles but this speed comes at a price of higher area than previous publications as well. Our floating point units should be a good match for situations where low latency is important

The final case study discusses how NoC architectures can be optimized for FPGAs. We find that a circuit switched network will be smaller than a packet switched network, but the difference is relatively small on Xilinx FPGAs when using the SRL16 primitive. This means that a packet switched network is very attractive to use in an FPGA.

12.2 Porting FPGA Optimized Designs to ASICs

Since all of the designs in this thesis depends on being able to instantiate FPGA primitives the designs could not be directly ported to an ASIC. However, writing a small compatibility library with synthesizable versions of flip-flop and other slice primitives allows the designs to be be easily ported to an ASIC. When ported directly using such a library the performance of the designs are adequate in the ASIC. However, by modifying the designs slightly it was possible to increase the clock frequency significantly. In both the processor and the floating point modules it was necessary to replace the DSP48 based multiplier with a version using a multiplier from Synopsys' DesignWare library. In the adder it was also necessary to replace the adder based on instantiated FPGA primitives with a behavioral version.

These were small changes that could be performed quickly and most of the FPGA specific optimizations are still present, including some adders consisting of instantiated FPGA primitives. This shows that FPGA optimizations need not be an obstacle to a high performance ASIC port.

Chapter 13
Future Work

As with most other research projects, the designs described in this thesis can not yet claim that they are finished. There are many interesting possibilities for future research here and the most important of these will be described in this chapter.

13.1 FPGA Optimized DSP

The soft processor described in this thesis has a promising architecture. However, it does lack some features that are necessary for massive deployment. Most importantly, it does not currently have a compiler. Creating a basic backend for GCC should be a fairly easy task although getting GCC to automatically use the DSP features of the processor will be harder.

Another challenging problem is how to implement caches with very high clock frequencies yet low enough latency to be suitable for this processor. This is not only a matter of creating a fast cache, integrating it into the processor is also not a trivial problem as a good strategy for handling cache misses, particularly in the data cache, has to be invented. (Cache misses during instruction fetch are easier to handle as they are visible much earlier in the pipeline.) Caches are also more or less a necessity if the address space should be increased from 16-bits to 32-bits.

The instruction set should be benchmarked thoroughly to determine if there are any important instructions that are missing by using standardized benchmarks.

Finally, there are some more minor details that could be added to the processor without too much difficulty. Interrupts could be added fairly easily if it is acceptable that a few clock cycles are spent to make sure that the pipeline is not executing a delayed jump. If some sort of improved branch prediction is used, it may be possible to avoid the use of delay slots. Automatic stalling of the processor due to hazards would also be a nice addition and could probably be implemented using extra bits in the instruction word (these bits may only have to be present in the instruction cache and not in main memory).

13.2 Floating Point Arithmetic on FPGAs

While the floating point adder and multiplier are probably the most polished projects described in this thesis there is still much that could be done in this research area. The most obvious improvement is to make the units more flexible by allowing parameters to be used to specify mantissa width and exponent width. This is not so interesting from a research perspective but very important from a practical perspective.

An interesting problem is how to create a fast floating point MAC unit. Right now, the units are not really optimal for this since the result of an addition is available after eight cycles, which will lead to some problems when trying to perform a MAC operation. A possible way to solve this is to perform eight or more convolution operations simultaneously but a better solution would be to create a MAC unit capable of accumulating one floating point value each cycle. How to do this in an FPGA is far from obvious if high performance is desired, especially if full IEEE-754 compliance is required.

13.3 Network-on-Chip

The NoC research area is still far from mature so there is obviously much to do here. In the implementation described in this thesis, the most important improvement is probably to improve the bus bridge by for example reducing the area and allowing different clocks to be used on the NoC and on the Wishbone bus.

Another interesting area is NoC friendly bus protocols as many of todays bus protocols are not very suitable when a pipelined NoC (or bus for that matter) is used.

Finally, adding some support for quality of service would be a nice addition to this NoC although it is not clear if this can be done without huge area penalties.

13.4 Backend Tools

The PyXDL package is already useful for a few tasks, but it is mainly intended to serve as an inspiration for other researchers to show that it is pretty simple to manipulate the netlists generated by Xilinx software. It would only be a matter of some programming to add support for more Xilinx FPGAs to PyXDL, improving the logic analyzer inserter, and creating more tools similar to the logic analyzer like statistics gathering.

However, a much more interesting research direction would be to create a toolchain that allows for automatic creation of partially dynamically reconfigurable designs. A stable tool which allows this to be done could be a huge boon to the FPGA community.

13.5 ASIC Friendly FPGA Designs

Further work is needed on how to design systems so that they are efficient in both ASIC and FPGAs. While the data in this thesis indicates that an FPGA optimized architecture is overall pretty easy to do with only small changes to the designs, more work is required in this area, especially to determine how the power consumption depends on the FPGA optimization. If structured ASICs increase in popularity it would also be interesting to determine the impact of FPGA optimizations when porting an FPGA design to a structured ASIC.

Bibliography

[1] D. Selwood, "Ip for complex fpgas," *FPGA and Structured ASIC Journal*, 2008. [Online]. Available: http://www.fpgajournal.com/articles_2008/20081209_ip.htm

[2] S. Singh, "Death of the rloc?" *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pp. 145–152, 2000.

[3] Ray Andraka, private communication, 2009.

[4] M. Keating and P. Bricaud, *Reuse Methodology Manual for System-On-A-Chip Designs*. Kluwer Academic Publishers, 2002.

[5] D. Liu, *Embedded DSP Processor Design: application specific instruction set processors*. Elsevier Inc., Morgan Kaufmann Publishers, 2008, ch. 4 DSP ASIP Design Flow.

[6] J. Stephenson, "Design guidelines for optimal results in high-density fpgas," in *Design & Verification Conference*, 2003. [Online]. Available: http://www.altera.com/literature/cp/fpgas-optimal-results-396.pdf

[7] Altera, *Guidance for Accurately Benchmarking FPGAs v1.2*, 12 2007. [Online]. Available: http://www.altera.com/literature/wp/wp-01040.pdf

[8] K.-C. Wu and Y.-W. Tsai, "Structured asic, evolution or revolution?" in *ISPD '04: Proceedings of the 2004 international symposium on Physical design*. New York, NY, USA: ACM, 2004, pp. 103–106.

[9] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," in *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2007.

[10] Actel, "Igloo low-power flash fpgas handbook," 2008. [Online]. Available: http://www.actel.com/documents/IGLOO_HB.pdf

[11] T. Tuan, S. Kao, A. Rahman, S. Das, and S. Trimberger, "A 90nm low-power fpga for battery-powered applications," in *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, 2006, pp. 3–11.

[12] D. G. Chinnery and K. Keutzer, "Closing the power gap between asic and custom: an asic perspective," in *DAC '05: Proceedings of the 42nd annual conference on Design automation*. New York, NY, USA: ACM, 2005, pp. 275–280.

[13] T. Savell, "The emu10k1 digital audio processor," *Micro, IEEE*, vol. 19, no. 2, pp. 49–57, Mar/Apr 1999.

[14] Xilinx, "Easypath fpgas." [Online]. Available: http://www.xilinx.com/products/easypath/index.htm

[15] Altera, "Asic, asics, hardcopy asics with transceivers." [Online]. Available: http://www.altera.com/products/devices/hardcopy-asics/about/hrd-index.html

[16] Pat Mead, private communication, 2008.

[17] eASIC, "nextreme zero mask-charge new asics." [Online]. Available: http://www.easic.com/pdf/asic/nextreme_asic_structured_asic.pdf

[18] L. S. Corporation, "Maco: On-chip structured asic blocks," 2009. [Online]. Available: http://www.latticesemi.com/products/fpga/sc/macoonchipstructuredasicb/

[19] L. Wirbel, “Fpga survey sees sunset for gate arrays, continued dominance by xilinx, altera,” *EE Times*, 2008. [Online]. Available: http://www.eetimes.com/miu/showArticle.jhtml;jsessionid=QM4Y35UD5BX3AQSNDLPSKHSCJUNN2JVN?articleID=211200184

[20] STMicroelectronics, “Methodology & design tools.” [Online]. Available: http://www.st.com/stonline/products/technologies/asic/method.htm

[21] C. Baldwin. Converting fpga designs. [Online]. Available: http://www.chipdesignmag.com/display.php?articleId=2545

[22] *Application Note 311: Standard Cell ASIC to FPGA Design Methodology and Guidelines ver 3.0*, Altera, 2008.

[23] K. Goldblatt, *XAPP119: Adapting ASIC Designs for Use with Spartan FPGAs*, Xilinx, 1998.

[24] Xilinx, “Recorded lectures: Asic user.” [Online]. Available: http://www.xilinx.com/support/training/rel/asic-user-rel.htm

[25] M. Hutton, R. Yuan, J. Schleicher, G. Baeckler, S. Cheung, K. K. Chua, and H. K. Phoon, “A methodology for fpga to structured-asic synthesis and verification,” *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, vol. 2, pp. 1–6, March 2006.

[26] T. Danzer. (2006) Low-cost asic conversion targets consumer success. [Online]. Available: http://www.fpgajournal.com/articles_2006/20061107_ami.htm

[27] J. Gallagher and D. Locke. (2004, 3) Build complex asics without asic design expertise, expensive tools. [Online]. Available: http://electronicdesign.com/Articles/Print.cfm?AD=1&ArticleID=7382

[28] Ian Kuon, private communication, 2009.

[29] P. Metzgen. (2004) Optimizing a high performance 32-bit processor for programmable logic. [Online]. Available: http://www.cs.tut.fi/soc/Metzgen04.pdf

[30] G. Bilski, "Re: [fpga-cpu] paul metzgen on multiplexers and the nios ii pipeline," 7 2007. [Online]. Available: http://tech.groups.yahoo.com/group/fpga-cpu/message/2795

[31] Xilinx, *XtremeDSP for Virtex-4 FPGAs User Guide UG073 (v2.7)*, 2008. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug073.pdf

[32] O. Gustafsson, A. G. Dempster, K. Johansson, M. D. Macleod, and L. Wanhammar, "Simplified design of constant coefficient multipliers," *Circuits, Systems and Signal Processing*, vol. 25, no. 2, pp. 225–251, 2006.

[33] M. J. Wirthlin, "Constant coefficient multiplication using look-up tables," *J. VLSI Signal Process. Syst.*, vol. 36, no. 1, pp. 7–15, 2004.

[34] Atmel, *ATC35 Summary*. [Online]. Available: http://www.atmel.com/dyn/resources/prod_documents/1063s.pdf

[35] D. Drako and H.-T. A. Yu, "Apparatus for alternatively accessing single port random access memories to implement dual port first-in first-out memory," U.S. Patent 5 371 877, 12 6, 1994.

[36] B. Flachs, S. Asano, S. Dhong, H. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H.-J. Oh, S. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, D. Brokenshire, M. Peyravian, V. To, and E. Iwata, "The microarchitecture of the synergistic processor for a cell processor," *Solid-State Circuits, IEEE Journal of*, vol. 41, no. 1, pp. 63–70, Jan. 2006.

[37] N. Sawyer and M. Defossez, "Xapp228 (v1.0) quad-port memories in virtex devices," 2002. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp228.pdf

[38] K. J. McGrath and J. K. Pickett, "Microcode patch device and method for patching microcode using match registers and patch routines," U.S. Patent 6 438 664, 8 20, 2002.

[39] "Opencores.org." [Online]. Available: http://www.opencores.org/

[40] M. Olausson, A. Ehliar, J. Eilert, and D. Liu, "Reduced floating point for mpeg1/2 layer iii decoding," *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, vol. 5, pp. V–209–12 vol.5, May 2004.

[41] Spirit DSP, "Datasheet: Spirit mp3 decoder," 2009. [Online]. Available: http://www.spiritdsp.com/products/audio_engine/audio_codecs/mp3/

[42] P. Ahuja, D. Clark, and A. Rogers, "The performance impact of incomplete bypassing in processor pipelines," *Microarchitecture, 1995. Proceedings of the 28th Annual International Symposium on*, pp. 36–45, Nov-1 Dec 1995.

[43] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill, "Mips: A microprocessor architecture," *SIGMICRO Newsl.*, vol. 13, no. 4, pp. 17–22, 1982.

[44] J. Gray, "Building a risc system in an fpga part 2," *Circuit Cellar*, vol. 117, 2000.

[45] Xilinx, *MicroBlaze Processor Reference Guide UG081 (v9.0)*, 2008. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf

[46] Altera, *Nios II Processor Reference Handbook*, Internet, 2007.

[47] Lattice, *LatticeMico32 Processor Reference Manual*, Internet, 2007.

[48] J. Nurmi, *Processor Design - System-On-Chip Computing for ASICs and FPGAs*. Springer, 2007.

[49] P. Metzgen, "A high performance 32-bit alu for programmable logic," in *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 2004, pp. 61–70.

[50] G. Research, *The LEON processor user's manual*, Internet, 2001.

[51] D. Lampret, *OpenRISC 1200 IP Core Specification*, 2001.

[52] Göran Bilski, private communication, 2008.

[53] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna, "Analysis of high-performance floating-point arithmetic on fpgas," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004, pp. 149+. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1303135

[54] R. Andraka, "Supercharge your dsp with ultra-fast floating-point ffts," *DSP magazine*, no. 3, pp. 42–44, 2007. [Online]. Available: http://www.xilinx.com/publications/magazines/dsp_03/xc_pdf/p42-44-3dsp-andraka.pdf

[55] Xilinx, *Floating-Point Operator v3.0*, 3rd ed., Internet, Xilinx, www.xilinx.com, September 2006.

[56] Nallatech, *Nallatech Floating Point Cores*, Internet, Nallatech, www.nallatech.com, 2002.

[57] J. Detrey and F. de Dinechin, "A parameterized floating-point exponential function for fpgas," in *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, 2005, pp. 27–34. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1568520

[58] J. Eilert, A. Ehliar, and D. Liu, "Using low precision floating point numbers to reduce memory cost for mp3 decoding," *Multimedia Signal Processing, 2004 IEEE 6th Workshop on*, pp. 119–122, 29 Sept.-1 Oct. 2004.

[59] R. Rojas, “Konrad Zuse’s legacy: The architecture of the z1 and z3,” *IEEE Annals of the history of computing*, vol. 19, 1997.

[60] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

[61] D. Wiklund and D. Liu, “Design of a system-on-chip switched network and its design support,” *Communications, Circuits and Systems and West Sino Expositions, IEEE 2002 International Conference on*, vol. 2, no. 29, 2002.

[62] Xilinx, “Xilinx ip center.” [Online]. Available: http://www.xilinx.com/ipcenter/index.htm

[63] “Wishbone system-on-chip (soc) interconnection architecture for portable ip cores,” 2002. [Online]. Available: http://www.opencores.org/

[64] W. J. Dally and B. Towles, “Route packets, not wires: On-chip interconnection networks,” in *Design Automation Conference*, 2001, pp. 684–689. [Online]. Available: citeseer.ist.psu.edu/dally01route.html

[65] K. Goossens, J. Dielissen, and A. Radulescu, “Aethereal network on chip: concepts, architectures, and implementations,” *Design & Test of Computers, IEEE*, vol. 22, 2005.

[66] D. Bertozzi and L. Benini, “Xpipes: a network-on-chip architecture for gigascale systems-on-chip,” *Circuits and Systems Magazine, IEEE*, vol. 4, 2004.

[67] L. Braun, M. Hubner, J. Becker, T. Perschke, V. Schatz, and S. Bach, “Circuit switched run-time adaptive network-on-chip for image processing applications,” *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 688–691, 27-29 Aug. 2007.

[68] T. Bartic, J.-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins, “Highly scalable network on chip for reconfigurable systemsin,” *System-on-Chip, 2003. Proceedings. International Symposium on*, 2003.

[69] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon, “Packet switched vs. time multiplexed fpga overlay networks,” *IEEE Symposium on Field-programmable Custom Computing Machines*, 2006.

[70] C. Hilton and B. Nelson, “Pnoc: a flexible circuit-switched noc for fpga-based systems,” *Computers and Digital Techniques, IEE Proceedings-*, vol. 153, 2006.

[71] A. Janarthanan, V. Swaminathan, and K. Tomko, “Mocres: an area-efficient multi-clock on-chip network for reconfigurable systems,” *VLSI, 2007. ISVLSI ’07. IEEE Computer Society Annual Symposium on*, pp. 455–456, 9-11 March 2007.

[72] M. Schoeberl, “A time-triggered network-on-chip,” *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 377–382, 27-29 Aug. 2007.

[73] Xilinx, “Xilinx design language,” *help/data/xdl/xdl.html in ISE 6.3*, 2000.

[74] ——, “Chipscope pro.” [Online]. Available: http://www.xilinx.com/ise/optional_prod/cspro.htm

[75] ——, “Jbits sdk.” [Online]. Available: http://www.xilinx.com/products/jbits/

[76] A. Megacz, “A library and platform for fpga bitstream manipulation,” *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’07)*, 2007.

Appendix A: ASIC Porting Guidelines

- Adders with more than two inputs are typically more area efficient in ASICs than in FPGAs. (See Section 5.5)
- While multiplexers are very costly in an FPGA, they are quite cheap in an ASIC. Optimizing the mux structure in the FPGA based design will have little impact on an ASIC port. (See Section 5.6)
- When porting an FPGA optimized design to an ASIC it may be possible to increase the performance of the ASIC by adding muxes to strategic locations such as for example by replacing a bus with a crossbar. (See Section 5.6)
- While a lot of performance and area can be gained in an FPGA by merging as much functionality into one LUT as possible, this will typically not decrease the area cost or increase the performance of an ASIC port. (See Section 5.7)
- If a design is specifically optimized for the DSP blocks in an FPGA the ASIC port is likely to have performance problems. The datapath with multipliers may have to be completely rewritten to correct this. (See Section 5.8)
- Create wrapper modules for memory modules so that only the wrappers have to be changed when porting the design to a new

technology. (See Section 5.9)

- Avoid large dual port memories if it is possible to do so without expensive redesigns. (See Section 5.9.1)

- If many small register file memories with only one write port and few read-ports are used in a design, the area cost for an ASIC port will be relatively high compared to the area cost of the FPGA version. On the other hand, if more than one write port is required, the ASIC port will probably be much more area efficient. (See Section 5.9.2)

- Design Guideline: If some of the memories can be created with a ROM-compiler, the area savings in an ASIC port will be substantial. (See Section 5.9.3)

- Avoid relying on initialization of RAM memories at configuration time in the FPGA version of a design. (See Section 5.9.4)

- When porting a design to an ASIC, consider if specialized memories like CAM memories can give significant area savings or performance boosts. (See Section 5.9.5)

- Consider if special memory options that are unavailable in FPGAs, like write masks, can improve the design in any way. (See Section 5.9.5)

- It is possible to port a design with instantiated FPGA primitives using a small compatibility library. For adders and subtracters, the performance will be adequate unless they are a part of the critical path in the ASIC. If this approach is used it is imperative that the modules with instantiated FPGA primitives are flattened during synthesis and before the optimization phase! (See Section 5.10)

- Manual floorplanning of an FPGA design will not have any impact on an ASIC port unless the design is modified to simplify floorplanning in the FPGA. (See Section 5.11)

- While pipelining an FPGA design will certainly not hurt the maximum frequency of an ASIC, the area of the ASIC will often be slightly larger than necessary, especially if the pipeline is not a part of the critical path in the ASIC. Designs that contains huge number of delay registers will be especially vulnerable to such area inefficiency. (See Section 5.12)