Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Improving and Extending a High Performance Processor Optimized for FPGAs

Examensarbete utfört i Datorteknik
av

Kristoffer Hultenius och Daniel Källming

LiTH-ISY-EX--10/4316--SE

Linköping 2010

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

Improving and Extending a High Performance Processor Optimized for FPGAs

Examensarbete utfört i Datorteknik
av

Kristoffer Hultenius och Daniel Källming

LiTH-ISY-EX--10/4316--SE

Handledare: **Andreas Ehliar**
ISY, Linköpings universitet

Examinator: **Andreas Ehliar**
ISY, Linköpings universitet

Linköping, 11 June, 2010

Avdelning, Institution
Division, Department

Division of Computer Engineering
Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Datum
Date

2010-06-11

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
__

ISRN
LiTH-ISY-EX--10/4316--SE

Serietitel och serienummer **ISSN**
Title of series, numbering __

URL för elektronisk version

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-56751

Titel
Title

Förbättring och utökning av en högpresterande processor anpassad för FPGAer

Improving and Extending a High Performance Processor Optimized for FPGAs

Författare
Author

Kristoffer Hultenius och Daniel Källming

Sammanfattning
Abstract

This thesis is about a number of improvements and additions done to a soft CPU optimized for field programmable gate arrays (FPGAs). The goal has been to implement the changes without substantially lowering the CPU's ability to operate at high clock frequencies. The result of the thesis is a number of high clock frequency modules, which when added completes the CPU hardware functionality in certain areas. The maximum frequency of the CPU is however somewhat lowered after the modules have been added.

Nyckelord
Keywords

FPGA, Soft CPU, Xi2, Embedded, Cache, Division, Interrupts

Abstract

This thesis is about a number of improvements and additions done to a soft CPU optimized for field programmable gate arrays (FPGAs). The goal has been to implement the changes without substantially lowering the CPU's ability to operate at high clock frequencies. The result of the thesis is a number of high clock frequency modules, which when added completes the CPU hardware functionality in certain areas. The maximum frequency of the CPU is however somewhat lowered after the modules have been added.

Sammanfattning

Detta examensarbete handlar om ett antal förbättringar och utökningar av en mjuk processor speciellt anpassad för fältprogrammerbara grindmatriser (FPGA). Målet har varit att göra förändringarna utan att göra större avkall på processorns förmåga att operera i höga klockfrekvenser. Resultatet av examensarbetet är ett antal moduler som klarar av höga frekvenser och kompletterar processorns hårdvarufunktioner. Dock reduceras maxfrekvensen på processorn något med modulerna tillagda.

Acknowledgments

We would like to thank our supervisor and examiner Andreas Ehliar for guidance during the thesis. We would also like to thank Johan Eilert for many interesting discussions and for the help with benchmark values. Finally we would like to thank our fellow thesis workers Karl Bengtsson and Olof Andersson for valuable discussions and suggestions.

Contents

Chapter 1

Introduction

This chapter will introduce the thesis and the report.

1.1 Background

Today, there exist several soft Central Processing Units (CPUs) targeted for Field Programmable Gate Arrays (FPGAs) on the market, ready to be used as alternatives or complements to hard counterparts. At the same time modern FPGAs are able to run at increasingly high frequencies, e.g. the FPGA targeted in the thesis has a maximum frequency of 500 MHz. However, none of the available soft CPUs seem to take advantage of this ability. A soft CPU that do take this advantage is xi2, but at the same time it lacks features compared to its competitors. A more thorough background is given in chapter 3.

1.2 Purpose

The purpose of the thesis has been to find differences in functionality in the soft CPU xi2 compared to similar devices, and implement extensions to xi2 containing functionality often found in the similar devices but not in xi2. The extensions were supposed to meet xi2's high maximum clock frequency (f_{max}), which at the start of the thesis was 345 MHz in a Virtex-4 of speedgrade 12.

1.3 Intended Readers

This document is intended for readers with basic knowledge in digital circuits, microprocessors and embedded systems. Necessary knowledge of concepts such as FPGAs and soft CPU cores will be explained in the initial chapters of the report.

1.4 Methods

Methods common to all implementation parts of the thesis are presented in chapter 2. Module specific methods are presented in corresponding chapters.

1.5 Limitations and Scope

A limitation which has had a large impact on the performed work is the maximum clock frequency, f_{max}, of the constructed modules. As one goal has been to not substantially lower f_{max} of the CPU itself, individual modules must handle *at least* as high frequencies. This limits the amount of logic complexity between registers. The purpose has been to extend the CPU, not make a new one, hence functionality and behavior that not absolutely had to be changed have been retained. Constructed modules have had to fit into the existing pipeline, or at least work together with it.

Available time has in the end turned up to be a major limitation. To construct modules that both work as expected and at acceptable frequencies is time consuming, more functionality could have been implemented if more time had been available.

Development has targeted a specific FPGA model, Virtex-4 speedgrade 12 (exact part number is xc4vlx80-12-ff1148), though there have been tests done where the CPU has been synthesized to other FPGAs in the the Virtex family. No synthesis to ASIC targets has been done.

1.6 Distribution of Work

The thesis work has been distributed on two persons. Some parts of the thesis are contributions by a single person, while other parts are joint efforts.

Daniel has contributed the divider, multiplier with following pipeline extension, and AGU.
Kristoffer has contributed caches, interrupts with following modifications of program counter and flag generation.

1.7 Thesis Outline

The report comprises twelve chapters.

Chapter 1 is this chapter. It contains an introduction to the report and thesis including limitations and scope.

Chapter 2 describes the concepts of FPGAs and optimization techniques for high speed FPGA constructions.

Chapter 3 describes the concept of soft circuits and compares it to other means of realizing circuits. Also presents xi2 and similar soft CPUs.

Chapter 4 describes advantages of memory caches and the implementation of caches in xi2.

Chapter 5 explains the work of an address generator and the implementation of a new one in xi2.

Chapter 6 introduces the concept of interrupts and describes the implementation of an interrupt controller in xi2.

Chapter 7 describes the multiplication situation in xi2, why a complementary short multiplier was needed and why the CPU pipeline in turn had to be extended. It also describes the implementation of those changes.

Chapter 8 explains the concept of integer division and some techniques to realize circuits performing such work as well as the implementation of a 32-bit integer serial divider in xi2.

Chapter 9 presents some smaller additions, enhancements and tests that have been done during the thesis.

Chapter 10 contains descriptions of and results from benchmarks run on the extended xi2 processor.

Chapter 11 presents and discusses the results obtained from the thesis as well as possible future work that can be done to xi2.

Chapter 12 presents the conclusions drawn from the thesis.

Appendix A describes how the instantiation of an adder can be done in a Virtex-4.

Appendix B contains resource usage numbers from synthesis reports of various versions of xi2.

Appendix C contains resource usage numbers from synthesis reports of Microblaze cores with/without hardware divider.

1.8 Abbreviations and Terminology

Explanations of terms and abbreviations used throughout the report.

AGU Address Generator Unit, generates addresses to data memories in possibly special ways.

Altera Large manufacturer of FPGAs.

ASIC Application Specific Integrated Circuit.

AU Arithmetic Unit.

CAD Computer-Aided Design.

CE Clock Enable.

CL unit Performs CLO and CLZ.

CLO Count Leading Ones.

CLZ Count Leading Zeros.

CM Constant Memory. Special memory in xi2 holding constants used in assembler programs.

Convolution Discrete convolution is defined as

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

CPU Central Processing Unit.

DCT Discrete Cosine Transform.

DM Data Memory.

DP Dual Port.

DSP Digital Signal Processing.

DSP48 Special multiplier blocks in Virtex-4, well suited for DSP operations such as MAC.

Flip-flop A clocked register, which delays data one clock cycle from input to output.

f_{max} In this report: the highest clock frequency at which a circuit still operates correctly according to static timing analysis with worst case parameters, which are 85 °C and 1.140 V for the targeted FPGA.

FPGA Field Programmable Gate Array. Programmable logic.

HDL Hardware Description Language.

kB kilo Byte.

LE Logic Element. A grouping in Altera FPGAs which consists of one LUT4 and one flip-flop.[4]

LSB Least Significant Bit.

LU Logic Unit.

LUT# LookUp Table with # inputs. Combinational block in an FPGA capable of realizing logic functions of its input signals.

MAC Multiply-and-ACcumulate. The act of multiplying a stream of input values and accumulating their resulting products. Essential DSP operation.

Microblaze Soft CPU made by Xilinx.

Modelsim Program for simulation of circuits in various forms.

MSB Most Significant Bit.

Nios II Soft CPU made by Altera.

NOP No OPeration. An "empty" instruction which in itself performs nothing.

NRE Non-recurring Engineering.

OpenCores An organization whose "main objective is to design and publish core designs under a license for hardware modeled on the Lesser General Public License (LGPL) for software."[29]

OR1200 OpenRISC1200. Open source soft CPU based on the OR1000 specification by OpenCores.

PC FSM Program Counter Finite State Machine.

PM Program Memory.

RAM Random Access Memory.

Slice Segment of Xilinx FPGAs consisting of LUTs and flip-flops with internal routing between the parts.

Speedgrade A number representing how fast a (Xilinx) FPGA can run. For Virtex-4, speedgrade 12 is the fastest at the time of writing.

SR Set/Reset. An input to a flip-flop forcing the output to 1/0.

Static timing analysis A method of computing the expected timing of a digital circuit which does not require rigorous simulations of the circuit.[26]

Stratix II High performance LUT4-based FPGA made by Altera.

Targeted FPGA Virtex-4 speedgrade 12, part number xc4vlx80-12-ff1148.

Virtex-II/4/5/6 High performance FPGA made by Xilinx. II and 4 are LUT4-based, 5 and 6 are LUT6-based.

Xilinx Large manufacturer of FPGAs.

Chapter 2

Field Programmable Gate Arrays

This chapter will describe the basic concepts of Field Programmable Gate Arrays (FPGAs), how they are structured and some optimization strategies for high-speed FPGA designs.

2.1 Overview

Field Programmable Gate Arrays (FPGAs) are integrated circuits that can be programmed/configured to carry out custom functionality, even after being manufactured. The configuration process is fast and can be done by relatively cheap tools, making it ideal for fast paced development and for projects with few resources. Thus, FPGAs are excellent for prototyping and low scale productions.

The FPGA market is dominated by two companies, Xilinx and Altera. Together they have a estimated market share of 87% as of 2009 [1]. Even though the companies develop their own FPGA architectures, the basic idea and structure are the same.

2.2 General Structure

FPGAs are Programmable Logic Devices (PLDs). They are built up of thousands of reconfigurable logic blocks connected through a large interconnect network. Different FPGA architectures have their own blocks but the most common ones usually share the same structure. The logic blocks can be divided into general logic blocks and specialized logic blocks. See figure 2.1 for an overview.

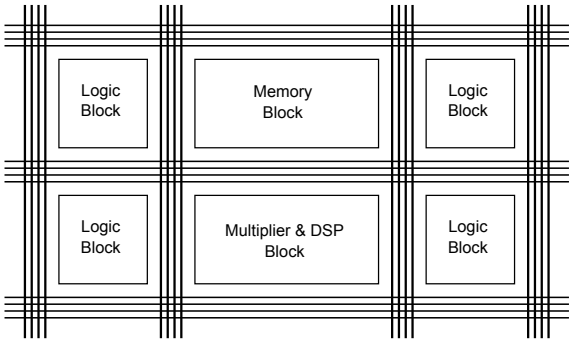


Figure 2.1: Typical FPGA architecture.

2.2.1 General Logic Blocks

The general logic blocks can be used to perform any kind of custom logic. These blocks normally include one or more of the following components.

- **Lookup Tables (LUTs)** LUTs are the most basic components of an FPGA. They are configurable lookup tables with typically 4 or 6 inputs and one output. Any boolean expressions (no matter the complexity) with up to 4/6 variable inputs can be realized by one LUT alone. Consequently, LUTs can be used as 16-bit/64-bit asynchronous ROM memories whereas the input bits are used as an address.

- **Flip-Flops** Flip-Flops are synchronous clocked registers that can hold one bit. Flip-Flops are often placed after LUTs. Flip-Flops can sometimes also be configured as synchronous ROM memories

- **Full Adders and Carry Chains** It is often both cumbersome and time consuming to make large additions/subtractions using only LUTs. Therefore special logic exists to ease these operations. In particular high-speed carry chains that removes the need for external routing between full adders.

- **Dedicated Multiplexers** Multiplexers are often used in digital designs. Dedicated multiplexers are both smaller and faster than LUTs and are therefore often present.

2.2.2 Specialized Blocks

General logic blocks are good enough for the most part, but there are some common tasks that can be very inefficient to realize using only general logic blocks. Many FPGAs therefore include specialized blocks which enables or vastly improves the specific functionality. Some of the most common ones are described below.

- **Memory Blocks** Memory blocks are densely packed SRAM-cells that can be used to synchronously store a lot of data. Memory blocks are by far more area and performance efficient than using flip-flops for the same task.

- **Dedicated Multipliers** Dedicated multipliers are used to speed up multiplications, which are complicated to implement using only LUTs. Some FPGAs also include an optional adder and an accumulator after the multiplier, making it suitable for DSP solutions.

2.2.3 Interconnects

All logic blocks and specialized blocks are connected to a large interconnect network. This network can be controlled by switch matrices which are also configurable.

2.3 Virtex-4

This thesis will mainly focus on Xilinx FPGA devices and in particular the Virtex-4 family. The Virtex-4 architecture will therefore be described a little bit deeper. The reader is referred to [34] for the complete specification.

2.3.1 Configurable Logic Blocks and Slices

The general logic blocks are called Configurable Logic Blocks (CLBs) in Xilinx architectures. These blocks consist of four Slices in Virtex-4. Each Slice has two 4-input LUTs, dedicated multiplexers and two flip-flops. The Slices are lined up two-by-two in order to support two carry chains. The carry chains are directly interconnected to the surrounding CLBs, allowing fast carry propagation by avoiding the external interconnecting network. See figure 2.2 for an overview of how CLBs and Slices are structured.

2.3.2 BlockRAM

The memory cells in Xilinx architectures are called BlockRAMs. In Virtex-4 these memory cells are 16 kbit (2 kB) in size. Each BlockRAM also has an additional space of 2 kbit, which can either be used freely or for parity bits. The BlockRAM width (bits per entry) and height (number of entries) can be configured in many ways (see table 2.1). BlockRAMs also have two “true” data ports which makes it possible to do two simultaneous read or write operations on different addresses at the same time.

One of the biggest drawback of using a BlockRAM is the long output delay. After the clock signal goes high it takes 1.65 ns until valid data can be seen at the output port. The same delay for a normal flip-flop is 0.27 ns.

Bits per entry	Entries
32 + 4	512
16 + 2	1024
8 + 1	2048
4	4096

Table 2.1: Possible BlockRAM configurations.

2.3.3 DSP Block

Virtex-4 is equipped with hardware DSP blocks which are capable of doing 18x18 signed multiplications in one clock cycle.

2.4 Designing for Virtex-4

2.4.1 Tools and Design Flow

The hardware design is described in a hardware description language (HDL), such as VHDL, Verilog or SystemVerilog. The design is then synthesized and mapped into hardware specific logic blocks. The last step is to place all logic blocks on actual hardware nodes and route them together through the interconnection network.

For this thesis Xilinx' own tools from the ISE Design Suite were used. The tools had been used for the previous work and project specific development resources were built around them. Other Virtex-4 compatible design suites/toolchains also exist, like Synplify from Synopsys.

2.4.2 Optimization

When optimizing for high performance designs, a lot of considerations have to be made. Not only has the high level HDL code be well thought-out, the tools used to translate HDL code to actual hardware components sometimes needs a helping hand. Experiences have shown that the used synthesis and routing tools are not always able to deliver acceptable solutions. The tools might also have other goals, like optimizing for area rather than speed even though speed has been specified as the primary goal by the developer.

For performance centric designs that approach the maximum Virtex-4 frequency of 500 MHz everything has to be near perfect. Xi2, which will be described in chapter 3, can be synthesized to run at a frequency around 340 MHz (cycle time of ~3 ns). To retain this high frequency a lot of optimization techniques have to be applied. The used techniques can be grouped into three categories.

HDL Optimizations

In order to achieve high performance the high level code needs to be built with the targeted hardware in mind and with a good feeling for what can be done in a given time period. If an operation cannot be performed in one clock period it has to be divided into two parts and be performed in two clock cycles instead. This method is called pipelining and is used substantially throughout the design. See listing 2.1 for an operation that could not be fitted into one clock cycle for a 400 MHz design and listing 2.2 for how it was solved using the pipelining technique.

Listing 2.1: No pipelining. Maximum frequency: 384 MHz

```
always @(posedge clk) begin
    if ((operandA[31:0] - operandB[31:0]) == 32'b0)
        z_flag <= 1'b1;
    else
        z_flag <= 1'b0;
end
```

Listing 2.2: Pipelining used. Maximum frequency: 431 MHz

```
always @(posedge clk) begin
    result <= operandA[31:0] - operandB[31:0];
    if (result == 32'b0)
        z_flag <= 1'b1;
    else
        z_flag <= 1'b0;
end
```

Manual Mapping

Instead of writing HDL code, logic can be directly mapped to specific hardware components. Manual mapping can be used to build highly optimized structures that the synthesis tools might not be able to construct. Manual mapping can also be used to access signals that otherwise would've been unconnected.

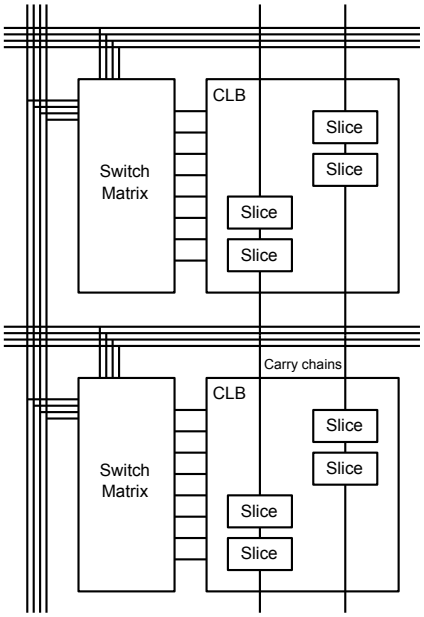
One manually mapped structure that has been used throughout this thesis is an optimized compare block. By utilizing the carry chain, normally used for additions, large comparisons (up to 64 bits) could be performed in a single clock cycle (340 MHz). Such high performance comparisons would not have been possible without manual mapping.

Floorplanning & Manual Routing

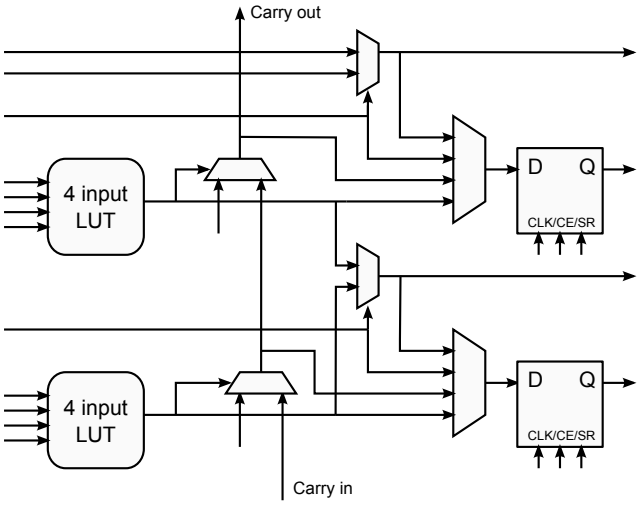
The most drastic way to optimize an FPGA design is to statically floorplan critical parts of the design and/or manually route the signals between them. Mapped hardware blocks are then placed either in absolute or relative positions on the

targeted FPGA device. Critical paths can be successfully eliminated by carefully moving involved hardware blocks closer to each other.

The downside of this method is that the floorplanned design might only fit one device family, or even worse, one specific device. There might also be problems when logic connected (directly or indirectly) to the floorplanned path changes. Floorplanning might in fact worsen the performance in the long run if not used carefully.



(a) Two Virtex-4 CLBs.



(b) Virtex-4 Slice.

Figure 2.2: A simplified view of Virtex-4 CLBs and Slices.

Chapter 3

Soft CPUs

This chapter will give a brief introduction to the concept of soft cores in embedded systems, differences to other types of cores and examples of available soft CPUs of different kinds. It will also present features common to soft CPUs which are missing in xi2.

3.1 Signal Processing in Embedded Systems

In an embedded system, there is bound to be one or several circuits doing some sort of signal processing. There are several ways to realize such a circuit.

3.1.1 General Purpose Microprocessor

If performance, power consumption and area are not major issues, a general off-the-shelf processor can be used. The main part of realizing the desired functionality is then done by writing software for the processor, utilizing its fixed instruction set. The advantage of this solution is the same as the disadvantage: the hardware functionality is fixed and done. No designing, testing, etc. of that area is needed, but at the same time the processor might lack processing power where needed while containing features never needed.[24]

3.1.2 Custom Chip Design

The opposite of using a general purpose circuit would be to create a fully custom Application Specific Integrated Circuit (ASIC). A very simplified description of traditional chip design is: after its desired functionality has been determined, a model of it is constructed, simulated and verified, CAD programs are used to lay out the actual chip, more simulations and verifications are done, before the design information is sent off to a specialized factory which produces the chip. A naked chip is seldom useful, why it must be placed in some kind of package which then interfaces with the outside world. Apart from the design time, there is a significant waiting time between sending off the design and receiving the packaged

chips. Then comes the tasks of testing and verifying. These things are all part of the Non-recurring Engineering, NRE. Finally, the chip might be put into real production.

Another variant of creating an ASIC is to take a soft core (see next section) and synthesize it to an ASIC target. Still there will be long waiting times, and the creation of expensive production masks is still required.[15]

When the design has been manufactured, nothing can be done about the functionality or layout, thus the term "hard" ware is very appropriate. If something has been neglected or badly laid out, it is possible that it isn't discovered until months after tape out. High costs of NRE and long waiting times can turn such a chip into a both time consuming and expensive mistake. Benefits of an ASIC are that it is fully tailored to the specific tasks it will carry out, thus there are great room for speed, power and area optimizations, and as production volume increases, NRE costs are spread thinner and the cost associated with a single chip is eventually reduced to levels which no other option can compete with.

3.1.3 Soft Cores and Programmable Logic

"A soft-core processor is a hardware description language (HDL) model of a specific processor (CPU) that can be customized for a given application and synthesized for an ASIC or FPGA target."[5] As seen in chapter 2, modern FPGAs have all the necessary building blocks for a general microprocessor. The same building blocks can be used to construct specialized modules for specific tasks. Simpler tasks, or tasks with low speed requirements can be run as software on a CPU core, while high throughput tasks are run on dedicated modules inside the same FPGA. The dedicated modules can also be used by the CPU i.e. by creating custom instructions, which is supported by most available soft CPU cores. Some cores are available as HDL open source, making even further customization possible. Soft cores used with an FPGA combines advantages of full custom ASIC and general purpose solutions along with reconfigurability of the hardware itself. For low volumes, programmable logic is also often considerably cheaper to use than an ASIC because of lower NRE costs, but this is not true for high volume products. FPGAs also fall short compared to full custom chips when it comes to optimal speed, area and power consumption. But, the reconfigurability and extensibility are properties of soft designs used with programmable logic solely, regardless of chip volumes.[14]

An FPGA-based design must not necessarily stay FPGA-based, it can later evolve to an ASIC or be used as an intermediate step in the design phase of an ASIC.

3.2 Xi2

The thesis report is based on work done to a soft CPU named xi2 and this section will give an overview and description of xi2's features and characteristics. For a more detailed description of the construction of it, see [15].

3.2.1 Origins

The xi2 CPU has its origin in another soft CPU named xi. While xi demonstrates that a soft CPU for FPGAs can be a good tool in certain applications, it has some undesirable characteristics. Mainly, the absence of result forwarding in the pipeline creates extra code scheduling complexity for the programmer. (More on forwarding can be found in section 3.2.4.) Main goals for xi2 were to make it *really* FPGA optimized (in order to run it at very high clock speeds) and add full forwarding. The development has targeted Virtex-4 speedgrade 12 in which xi2 reaches a clock frequency of 357 MHz with manual floorplanning.[15]

It should be noted that when the thesis work was started, it was impossible to synthesize xi2 to a higher clock frequency than 345 MHz. This will therefore be the frequency we use as reference throughout the report. Reasons for the lower f_{max} are probably updates and changes to (and combinations of) the tools used.

3.2.2 Pipeline Layout

Xi2 has a seven-stage pipeline comprising instruction fetch (IF), decode (DE), operand fetch (OF), forward (FW), execute one (EX 1), execute two (EX 2) and write back (WB) stages, which is illustrated with a block diagram in figure 3.1. In order to avoid large, slow muxes in to (as an example) the write back of the register file, non-active execution units output zeros. This way, all results from an execution stage can be input to an or-gate, where the single 32-bit output is the correct result from that stage. Muxes are still needed to select between stages in for instance the forwarder, but the amount is clearly reduced.[15] Pipeline parts of special interest are presented below.

3.2.3 Decoder Stage

Apart from the obvious decoding of instructions, the decoder stage does detection work for the forwarder. It looks for instructions in the execution stages writing to registers used as operand registers by instructions soon-to-be in the execution stages. The information is then checked by the forwarder in the next stage. The decoder also checks for false positives and sets the matching information to zero by using reset inputs on the flip-flops.

3.2.4 Forward Stage

When an execution unit is finished and has produced a result, one would probably expect the result to be available the next clock cycle. However, as updating of actual register values is not done until the result has flowed down to the writeback stage, this is not the case. Also, some operations take less clock cycles to execute than others, but without forwarding there are no benefits from this because the result still has to flow through the remaining execution stages. The forwarder logic solves these problems, though it should be noted that it is in no way magic; it can't make results that takes e.g. 2 clock cycles to produce appear any sooner.

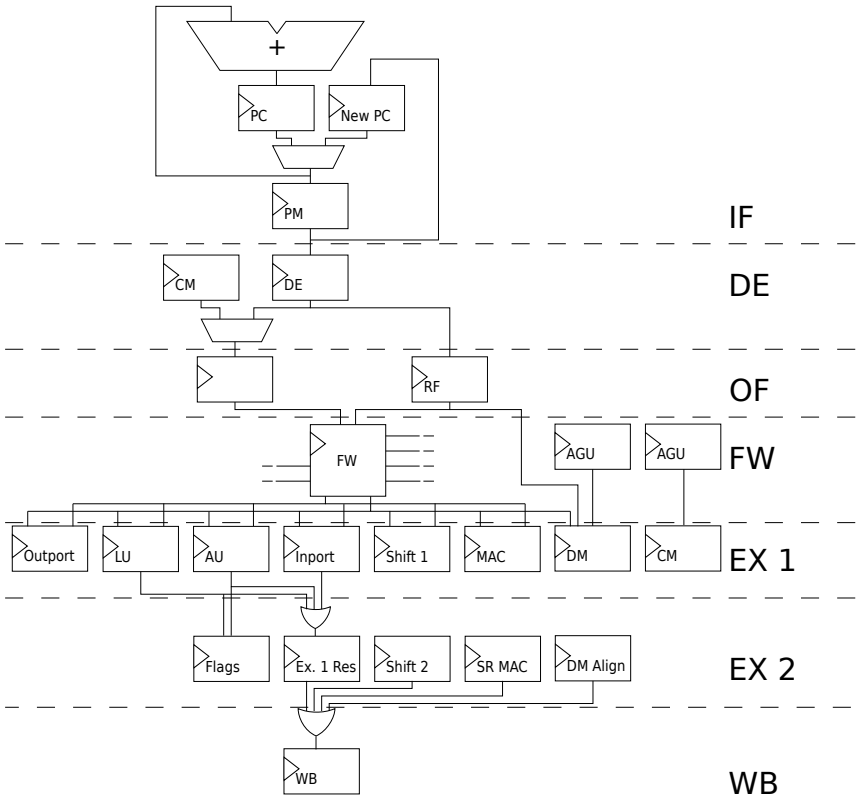


Figure 3.1: Xi2 pipeline.

Forwarding done by the forwarder also has a minimum waiting time of one clock cycle, but internal forwarding is present in some of the execution units which mitigates this issue. For instance, the arithmetic unit can use a result from the previous arithmetic operation directly in the next clock cycle, without any clock cycle waiting. This is possible because the result is fed back combinationally, avoiding an extra result register. There is however no such direct forwarding between different execution units.

Another solution to utilize faster execution units would be to allow write backs to the register file after each execution stage. Doing this unfortunately creates code scheduling hazards, as several stages possibly would want to write results back at the same time.

The forwarder also takes care of delivery of constant memory and data from data memories addressed with the Address Generator Unit (AGU) to the execution units and performs various pre-processing for the arithmetic unit.

3.2.5 Memories

Based on a kind of harvard memory architecture, xi2 has separate instruction and data memories, as well as a special constant memory. All constants used by assembler programs are extracted and placed in the constant memory at assembly time. It is strictly read-only at run time, and constants enter the data path via the forwarder muxes.

Address Generator

To handle DSP data in a reasonable way, a special address generator is used. Both data and constant memories can be addressed by the AGU, and it is placed in the same pipeline stage as the forwarder. Data originating from this addressing enters the data path via the forward muxes, via their own memory outports. More information about the address generator can be found in chapter 5.

3.3 Existing Soft CPU Designs

There exist several soft CPUs on the market, in various forms and under various forms of licenses. A selection can be found in table 3.1. Performance variants have been chosen if such exist. All, except xi2, have some variant of instruction and data caches. Other more peripheral units such as MMUs (Memory Management Units) are not included in the numbers. Notable are the models where the vendor has been happy to brag about the clock frequency in FPGAs similar to our target, which is the case for Cortex-M1 and Microblaze.[8, 15]

Cortex-M1 is an ARM-based soft core targeted for FPGA. A small FPGA maker named Actel is involved, but the core is free to synthesize to any FPGA, in contrast to Microblaze and Nios II.[8] A major advantage might be that it implements a relatively well known instruction set, Thumb-2.

Microblaze is Xilinx' own soft core. It comes in two pipeline versions, with the longest (and fastest) being presented here. Still, it appears to be optimized towards resource preservation rather than high f_{max}.

The other dominant FPGA manufacturer beside Xilinx, Altera, has a soft CPU named Nios II. The fastest variant, /f, is reported to run at 185 MHz. There is also a much smaller version, /e, which actually runs at a slightly higher frequency of 200 MHz, but on the other hand waits for each instruction to pass through the CPU before starting the next one. /e "executes at most one instruction per six clock cycles" and ends up considerably slower in total.[3] The reported resource numbers are measured in LEs, Logic Elements, which consist of one LUT4 and one flip-flop. LE measurements should in other words be somewhat comparable to LUT measurements.[4]

Open source wise there is OpenCores' OR1200. The size measurement is from a synthesis to Virtex-II, which also is LUT4-based. While it has a noticeably lower

CPU	Vendor	f_{max} (FPGA) (Speedgrade)	Size [#LUTs/#LEs]
Cortex-M1	Actel/ARM	150 (Virtex-4) (12)	2300
Microblaze	Xilinx	200 (Virtex-4) (12)	2181
Nios II	Altera	185 (Stratix II) (fastest)	1800
OR1200	OpenCores	94 (Virtex-4) (12) [16]	4888 (Virtex-II)[10]
xi2	-	345 (Virtex-4) (12)	1814

Table 3.1: Soft CPUs.

f_{max} than all other cores mentioned here, it is fully customizable due to its open source nature.

3.3.1 Features of Soft CPUs

Evidently, xi2's clock frequency stands up well against the competition, but what about other features? One of the first things done in the thesis was looking through the instruction sets of other soft CPUs and a few traditional hard CPUs popular in embedded applications. Comparing xi2 with other CPUs, xi2 lacks some common instructions[33, 23, 7, 3], most notably an integer divide instruction and a multiplication instruction that can be used just as any other instruction. (The current xi2 multiply instruction uses hardware placed outside the pipeline, takes several cycles to perform and the result must be fetched from a special register.) On the hardware side, missing features common in other CPUs are caches, interrupt and exception support and memory management unit. The missing instructions are also related to the hardware area, as there simply is no suitable hardware to realize their functionality. This thesis has focused on the instruction/hardware area.

On the tool-chain side, which has not been touched by this thesis, xi2 for example lacks compilers for any high level language and a software simulation model.

Chapter 4

Cache Memory

While xi2 was designed for maximum performance, it lacked one of the most important performance components, CPU cache memories. This chapter discusses different types of cache memory architectures and describes how cache memories were implemented in xi2.

4.1 Introduction

Cache (in computer science) is a local storage containing partial and temporary data copies of a backend storage. The cache storage is normally much faster than the backend storage, both in terms of latency and bandwidth per time unit, but at the price of less data capacity.

The main idea behind cache structures is to speed up program execution by storing frequently used data (including program instructions) in fast memories close to the CPU. There might also be other benefits aswell, like lower activity on shared data buses and the possibility of storing necessary dynamic branch prediction statistics together with the instructions.

Figure 4.1 shows the hierarchical storage structure of a typical computer system and some characteristic sizes and latencies for each level. Everything above the backend storage can in some way act as cache storages. However, only CPU cache memories (L1, L2, L3) are considered as true caches since they mirror the respective backend storage (the main memory) completely transparently.

This thesis will only deal with CPU cache (cache memories between the CPU and the main memory). Typical architectures and solutions for other kinds of cache storages are therefore not considered.

4.1.1 Execution Locality

As seen in figure 4.1, the CPU cache memory in a typical computer system can have a capacity lower than 1/1000 of the main memory and still perform well.

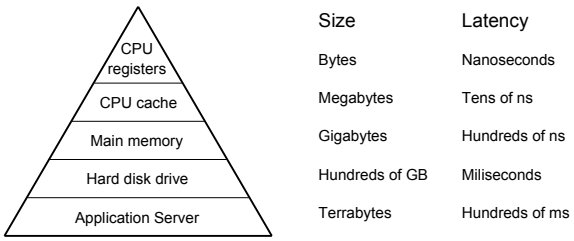


Figure 4.1: Storage hierarchy for a typical computer system. The values are compiled from various sources and should not be interpreted as fixed values.

This is due to the locality property of program execution. The locality property can be divided into three specific behaviors [17].

- **Spatial Locality** - If one data location is referenced, it is very likely that the same or any nearby locations will be referenced in the near future.
- **Temporal Locality** - If one data location is referenced, it is very likely that the location will be referenced again in the near future (special case of spatial locality).
- **Sequential Locality** - If one data location is referenced, it is very likely that the next location (location + 1) will be referenced soon.

These important patterns explain why the idea behind CPU cache structures works, despite being so small in capacity. If a cache structure can be built to accommodate these patterns, a lot of performance can be achieved with few resources. There is even a famous "thumb of rule" saying that roughly 10% of the code is executed 90% of the time.

To understand the importance of CPU caches and the locality patterns, consider example 4.1. Using a cache in this example would speed up the execution by a factor of 7.87 and at the same time lower the CPU <-> Memory bus activity to 10%.

Example 4.1

A simple program consists of 1000 instructions. Let's assume 10% of these are running 91% of time. If 100 of the instructions are being executed 91 times each (9100 executions), the other 900 would only be executed once (900 executions).

Time penalty for cache matches: $t_{match} = 3ns$
Time penalty for cache misses: $t_{miss} = 100ns$

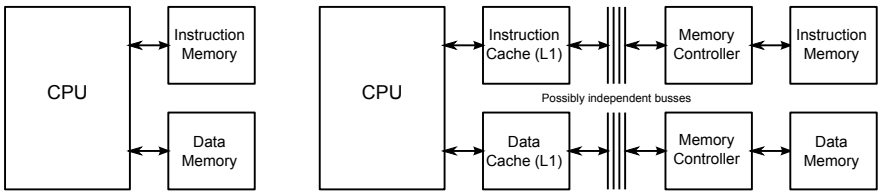
No cache: $T = t_{miss} * 10000 = 1000000ns = 1000\mu s$
With cache: $T = t_{miss} * 1000 + t_{match} * 9000 = 100000ns + 27000ns = 127\mu s$

4.1.2 CPU Cache Levels

CPU caches can be divided into several distinct cache levels (L1, L2, L3) with different memory sizes and speeds. This approach has been shown to perform better than having just one big cache memory [30].

When the CPU wants to access a memory location, a request is sent to the closest cache memory (L1, smallest and fastest). If the requested memory location can't be found there, the request will be forwarded to the next cache level (L2) and so forth until the memory data is found. When the data is found it will be passed down to the CPU and will also be stored in all cache memories on the way.

Since xi2 was built around the Harvard architecture with separated instruction and data memories it made sense to keep the memory structure separated. Therefore two different L1 caches were needed, one for data and one for instructions. The goal was simply to switch out the instruction and data memories with corresponding cache memories. See figure 4.2.



(a) Before. (b) After.

Figure 4.2: Implementation goal for the CPU caches.

Having another level of cache (L2) was not considered at this time but might be needed in the future.

4.2 Cache Architectures

Cache memories can be built in a number of ways. This section will outline the most common types of cache architectures and describe their advantages and disadvantages.

4.2.1 Cache Organization

Cache memories can be organized in different ways. There exist three basic organizations, direct mapped cache, fully associative mapped cache and N-way set associative mapped cache.

All of these cache organizations share the same basic idea. The memory address is partitioned into parts with different meanings (see table 4.1). The tag and index parts are then used to look up cached data, either directly or indirectly through

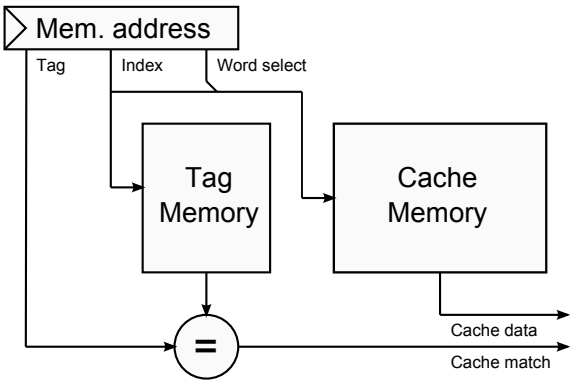


Figure 4.3: Direct mapped cache.

a directory memory. The word select part then points out which word should be returned if the cache line contains multiple words. The byte select part of the address is normally not used by the cache architecture.

Name	Bits	Description
Tag	18	Block index for the main memory
Index	10	Cache line index for the cache memory
Word Select	2	Word index for the cache line
Byte Select	2	Byte index for the word

Table 4.1: Typical partition of a 32-bit memory address.

Direct Mapped Cache

Direct mapped cache memory is the simplest cache organization. All memory addresses are statically mapped to a specific cache memory slot. Which slot depends on the index part of the address. The memory slot contains both memory data and the tag part of the memory address which the data belongs to.

When an address is referenced, the index part of the address is used to point out a cache memory slot. The tag in the memory slot is then compared with the tag from the referenced address. If both tags match there is a cache match and the data will be sent to the CPU. If they are different there is a cache miss and the request will be forwarded to the next cache level (or main memory if no more cache levels exist). See figure 4.3 for a block diagram.

Advantages: Easy to implement, fast, and low cost in terms of hardware area.

Disadvantages: Not good for programs that jump between memory blocks.

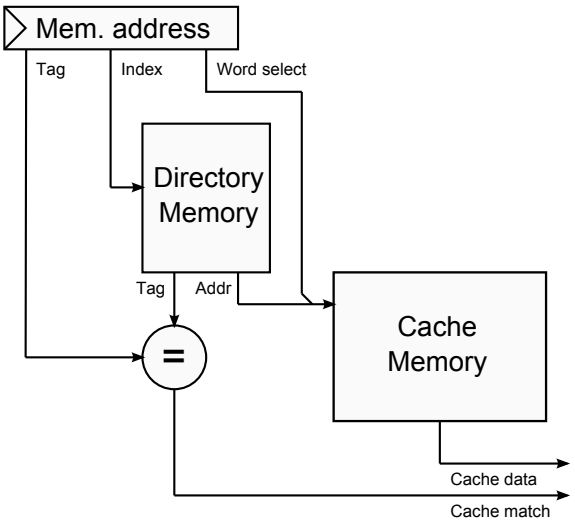


Figure 4.4: Fully associative mapped cache.

Fully Associative Mapped Cache

Fully associative mapped cache is a cache organization that dynamically maps addresses to the cache memory. This is done by having two separated memories, one for the cache itself and one for a directory. The directory holds information on all memory locations stored in the cache and their respective addresses in the cache memory.

When a memory request is made all entries in the directory are checked against the upper part of the memory address (tag + index). If a match is found the entry cache line address, along with the word select part of the memory address, is used to reference the cache memory containing the actual cached word. See figure 4.4.

Advantages: Dynamic mapping/full associativity.

Disadvantages: Slow, needs two consecutive memory lookups. Complicated comparisons if the directory is large.

N-way Set Associative Mapped Cache

The third cache organization is called N-way set associative mapped cache, where $N >= 1$ (typically $N = 2^M, M \in \mathbb{N}$). The idea is similar to direct mapped cache but instead of having just one data cache memory there are now N numbers of memories (called "ways"). Direct mapped cache is in fact a special case of N-way set associative mapped cache when $N = 1$.

When a memory request is made all ways are checked simultaneously and if there

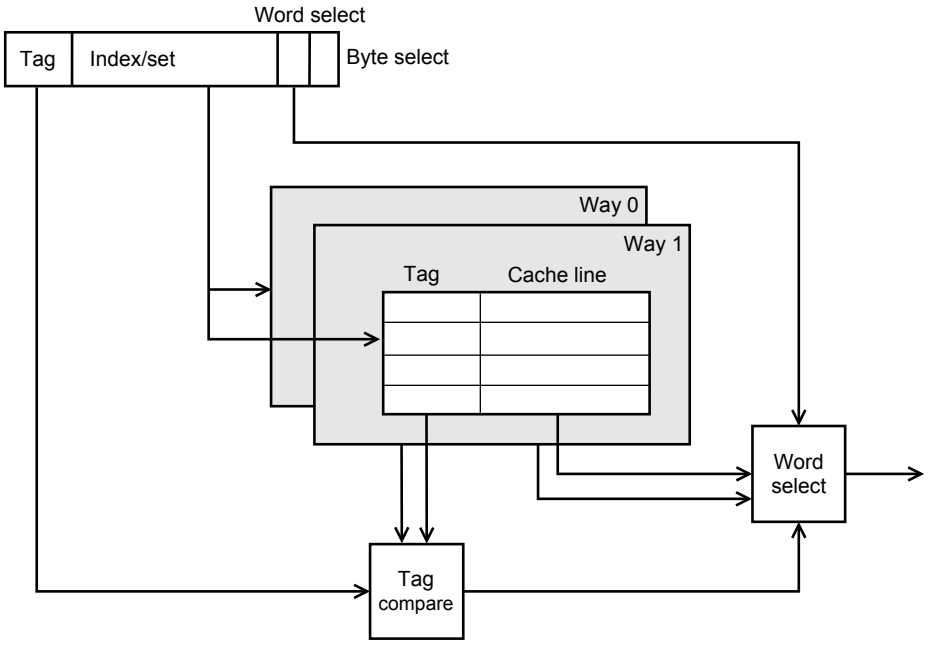


Figure 4.5: 2-way set associative mapped cache.

is a match in any of these the respective cache data is multiplexed out to the CPU. See figure 4.5 for an example where $N = 2$.

Advantages: High associativity if N is high.

Disadvantages: Slow (large compares and muxes) if N is too large.

Miss Rate Comparison

Since all three types of organizations have their own advantages and disadvantages a deeper comparison was needed to understand how they performed in real applications. Direct mapped, 2-way set associative, 4-way set associative and fully associative organizations were considered to be worth an evaluation.

A good cache organization evaluation was done by Mark D. Hill and Jason F. Cantin [20][9]. They carried out the SPEC CPU2000 benchmark on a SimpleScalar simulator, emulating an Alpha processor. The authors of the evaluation explain their choice of benchmark program as following: *"The SPEC CPU2000 benchmark suite (http://www.spec.org/osg/cpu2000) is a collection of 26 compute-intensive, non-trivial programs used to evaluate the performance of a computer's CPU, memory system, and compilers. The benchmarks in this suite were chosen to represent real-world applications, and thus exhibit a wide range of runtime behaviors."* [9]

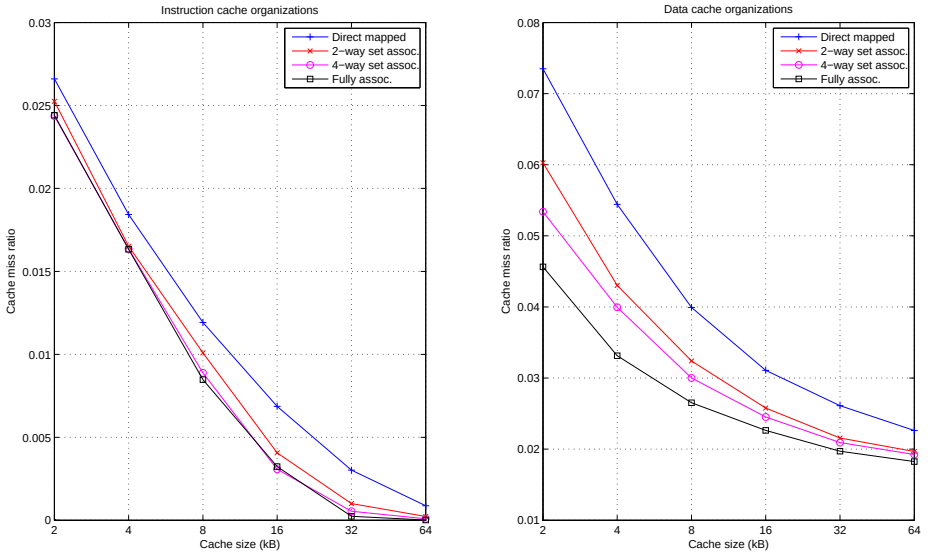


Figure 4.6: Cache miss rates for different cache organizations. Values are taken from [9].

Results for different types of cache organizations and sizes were collected and saved in result tables. Interesting result values for this thesis have been plotted in figure 4.6. Direct mapped cache shows a very high cache miss rate compared to the other organizations. This is not surprising since it also is the simplest organization. 2-way set associative decreases the miss rate quite a bit despite being just a little different from direct mapped cache. Fully associative has the lowest miss rate in this test but because of the high logic complexity even for smaller cache sizes it is not a suitable choice for xi2. It should also be noted that a good organization for the data cache is considerably more important than for the instruction cache, at least in this evaluation.

4.2.2 Cache Size

Another parameter which greatly affects the performance of a cache memory is the size. If the cache size is increased more data can be cached and there is a higher probability of cache matches (as clearly seen in figure 4.6). While higher cache sizes are to prefer, there is always a roof.

There are however other aspects to consider. One example is the balance between the cache line size and tag size. A large cache line size results in lower tag/directory sizes (tag size = cache size/words per cache line) and possibly fewer cache misses

for programs with high sequential locality. Consequently, programs with a lot of random accesses might suffer from large cache line sizes.

4.2.3 Replacement Policy

When a cache miss occurs, one cache line has to be thrown out in order to make room for the new one. This is not a problem if an empty cache line exists or when a direct mapped cache organization is used (only one choice). However, for the other cache organizations there are multiple cache lines to choose from. Therefore a replacement policy is needed. Many replacement algorithms exist. Three of the most common ones are listed below.

- **Least Recently Used (LRU)** The least recently used cache line will be replaced. This requires some kind of "age bits" for each cache line. Every time a cache line is requested the corresponding age bits has to be updated.
- **Most Recently Used (MRU)** Contrary to LRU, the most recently used cache line will be replaced. This policy has been proven good for program with large data sets and many random accesses [13].
- **Pseudo-Random** The cache line to replace is picked randomly. This solution is the easiest to implement.

4.2.4 Write Policy

Accessing the main memory is normally a time consuming task because of high latency. When data needs to be written to the memory there has to be an efficient write strategy that doesn't affect the instruction execution rate too much.

One solution is to use the **Write-through** policy. When this policy is used all writes are queued into a write FIFO buffer. This buffer is then written into the memory concurrently while the CPU executes new instructions. One problem with this approach is to determine a large enough buffer size. Memory read operations also needs to check the write buffer in addition to the cache memory.

Another solution is to use the **Write-back** policy. This policy only updates the local cache line and sets a "dirty bit" high. If the cache line has to be replaced at a later stage, it will first be written to the memory since the dirty bit is high. The CPU will be stalled while the write is performed.

The write-back policy might result in fewer memory writes and less bus usage, but when the write-back occurs the whole CPU has to be stalled. A combination of write-through and write-back would be desirable.

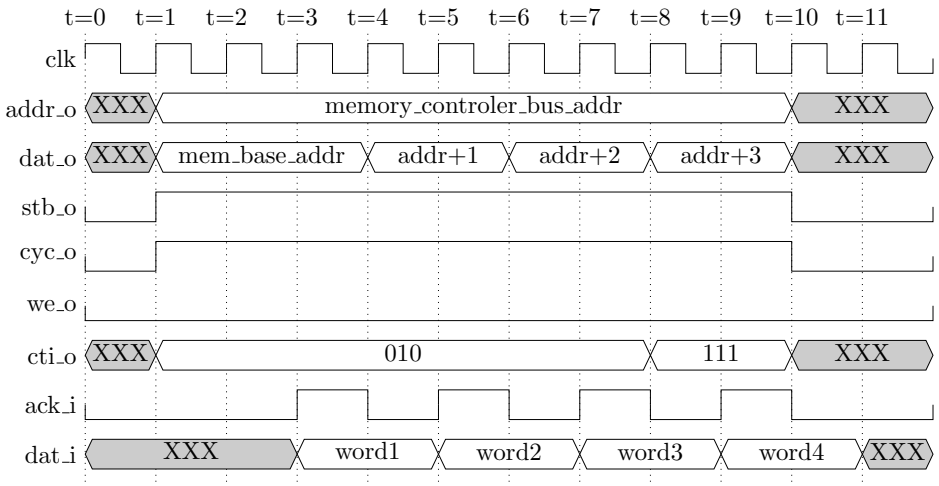


Figure 4.7: Cache fetch transaction on the Wishbone bus.

4.3 Memory Interface

In order to communicate with the main memory controller a memory bus was needed. There was no specific requirement more than it had to be fast enough to work at the maximum CPU speed. Wishbone was immediately chosen as a good bus for xi2. Not only is it well documented, fast enough, easy to use and royalty free, it is also well used in many OpenCores projects. [29]

There are two types of interfaces to the bus, master and slave. A master interface can do requests (with or without data) to any slave interface and the requested slave then responds (with or without data). Every member of the bus has a unique address or address range. It is up to the bus members themselves to listen to all broadcasted requests and act thereafter. See the specification [28] for more information about the Wishbone bus.

Figure 4.7 shows a timing diagram of a cache memory fetch transaction on the wishbone bus. The cache memory requests four words by utilizing the burst feature in the Wishbone specification.

4.4 Implementation of Instruction Cache

4.4.1 Overview

As described in section 3.2.2, xi2 uses a seven stage pipeline. The first pipeline stage is used to fetch an instruction from the instruction memory. If the instruction

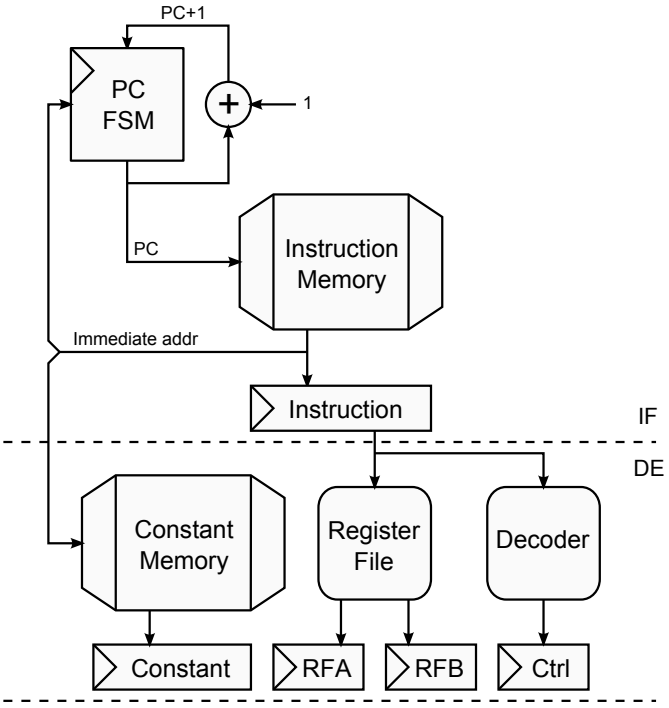


Figure 4.8: Instruction fetch before implementing cache.

is a direct jump (JAL, BRA) the immediate data part of the instruction word will be forwarded directly to the Program Counter Finite State Machine (PC FSM). This has to be done in order for the PC FSM to react on jumps quickly and thus avoid having more than one delay slot.

Based on the instruction type, any required data operands are then fetched from the register file and/or constant memory at the second pipeline stage. The instruction specific control/mux signals in the decoder are also set in this stage. Figure 4.8 gives a simplified view of the described pipeline stages. Signals and functions that are not of importance have not been drawn.

The instruction memory is implemented using a single BlockRAM. As described in section 2.3.2, BlockRAMs can be used to store large amounts of data (2 kB per BlockRAM) but at the cost of high output latency. The instruction memory output should therefore be used carefully to avoid introducing a new critical path.

The design goal was to transparently switch out the instruction memory with a fast and efficient instruction cache memory without affecting the maximum CPU speed in a negative way.

4.4.2 Configuration

Before the implementation could begin a cache memory configuration had to be decided. Many structural parameters existed, as discussed in section 4.2. By using this knowledge, the current implementation of the instruction memory and the competition (table 4.2) the following list of decisions were compiled.

CPU	Organization	Size	Line Size
Microblaze [33]	Direct mapped	64 B - 64 kB	4/8 words
Nios II [2]	Direct mapped	512 B - 64 kB	8 words
OpenRISC [21]	Direct mapped	1-8 kB	4 words

Table 4.2: Instruction cache configurations for other soft CPUs.

- **Cache Organization** Direct mapped cache was chosen as the primary cache organization simply because of the high speed requirements. The 2-way set associative cache was also considered as a viable organization since it was similar to direct mapped cache but delivered substantially lower miss rates at the cost of slightly more logic. Because of the similarities between the organizations both where decided to be built (with an configuration option to chose organization).
- **Cache Line Size** The cache line size was statically set to be 4 words (16 bytes). That size was assumed to be a good enough size. A lower value would increase the tag memory and therefore lower the total cache size. A larger value would allow for larger cache memories but would suffer if smaller caches were to be used (too few cache lines).
- **Cache Size** Given the speed requirements and the slow BlockRAMs, a too large memory size (many BlockRAMs) would have reduced the CPU speed because of long routing delays between the memories. Therefore, a maximum of 8 kB (four parallel BlockRAMs) was set as the upper limit. That way the tag memory could be fitted into a single BlockRAM. The lower limit was set to 2 kB (one BlockRAM).
- **Replacement Policy** LRU seemed to be the best choice but would've required a lot of space to save all "age bits" and comprehensive logic to update these bits on every access. Thus, the simple Random policy was decided to be good enough.

To summarize, a cache memory with the possibility to use either a direct mapped or a 2-way set associative organization was going to be built. The memory size should be configurable to 2 kB, 4 kB or 8 kB with 4 word cache lines and the random replacement policy.

4.4.3 Implementation

This section describes some implementation details for the instruction cache. See figure 4.9 for an overview of the implementation.

Normal Operation

When the instruction cache receives an instruction address (program counter value), the index part of the address will be given as an input address to a tag BlockRAM and one, two or four cache BlockRAMs (depending on configuration).

If more than one BlockRAM is used to store cached instructions the combined output values from all BlockRAMs will form the cached instruction. So, if an 8 kB cache is used the total memory is spread out among four BlockRAMs and 8 bits from each BlockRAM forms each instruction. Doing like this avoids the need for a multiplexer prior to storing the instruction into flip-flops.

In the same clock cycle as the instruction is fetched from the cache memory, the tag from the tag memory is also fetched and checked against the tag part of the instruction address. If they match everything will continue as normal and the cached instruction will be processed in the decoder at the next clock cycle.

Handling of Cache Misses

When a cache miss occurs the first thing to do is to invalidate the false instruction, which at this time is processed in the decoder. The instruction will be forced to a NOP instruction, meaning it will do nothing in the execution stages. As long as the cache fetch proceeds, all instruction will be forced to NOPs.

The PC FSM is at the same time notified via the global icache_miss signal. The previous program counter, the one that triggered the cache miss, will then be set as the new PC and the value will be frozen until the icache_miss signal goes low again. Since the rest of the pipeline (forward stage, execute stages, write back stage) is still executing, any instruction that can affect the execution flow has to be taken care of. For instance, conditional jump requests to the PC FSM needs to be queued and then processed as soon as the icache_miss signal goes low again.

For the cache itself a main memory fetch phase is entered. The requested instruction address is sent to the memory controller through the Wishbone interface. The burst feature of the Wishbone bus is used to get four words (one cache line) as soon as possible from the memory controller. The instructions are then inserted into the cache memory as they come and when all instructions are fetched the icache_miss signal will be set low again, resuming the program execution.

2-Way Set Associative Cache

Implementing a 2-way set associative instruction cache proved to be a tough assignment. Since the correct instruction cannot be decided at the instruction fetch (IF) pipeline stage due to timing constraints, it has to be done in the decode stage instead. This is a big problem. Not only must logic be added to an already time critical part of the processor, some resources have to be duplicated in order to handle both instructions.

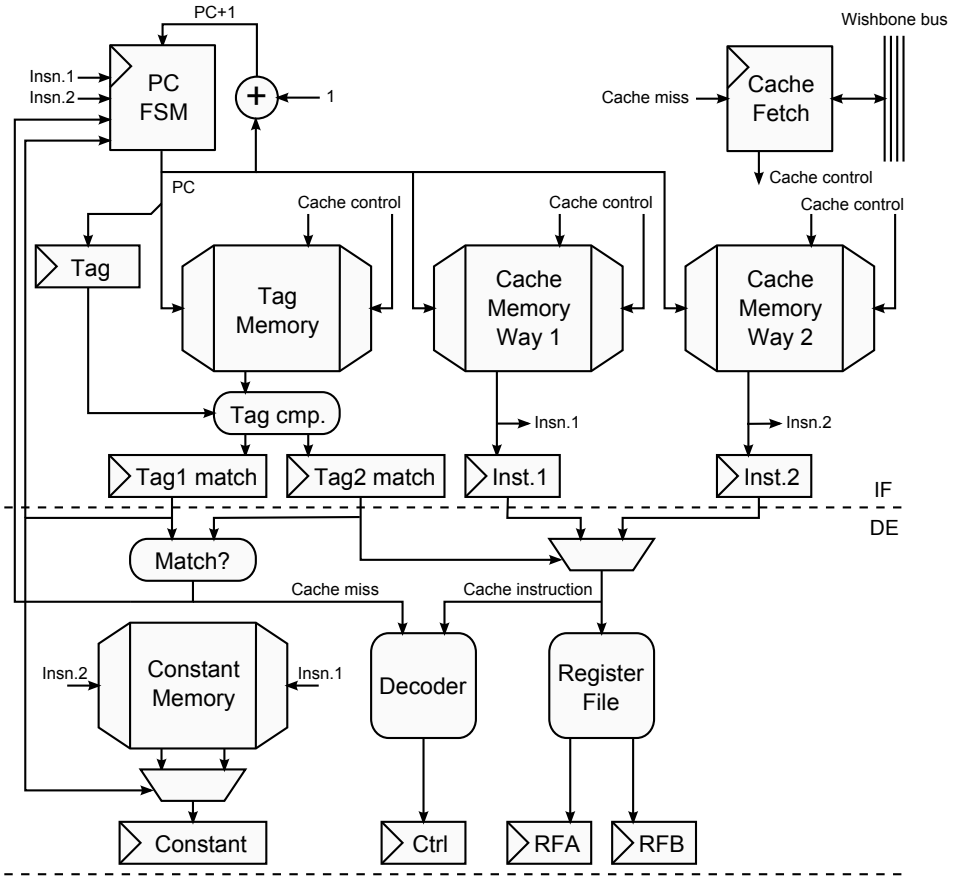


Figure 4.9: Implementation of 2-way set associative instruction cache.

For instance, the constant memory only has one port ready to fetch operands. The other one is occupied by the AGU, which is further described in chapter 5. When direct mapped cache (or no cache) is used, the output from the cache memory can be directly connected to the constant memory address input already in the instruction fetch stage. If the instruction is false and has to be invalidated, the constant operand will simply be ignored. When a 2-way set associative cache is used two ports are needed and the whole constant memory thereby has to be duplicated.

Another thing that has to be duplicated is some logic in the PC FSM. Since direct jumps include addresses as a part of the instruction word and need fast processing to avoid delay slots, both instructions are processed concurrently and the correct one is chosen at a later stage.

4.5 Implementation of Data Cache

4.5.1 Overview

The data memory has a few differences from the instruction memory. The biggest difference is the location of the memory. It is placed in the execution pipeline stages (EX1 and EX2) and is therefore, for the most part, independent from the rest of the CPU. The data memory also has two cycles to perform a memory operation, in contrast to only one for the instruction memory. Other differences are the extra logic needed to support write operations, byte alignment and shifting.

4.5.2 Configuration

Many of the conclusions for the instruction cache configuration (section 4.4.2) also holds for the data cache. Cache size, cache line size and replacement policy were decided in the same way. The 2-way set associative organization was chosen as the preliminary organization as the timing situation is more forgiving in the execution part of the CPU. As seen in figure 4.6, 2-way set associative data cache practically compares to a direct mapped data cache with twice the size. Direct mapped cache was also set to be implemented because of the similarity between the two organizations.

A write policy had to be decided in order to handle data writes. Write-back was chosen as the best option based on low area utilization and low performance impact. Write-through in contrast requires a ”long enough“ write queue and logic to check the queue in conjunction with the data cache, as valid data could reside in the queue. Write-back also substantially reduces memory bus activity compared to Write-through.

Table 4.3 list data cache configurations for some other soft CPUs.

CPU	Org.	Size	Line Size	Write Policy
Microblaze [33]	DM	64 B - 64 kB	4/8 words	—
Nios II [2]	DM	512 B - 64 kB	1-8 words	Write-back
OpenRISC [21]	DM	1-8 kB	1-4 words	Write-through

Table 4.3: Data cache configurations for other soft CPUs.

4.5.3 Implementation

This section describes some implementation details for the data cache. See figure 4.10 for an overview of the implementation.

Normal Operation

The basic idea is the same as for the instruction cache. The tag memory is moved up one pipeline stage to detect cache misses as soon as possible. When a write operation has to be performed the destination cache way is still unknown.

This is solved by using the BlockRAM READ_FIRST property. Both cache ways are then read and written at the same time. If a cache way is overwritten incorrectly the old data is rewritten at the next clock cycle.

Cache Fetch

The data cache fetch works in the same way as the instruction cache fetch, except for the Write-back policy. If the cache line which is going to be thrown out is marked as dirty, it needs to be written to the main memory first.

Stalling the CPU

The most difficult task was to integrate the data cache with the rest of the CPU. When a data cache miss occurs the whole CPU has to be stalled while the cache is requesting data from the main memory. This would not be a problem if a stall infrastructure existed, which unfortunately is not the case for xi2. Building a stall infrastructure would require a stall signal to the CE input of almost all flip-flops in the design. This would have reduced the maximum frequency a lot and introduced new kinds of problems. Another solution had to be found.

The solution in this case was to constantly flush and restart the upper part of the pipeline (EX1 and "above“) until the requested data was delivered to the data cache. That way all instructions positioned before the memory operation could finish their execution while instructions after the memory operation were practically stalled. The already existing infrastructure for invalidating instructions was used to flush the pipeline and the PC FSM was rebuilt to handle restarts.

Since cache misses are detected in the EX2 stage one of the biggest problems with this solution was to stop the instruction directly after the memory operation (in EX1) from being partially executed. For instance, an OUT instruction would have written two messages to the output port and a MAC instruction would have started the MAC twice. A good deal of logic had to be rewritten to handle this situation and the outport module had to be delayed to EX2.

Another big problem was to handle other events that had to change the PC at the time of a cache miss. This is described more in section 4.7.

4.6 Tests

At the start of the implementation a test bench was built to automatically test and verify the operation of the cache memory. As the implementation grew larger,

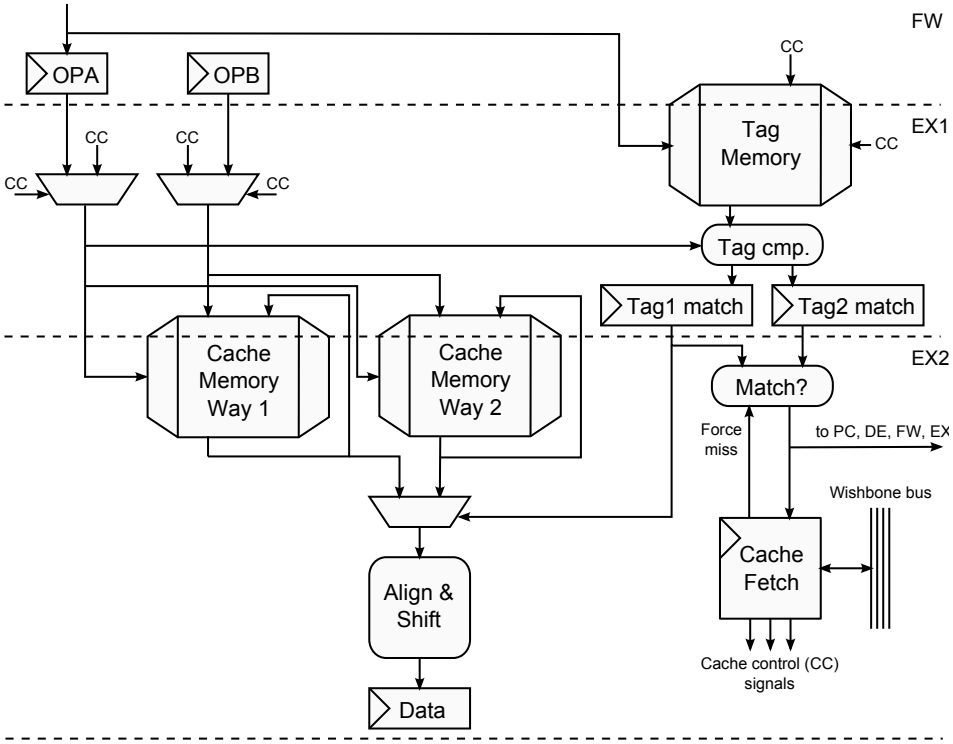


Figure 4.10: Implementation of 2-way set associative data cache.

was fragmented into instruction and data caches and had to support additional cache configurations the test bench was not updated to reflect the changes. A basic test bench with manual verification was used instead. It was hard to build a complete test bench since the most complicated part of the implementation was to integrate the cache memories with the rest of the system.

Another verification method was to write test programs with corner cases. Such a program was written for the data cache. The instruction cache was tested by running a regression test with many test programs.

4.7 Results

Both instruction and data caches have been successfully integrated into the CPU without lowering the maximum frequency or with the need of a system wide stall signal. The caches can be configured in a number of ways, both in size and organization.

Due to tight timing requirements in the first pipeline stages, a 2-way set associative instruction cache was hard to implement. Such architecture required a lot of hardware duplication to work and still missed the performance goal. Using a 2-way set associative organization would lower the maximum frequency of the final design to around 287 MHz. Direct mapped cache on the other hand meets the requirements, at the cost of a slightly higher cache miss rate.

The data cache had room for more complicated architectures and thus had no problem to fit a 4 kB 2-way set associative organization, which is now set as the default organization in the configuration file. This configuration compares to 8 kB direct mapped data cache.

One big consequence of the cache implementation was the impact on the PC FSM. From being quite simple this part of the CPU is now extremely complicated. Many things can happen at the very same time (instruction cache miss, data cache miss, conditional jump, unconditional jump, interrupts). Since these events can neither be executed concurrently nor be ignored, a small priority queue is holding them apart. This solution seems to work for the most part but needs further verification.

Chapter 5

Address Generator Unit

Used for the special addressing modes needed in signal processing, the Address Generator Unit (AGU) is definitely one of xi2's DSP distinctions. Due to cache implementation, the AGU had to get more integrated with the pipeline and, as it turned out, also heavily modified.

5.1 Old AGU

The old AGU was placed in the forwarding stage in the pipeline and had its own memory port on the constant and data memories. This meant a rather trouble-free environment and the dependency on what was going on the rest of the CPU was low. Theoretically, any instruction could use it, and data was fed as operand A and B via the forwarder. It featured circular addressing with configurable step size, and had its own two bits in the instruction word, telling it to step up the addresses for respective memory.

5.2 The Multiply-and-Accumulate Unit

In xi2, the Multiply-and-ACcumulate (MAC) unit takes care of multiply-and-accumulate and pure multiply instructions. The multiplication part can be signed*signed or unsigned*unsigned. Operation is as follows: operands are fed from the forwarder (as for all standard instructions), multiplied and added to/stored in a 64-bit internal accumulation register. In order to use a result, the high and low part of it must be read in to two normal registers. When doing accumulation, a final finishing instruction must be issued, because of how the MAC unit is constructed. Also differentiating it from other execution units is the five clock cycles latency. The special addressing modes provided by the AGU is related to DSP, and so are operations using MAC, for instance convolution.

5.3 Ideas for a New AGU

The following ideas and properties for a new AGU were formulated:

- The cache is using both ports on the data memory, but because the complex situation of double cache misses (one on each port) is not handled, only one can be used reliably. Thus, an AGU would have to access memory the same way as ordinary memory operations, and be placed in the forwarding stage. Constant memory has remained the same, why access to it still is rather uncomplicated.
- Rather than to let all instructions use special addressing modes, only operations using the MAC unit should be allowed to, to avoid further complications in for instance the forwarder. The MAC unit should be connected directly to the data cache and constant memory outputs.
- Because data from the cache machinery is ready in third execution stage, the MAC unit would be moved there.
- The AGU should still support circular addressing and with configurable step size.
- Because of the sole memory port, the AGU would have to let ordinary memory accesses through seamlessly.
- The event of a cache miss must be handled. As it is not known if a certain address causes a miss until a few clock cycles after it is given to the cache, address registers must be possible to rewind in some way. During cache-miss recovery, instructions in the forwarding stage are restarted several times. As this is where the would AGU act, it would have to look out for these "false restarts". The AGU should also handle miss-predicted jumps.

New Instructions

In order to use and manage the new AGU, the instructions in table 5.1 were created. MAC and MUL with signed/unsigned variants came for free by using the same bits in the instruction word as the existing MAC unit instructions, thus that functionality didn't need any new decoding. Two additional bits in the instruction word are used to mark the new instructions as AGU specific. ACC is the accumulation register inside the MAC unit, ACC_int is used to symbolize that the finalizing instruction must be issued in order to get the final result in ACC.

5.4 Implementation

After some testing, a structure according to the block diagram seen in figure 5.1 was constructed. Ordinary memory addresses are, just like before, computed as RF + IMM. IMM is the immediate offset field in the instruction word.

Mnemonic	Description
conv.mul[s]	ACC = cm(AR0)*dm(AR1), AR0++, AR1++
conv.mac[s]	ACC_int += cm(AR0)*dm(AR1), AR0++, AR1++
convsetup	STEP = OpA, AR = OpB
modsetup	TOP = OpA

Table 5.1: AGU instructions.

In the generator part the main address register is called AR. AR_1 and AR_2 are delayed versions of AR, especially needed when a cache miss has occurred and AR has to be rewound. Setup instructions are used to load values from the register file into the STEP, AR and optionally TOP register. The bottom address (or base address) is taken directly from the register file when the generator is engaged. AR is set to the same value as STEP at setup and accumulates one more STEP value each time a convolution instruction is executed. The first AGU mode address after setup is however merely the base address, realized as RF + IMM (IMM set to zero). Following addresses are then computed as RF + AR. Clock enable inputs are used on most registers in order for them to keep their values during non-AGU operation.

If the TOP register is written to, circular addressing mode is activated. The circuit in the lower right part of figure 5.1 resets AR when the time has come to wrap around. Circular addressing is turned off if AR is rewritten. The reset condition is at the time of writing computed as AR_1 == DIFF and fed combinationally to the reset input. Because the reset input of the flip-flop in (at least) the targeted FPGA input is slow, it should preferably be connected directly to a register, as shown in figure 5.1. It would seem trivial to do the computation with AR instead of AR_1 and store the result in a register, the problem lies in that AR sometimes holds something else than the future value of AR_1 and thus it is not trivial to do the calculation one clock cycle "earlier". The discrepancy in the relation between AR and AR_1 is due to the rewind procedure used when for instance cache misses occur.

After some initial testing, it was found that AR_1, not AR, actually is the register which has the timing with RF wished for when doing the final address addition.

As one would probably want different behavior of the constant memory AGU and data memory AGU, they should be separate and be possible to configure individually. During implementation the address fed to constant memory has however simply been a delayed copy of the data memory address (compensating for the cache latency), and because of time limitation this is still the case.

Constant memory contents is set by using the desired constants in beginning of the assembler program, i.e. by setting register to immediate values. Had there been more time, a more convenient way to set the contents should have been constructed.

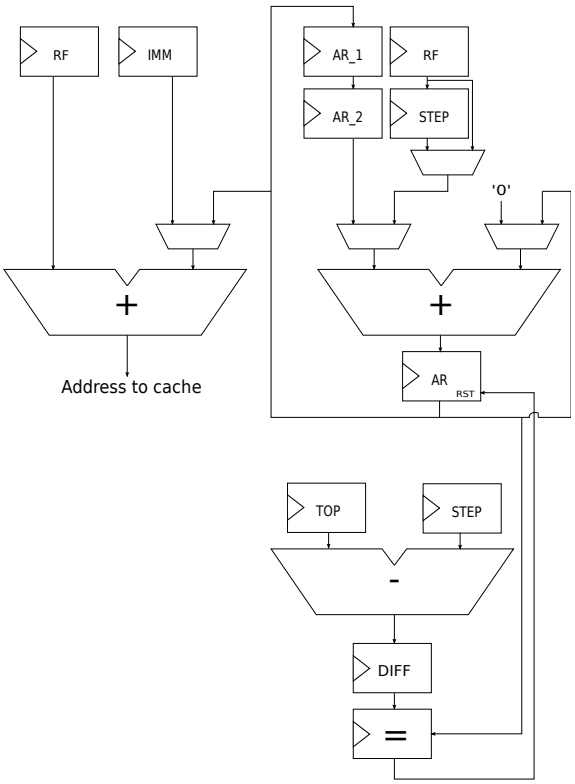


Figure 5.1: Address generator block diagram.

When adapting the MAC unit to the new AGU and convolution instructions it was mostly untouched and the non-AGU operation mode instructions still exist, but the adaption has caused differences in the behavior. As mentioned, the MAC unit was moved to the third execution stage, which creates a need to delay "normal" operands through first and second stage. For the same reason this delays the result two clock cycles, which can be considered wasted cycles. Before cache implementation, the timing between execution of assembler instructions was exact. This enabled the user to write several multiplications in a row in an assembler program, and read corresponding results in a row from the result register. With the cache however, there is no guarantee on the exact amount of cycles between execution of any two instructions, and later multiplications in a sequence can overwrite an earlier result before the corresponding read has been performed. As a test, a FIFO buffer was implemented for mitigation of this potential issue. While it solved the result reading timing problem, the large mux required to select between slots in the buffer was in combination with the rest of the read out logic, not surprisingly, too slow. In addition the buffer had to be rather large, about #(result bits) * #(slots) = 64 * 7 = 448 flip-flops. If such a solution is to be used, it should utilize a BlockRAM or 64 SRL16 (16-bit Shift Register LUT) primitives[34] instead

of flip-flops for storage.

5.5 Testing

First testing was done on the AGU only, with constant step size, no considerations to cache taken and some manually input values in order to check basic behavior and functionality. As the functionality depends so heavily on the interaction with the rest of the CPU pipeline, the AGU was integrated early before any further testing was done. After integration, testing of new instruction assembling and basic functionality was done with small assembler programs. The programs were then extended in order to trip cache misses and verify the cache miss handling, later in combination with circular addressing wrap around. Much of the testing has been rather manual, by looking at signals in Modelsim.

5.6 Results

The AGU in its current shape, seen in figure 5.2, has been synthesized to a maximum frequency of 333 MHz. The critical path is not surprisingly from AR_1 to AR, via the reset input. When integrated with the rest of the CPU, new critical paths arise, often related to the select signal to the mux before the last adder, which depends on cache miss signals (directly from the cache module) and internal AGU signals. The AGU is an optional part of the CPU, hence its negative impacts can be avoided until they are fixed.

5.7 Future Work

Due to time limitation there are still details to implement and things that can be improved:

- The CM and DM AGUs should be separated and be individually configurable. This is probably trivial to do.
- The signal to the AR reset input should come from a clocked register.
- The control signals to muxes and CE inputs are currently generated in a somewhat incoherent fashion. A more formal control FSM could be of use both for better understandability and identification of possible problems and optimizations.
- There is no way to read out the current address output from the AGU, but this should be implemented. If nothing else then to ease debugging.
- No handling of miss-predicted jumps exists, but some modifications to the cache miss handling will probably work in order to handle them. Though there are fewer cycles available for correction in this case.

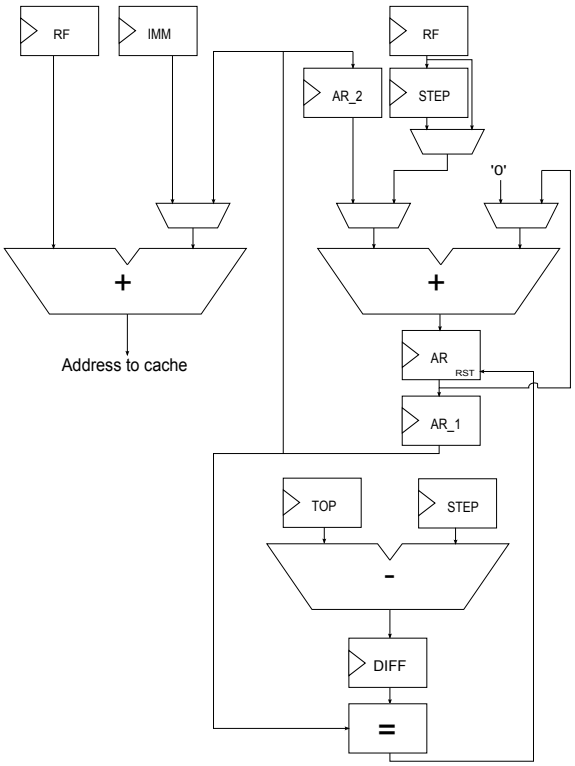


Figure 5.2: Block diagram of the current form of AGU.

- There is at least one known bug, which is related to reaching the circular top address very close to a cache miss and a certain pattern of instructions.
- If the double cache miss situation can be figured out and resolved, usage of the second memory port should simplify the AGU.
- Overall, the cache adds much complexity to the CPU and more thorough testing is needed. This is even more true for the AGU operation as it is so tightly coupled with the memory and operates in several pipeline stages of the CPU, stages that often are affected differently when e.g. a cache miss occurs.

Inspecting the list above above may lead to the conclusion that the AGU needs a different kind of architecture because of the amount of problems with the current one. It is however the author's opinion that a better control FSM and AR reset signal should be tried before scrapping the current architecture. Handling miss-predicted jumps could however add more control complexity to the last adder's mux, which would slow the whole CPU down when integrated.

Chapter 6

Interrupts

Interrupts is an important concept to improve processing efficiency in a computer system. Instead of letting the CPU actively poll a specific resource to detect a possible event, the resource itself can send an asynchronous signal to get the CPUs attention when needed. This signal is called an interrupt request (IRQ) and will be further described in this chapter, along with implementation details.

6.1 Overview

Interrupts can be divided into four classes [31].

- **Timer interrupts** - An interrupt is requested after a particular time period. Typically used for processes scheduling.
- **Input/output interrupts** - I/O devices are normally much slower than the CPU. Instead of letting the CPU waste thousands of clock cycles to wait for an I/O operation to complete the device can signal the CPU when the operation is done.
- **Hardware failure interrupts** - Used when a hardware node wants attention. For example when a computer system powers down or on memory read errors.
- **Program interrupts** - Invoked by execution of an instruction. This class of interrupts has its own name and will be referenced to as exceptions. Exceptions are described more in section 6.3.

When an interrupt is requested the current running program is put on hold and the corresponding interrupt handler (also known as interrupt service routine) starts to execute. An interrupt handler is a short program that handles the specific interrupt. For instance, in case of an I/O read interrupt occurs a specific interrupt handler copies the input data and stores it in kernel space or awakes the process that was requesting the data.

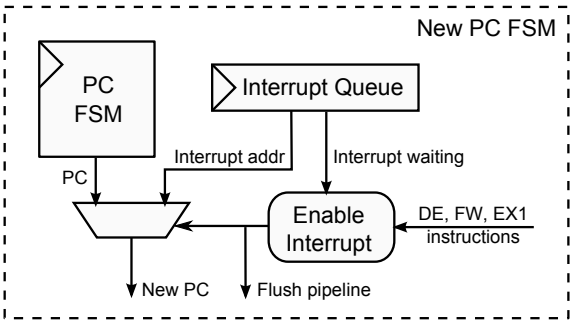


Figure 6.1: Basic implementation in the PC FSM.

Interrupts can have different priorities, meaning that an interrupt of a higher priority can have precedence over an interrupt with a lower priority.

6.2 Implementation

6.2.1 Starting the Interrupt Handler

The basic interrupt functionality was implemented almost completely in the Program Counter Finite State Machine (PC FSM). The IRQ interface to the FSM is one IRQ signal along with an IRQ address. The IRQ address represents the program memory address where the corresponding interrupt handler would be placed. When an interrupt is requested the IRQ address will be stored in a FIFO buffer and will be forced into the instruction flow as soon as possible (section 6.2.4 describes why this cannot be done directly). Currently the FIFO buffer can only hold one address, meaning that only one interrupt can be queued at this time.

See figure 6.1 for the basic idea of how interrupts were implemented in the PC FSM.

6.2.2 Saving Flags and Program Counter

Jumping to an interrupt handler in the middle of a program execution can have implications. For instance, if the interrupt handler does any kind of computation it might destroy the status flags bits. When the interrupt handler is done and the previous running program regains control and starts executing again it might run with modified status flags.

Consider a program that is executing inside a finite loop. At some point of each iteration the loop counter will be decreased by one. If the loop counter is non-zero a new iteration will be performed, otherwise the program jumps out of the loop. If an interrupt handler gains control and destroys the zero-flag in this crucial state (right before the zero checking) the program might jump out of the loop to

early (or run one iteration too much). This could have devastating consequences, including data corruption.

This problem was avoided by letting the PC FSM be able to read/store and set the status flags. Reading the status flag values was done by directly connecting the outputs of the flag registers to special flag storing registers in the PC FSM. In order to be able to set the status flags some minor changes were made to the flag module and a new set signal had to be connected from the PC FSM to the flag module. Because of the relative small changes no performance loss was noted.

The storing and setting of the flag registers was developed to be performed transparently by the PC FSM. The user does not have to worry about the flag registers at all. When an interrupt occurs the flags will be automatically saved and when the interrupt handler is done executing the flags will be automatically restored. In addition to saving the flags during an interrupt request, the current executing program address (program counter) also had to be saved. Otherwise there would be no address to jump back to after the interrupt handler exited. This was easily solved by adding another storing register in the PC FSM, in the same way as a status flag register.

6.2.3 Returning from an Interrupt Handler

Many processors uses the specific assembly instruction RTI (ReTurn from Interrupt) to return from an interrupt handler. Since the PC FSM already knows when an interrupt handler is running it felt unnecessary to introduce a new instruction, especially since the instruction space was limited. Thus, RTS (ReTurn from Subroutine) was used to return from the interrupt handler instead. RTI is still easy to implement in the future, and even necessary in order to support subroutine jumps inside of interrupt handlers.

In order to dynamically control the program flow a solution to read and set the registers described in section 6.2.2 (return flags and return program counter) were also implemented. Any interrupt handler can do these operations using the IN and OUT instructions. This functionality makes it possible for an interrupt handler to get an address from the data memory, overwrite the stored program counter and use RTS to jump to return to this new address. Section 6.4 includes an example that explains how a basic program scheduler was implemented using this functionality.

6.2.4 Instruction Flow Problem

As discussed in section 4.7, the instruction flow was already complicated prior to the interrupt implementation. Since interrupts don't need to be executed right away a decision was made to avoid making the instruction flow even more complicated. Instead of forcing a jump to the interrupt handler at the first possible occasion, a good opportunity is awaited. Logic was implemented to check the current running instructions to ensure that no jumps or memory operations is in the

pipeline. When such an opportunity exists the jump is performed and the decode pipeline stage is flushed using existing invalidate logic to make sure that no new instructions enter the pipeline until the interrupt handler starts executing.

While this solution requires very little hardware and doesn't worsen the complex instruction flow problems there is also one big drawback. If one section of a program consists of densely packed jumps and memory instructions it could take very long for an interrupt to get through.

6.3 Exceptions

As previous explained exceptions are classed as program interrupts. These are interrupts that somehow are software related. Exceptions can be invoked directly and explicitly by software (like system calls), or indirectly through invalid input data (like division by zero), page faults, etc.

Hardware interrupts can often be delayed for a small amount of time without affecting their purpose. This is however not the case for exceptions. Exceptions have to be handled directly as they arise. For instance, when a program wants to do a read system call, the execution of the program must be stopped immediately. Otherwise, a major program fault might be at hand if the consecutive instructions are using undefined data.

Because of this, exceptions could not be easily implemented in xi2. Because of the already complicated PC FSM and time constraints it had to be left out.

6.4 Tests and Results

In order to test the interrupt functionality a basic scheduler was built. The intent was to switch context between two user programs when a timer interrupt occurs and show that registers, flags and program counter were retained. A basic interrupt module was also built to generate these interrupts with a given interval.

The programs loaded into program memory were the following:

- **Initialization** Program addresses and states are loaded into memory. Interrupt handler address set to the scheduler.
- **Scheduler** Saves the state (flags, program counter, registers) of the current running program into memory, loads the state of the other program and returns.
- **Program 1** Computes N! ($N < 9$)
- **Program 2** Computes the Fibonacci sequence (0, 1, 1, 2, 3, 5, 8, 13, 21, ..)

The output from this test was satisfying and the simulation results were showing that everything was executing as expected.

Chapter 7

16-bit Multiplier and Pipeline Extension

The multiplication instruction that existed in xi2 prior to this thesis work uses the same hardware as the multiply-and-accumulate instructions, which lies outside the regular pipeline. This unit takes two 32-bit operands, produces a 64-bit result in five clock cycles and places the result in a special register. To use the result, it must be read in from this 64-bit register to two 32-bit regular registers. Enabling the new AGU described in chapter 5 further delays the result, for reasons explained in that chapter. This makes multiplication cumbersome, especially if the high precision isn't needed. The goal with the design presented in this chapter was to fit some kind of multiplier unit into the regular pipeline. In order to implement and evaluate this "short" multiplier, the CPU pipeline was extended with one stage between EX 2 and WB. Section 7.4 describes that work.

7.1 DSP48 Blocks

Both the MAC unit and the short multiplier uses instantiated dedicated multiplication blocks called DSP48. In addition to the multiplier they contain an adder, accumulation register, optional pipelining flip-flops and various options for interconnection of several DSP48 blocks. A sole DSP48 block in its simplest form takes two 18-bit operands and outputs a 48-bit result. It may seem undesirable to lock on to a vendor specific block in a specific FPGA model, but the alternative would be to realize the multiplier using general FPGA fabric, which for large general multipliers would mean a lot of LUTs consumed and many additions of partial sums to perform. Such a solution will most probably be both resource consuming and slow. In addition, dedicated multipliers are common in modern FPGAs and not using them is a way of wasting them.

7.2 Structure of 32-bit and 16-bit Multipliers

The MAC unit, which performs the 32-bit multiplication, utilizes several interconnected and pipelined DSP48 blocks. Further explanations of how it was built can be found in [15]. An important detail is that the specification of these blocks requires a certain amount of pipeline registers to be enabled for certain clock speed limits. Therefore, not much can be done about the five cycles without sacrificing clock frequency, precision or the usage of DSP48 blocks themselves. The 16-bit multiplier uses one DSP48 block, which at the targeted clock frequency range requires three internal pipeline registers to be enabled. The result register is 48 bits wide, but only the 32 LSBs are used, and the inputs are the 16 LSBs of operand A and B (sign extended to 18 bits for signed multiplication and zero extended for unsigned).

7.3 Resource Sharing

The DSP48 blocks have several operational modes that can be changed at runtime, and it would probably be possible to change the modes of one of the blocks in the MAC unit's structure to use it for the 16x16-bit multiplier, in order to save one block. This would however create code scheduling traps that the programmer must be aware of. It has because of time limitation not been investigated further.

7.4 Pipeline Extension

The execution stages had to be extended from two to three in order to fit the multiplier into the regular CPU pipeline, and other smaller structural changes in the CPU pipeline were necessary because of the integration of cache and the new AGU. This section presents these modifications and changes in the CPU behavior because of them. Figure 7.4 might be of help when reading.

7.4.1 Decoder

Parts of the forwarding work is done in the decoder, actually, this is where the basis for the control signals to the forward muxes are generated. One more execution stage meant one more set of results to consider for forwarding. It also meant one more "no match" signal to generate and modification of the existing ones. The "no match" (from here on called NM) signal is used to remove false positives generated by the forwarding logic. Another thing added was matching logic telling which results are ready in the third execution stage, and therefore available for forwarding from there.

In addition to mere extensions, one modification was done to the matching logic and registers. Instead of feeding the NM signal to the reset input of a register, it is now input as one of the bits of the register number to be matched. In the corresponding bit position in the other register number, zero is always input. This

way the desired behavior is achieved, whenever NM is true the matching result is forced to false, but the slow reset input is avoided. This is especially beneficial because the generation of NM signals is complex (therefore slow), and the reset input of the flip-flops is also slow. The now utilized bit position was previously completely unused and had no impact on the matching. It was present because of an earlier extension from 16 to 32 registers, done by another group of thesis workers.[6] 16 registers requires four bits addressing, which leads to a comparator fully utilizing two LUT4s. This solution can be seen in figure 7.1. 32 registers requires five bits, which leads to a comparator utilizing three LUT4s, but only two inputs on the third. This is where the "free" bit comes from. The new variant is illustrated in figure 7.2.

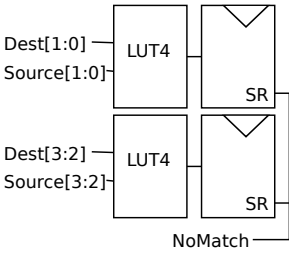


Figure 7.1: 4-bit matcher using SR inputs.

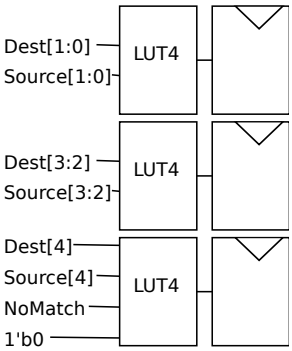


Figure 7.2: 5-bit matcher avoiding SR inputs.

7.4.2 Forwarder

The primary obstacle in the extension was the forwarder, not because of complex logic as much as that almost all of it consisted of instantiated FPGA primitives, where binary init values decided the logic function of the LUTs, and additional documentation was sparse. Step one was therefore to "translate" it back to readable RTL code. Adding a third execution stage *required* modifications of the forwarder, but there were some other issues taken care of at the same time. Because of cache

integration to the CPU pipeline and the new AGU's operation mode, the special memory ports on Data Memory used for AGU addressed data were removed, and that in turn removed the possibility to feed any execution unit with AGU data via the forwarder (for both Constant and Data Memories). This left some dead inputs in the forwarder tree. Also, instructions added by another group of thesis workers were at the moment hijacking results from the corresponding execution unit into the forwarder, outside the actual forwarder module.[6] In the end, the forwarder was more or less totally rewritten. After writing RTL code for the new tree, it was translated back to instantiated primitives as tight realization and placement is crucial for the overall CPU performance and the synthesis tools didn't seem to grasp that fact at all times. The new forwarder has on its own been synthesized to ~385 MHz and its basic structure can be seen in figure 7.3. What is not shown are OR-gates, optional inverters and swapping, which at least partially are realized in the same LUTs as the muxes. The inverters and swapping are pre-processing to the arithmetic unit.

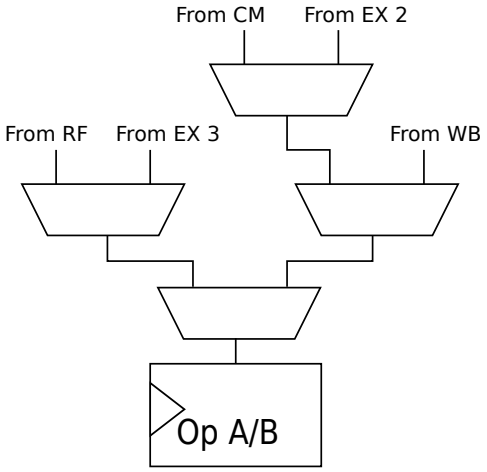


Figure 7.3: Basic layout of new forwarder.

7.4.3 Memory

When computing memory addresses, one part comes directly from the register file and not, as it might seem, from the forwarder logic. This must be kept in mind, especially now when the register write back stage is place one clock cycle later in the pipeline. This could possibly be solved by using the forwarders result, though the address calculation is a critical operation and might not tolerate the forwarder's delay.

7.4.4 Other Peculiarities

The CPU had, even before this thesis, a peculiarity in the subroutine handling: there is a lower bound on the number of instructions that has to be in a subroutine. This bound is pipeline dependent and is now seven instructions, one more than before. For shorter subroutines this means that insertion of useless NOPs is necessary. This has not been investigated further because of time limitations, though it certainly should be.

7.5 Resulting Pipeline

The resulting pipeline, with the 16x16-bit multiplier can be seen in figure 7.4

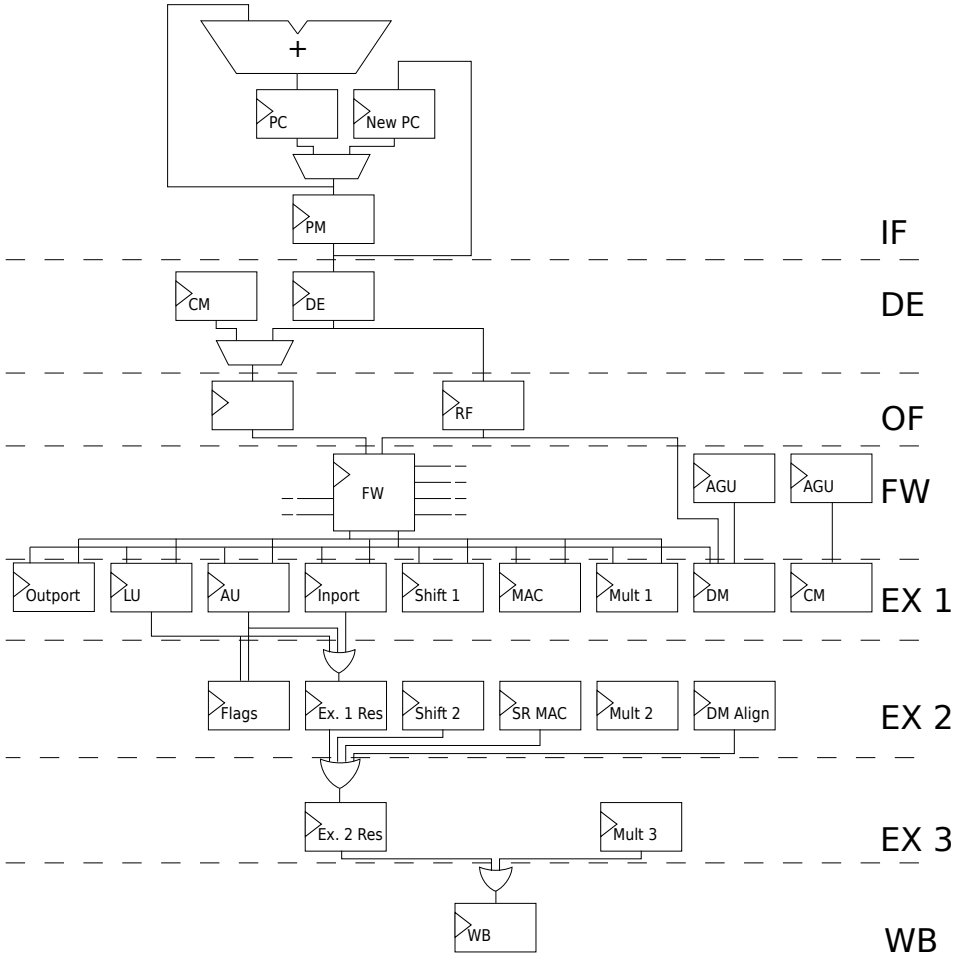


Figure 7.4: Xi2 with long pipeline.

Chapter 8

32-bit Integer Serial Divider

One of the identified short-comings of xi2 is the lack of hardware supported division. For fairly large operands, software implemented algorithms consumes many clock cycles and as seen in chapter 3, a divide instruction is probably expected to exist. This chapter will present some of the discussed ideas and implementations done to resolve this short-coming.

8.1 Hardware Division Algorithms

Initial research made it clear that there exist many different algorithms for hardware assisted division. From assisting a single division step to converging algorithms using e.g. Newton-Raphson. Basic algorithms work very much like "pen-and-paper" division and are comparably easy to understand. These can also be modified and extended in several ways to reduce the number of clock cycles required, and that's where things easily get difficult to grasp. SRT division is an example of an algorithm extending the basic *non-restoring* algorithm.[27]

The implementation focused on the most basic variants, and reasons for this were several. Initial guidelines from the supervisor, a quick glance at the performance of dividers of other soft CPUs (further discussed in section 8.5) and last but not least that even a simple, working, hardware divider is faster than doing all the corresponding work in software. A more advanced divider capable of running at acceptable clock frequencies certainly would have been an interesting challenge to implement, but the author's experience of the construction of arithmetic machineries was modest and as already mentioned, logic complexity must be kept low, multiplexers small, etc. This puts quite tight constraints on a design that would already be complicated enough. The used algorithms are presented in the following sections. In some cases, the following notation will be used: D for divisor, P for partial dividend, Q for the quotient and Qx for the currently generated quotient bit.

8.1.1 Restoring Division

Restoring and *non-restoring* division seemed to be the most popular algorithms to describe in various educational material for computer hardware. While both basically consist of a number of shifts and subtractions, there are some detail differences. The shifting is used to compare all possible up-scaled values of the divisor with the dividend, a basic principle is to right-shift the divisor one step per iteration. The subtraction is used for the actual comparison, to see if the current scaling of the divisor fits into the dividend. Figure 8.1 illustrates a typical flowchart for restoring division. Some more advanced pre-processing such as normalization is common. Often, the divisor is left-shifted until its MSB is at the same bit position as the dividend's. This reduces the number of times the following loop is taken, because the more exact up-scaling of the divisor is a substitution for comparisons where the resulting quotient bits are guaranteed to be a string of zeros.[12, 31, 27]

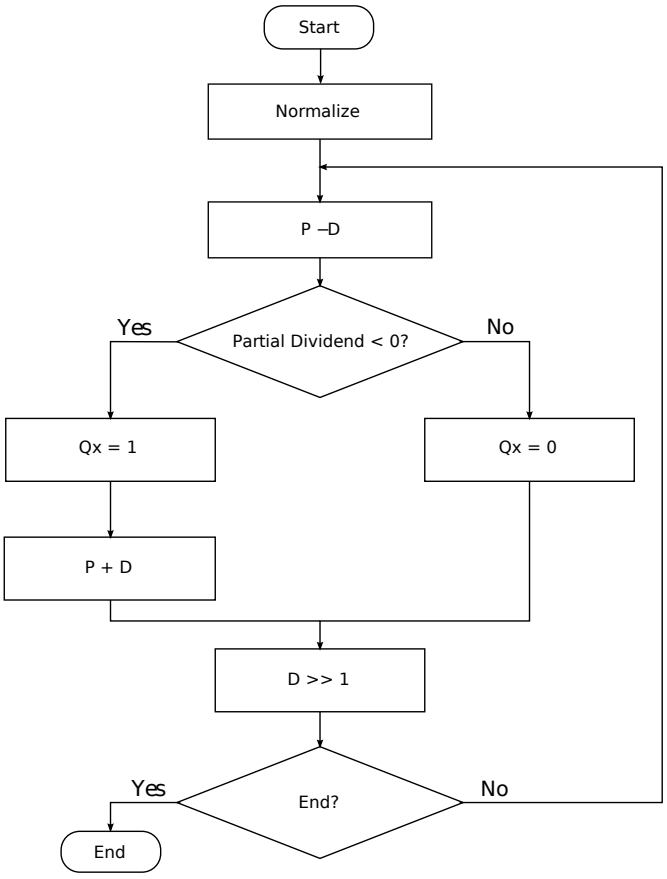


Figure 8.1: Flowchart for restoring division.

Each iteration generates one quotient bit, which is shifted in to the LSB of the

quotient. The number of iterations depends on the actual implementation and/or operands, and quotient bits not generated in the loop is in the case of normalized operands set to zero. If the result of a subtraction is negative, the current scaling of the divisor does not fit, and is added back to the dividend. Note the seemingly unbalanced tree with more operations in the "restore" branch. The restoring division can also be presented in this way:

- Normalize
- Repeat until end:
  - P - D
  - If result < 0: P + D
  - Generate and shift in Qx to Q LSB
  - D/2 (shift right one step)

8.1.2 Non-restoring Division

Non-restoring division exploits this fact in restoring division: if the subtraction result is negative, D is added back to P, followed by D/2 being subtracted from P in the next iteration. These operations can obviously be combined to +D - D/2 = +D/2. If the subtraction result had been positive, no restoration would have been done and the resulting corresponding operation is simply -D/2.[12] The combining results in a single operation in both branches of the division loop. The rest of the operations are just like in the *restoring* division variant. Figure 8.2 shows a flowchart for typical *non-restoring* division.

8.1.3 Algorithm Selection

As already mentioned, the implementation focused on the basic division algorithm. As the *non-restoring* variant appeared to be the faster of the two, the first implementation attempt used that one.

8.2 Integration With Existing CPU

As seen in section 3.2.2, the pipeline allows for a maximum execution time of two (or three, with the extended pipeline) clock cycles, without possibility to stall. Logic paths can't be too long either, as discussed in section 2.1, which excluded an approach with more logic operations between clocked registers. Thus it would have been impossible to fit even a relatively fast (in terms of clock cycles) 32-bit divider into the main pipeline, and the divider was therefore placed outside the regular CPU pipeline and communicated with using xi2's out and in ports.

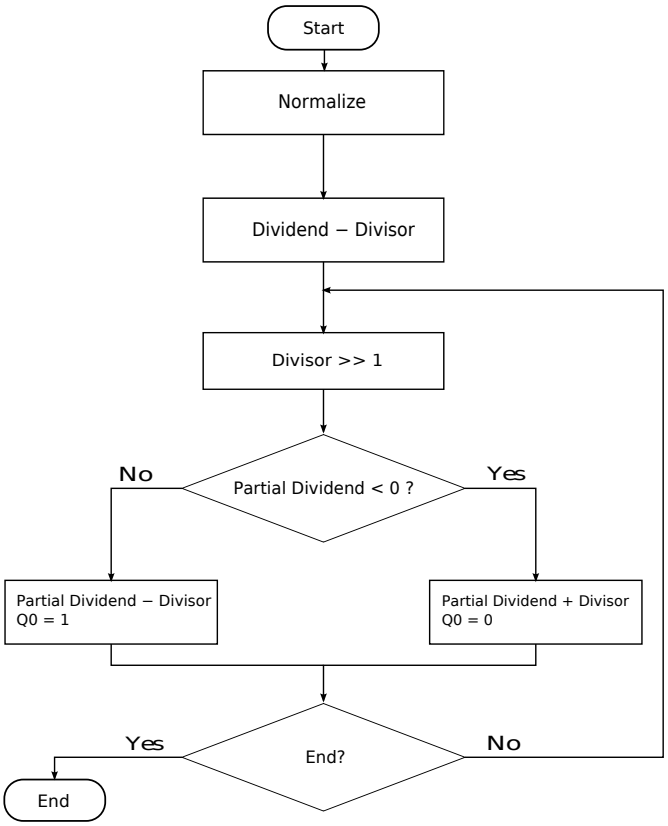


Figure 8.2: Flowchart for non-restoring division.

8.3 First Implementation

First implementation was done as a state machine, more or less directly from the books descriptions of the algorithm. From this state machine several operations with speed problems could be identified. First off, the normalization step required counting the number of leading zeros (CLZ) in both operands and then the computation of the difference between them. Another group of thesis workers had already implemented hardware for CLZ kind of operations [6], capable of running acceptably fast, and their ideas were used to implement this step. Rather heavy pipelining was required, which meant quite a few cycles spent on calculating the size of the following shift. Second speed problem was the shift operation itself, as the worst case is a 32-bit shift. Eventually the structure from the shifter unit inside the CPU was used, as it was known to meet timing constraints. Final speed problems lay, not surprisingly, in the repeated subtraction/addition and 1-bit shifting of the divisor in the main division loop. Pipelining the problems away by partitioning the add/sub into several sub-steps didn't seem like a good idea

here, because adding extra cycles to the loop would easily result in a ridiculous amount of cycles for the whole division process. Additionally the existing CPU had shown that a 32-bit addition plus some small extra logic should be possible to run at acceptable speeds. A somewhat successful approach was to (anyway) divide the loop into two stages. Both the addition and subtraction of D/2 to/from P were carried out in parallel in the first stage and the results temporarily stored. In the second stage, current P was examined and the decision which result (from subtraction or addition) to update P with was done, as well as divisor shifting and quotient bit generation.

8.3.1 Results

Besides using two clock cycles per bit in the division loop, this design uses one extra adder and all in all it is on the heavy side on both clock cycles and hardware usage.

8.4 Second Implementation

As it was the opinion of both the author and supervisor that a better design should be possible, it was time to rethink some steps in the original design. Primary changes and the result are presented below.

8.4.1 Removed Normalization

The idea behind first normalizing the operands is to skip iterations in the division loop. This is beneficial in a software implementation, where the result probably will be used as soon as the division is finished. But, when using hardware for the task the user would have to remember how many cycles to wait, for all combinations of operands, to benefit in a similar way. (Unless the pipeline is stalled or a construction where the user can wait for e.g. an interrupt to be fired is created. Since no such functionality existed at the time, this was not considered, but nothing hinders such modifications later on.) The realization of this, paired with the cycle expensiveness of the normalization computations, led to another approach. It meant comparing all up-scaled versions of the divisor with the dividend, but instead of shifting the divisor up and losing MSBs, the dividend is shifted down. (The same shifting of the divisor relative to the dividend is achieved.) In this design two 32-bit registers, named P MSW (Most Significant Word) and P LSW (Least Significant Word), are used for the dividend. P MSW is used as P in the division loop and P LSW stores the lower bits of the dividend, which are not yet used in the loop. At start, the dividend is placed in P LSW, and P MSW is set to zero. In the division loop, the MSB of P LSW is shifted into the LSB of P MSW each iteration. The loops runs 32 times regardless of operand values.

8.4.2 Algorithm Change

The problem with *non-restoring* division, at least when implemented in this FPGA, is that a decision to add or subtract D/2 next iteration must be taken immediately after the current addition/subtraction. Bit 32 (the sign-bit) in the partial dividend is used for this decision, the quotient bit generation and as operand in the following addition or subtraction, making it rather heavily loaded. As mentioned in section 8.3, two parallel adders were used instead of switching, but this also meant that the full partial dividend had to be routed to two adders, both in the critical path. As routing in general is a major problem, this was probably not favorable for overall performance.

In *restoring* division, the restore step can actually be easily "removed" by keeping the latest positive partial dividend and use that one next iteration if the current iteration produces a negative partial dividend. The 32nd bit of the partial dividend still participates in a decision, but now as a select signal to two muxes only. Furthermore subtraction is performed regardless of previous loop result and no switching or selection between add/sub exist.

8.4.3 Instantiation

The critical path lies in the carry-chain adder, from the selection of the second partial dividend bit to the generation of the 32nd result bit, as the 32nd bit decides which register to use as partial dividend (the current or stored result). The first partial dividend bit is always the MSB of P LSW, regardless of the previous result. Clearly, any extra logic between the 32nd bit and the selection logic, or extra complexity in the carry-chain, is unwanted. A LUT4 takes four inputs and that is exactly as many as the inputs to the selection logic and adder, as can be seen in figure 8.4. (Carry-in could be considered a fifth signal, but is a stable 1 and enters the first slice via another path.) This fact was nothing the synthesis tool used to reach targeted speeds though, instead creating some other structures. One placed an extra LUT before each bit selection, making it impossible to meet timing. Another structure can be seen in figure 8.3a. This version generates a signal with help of bit 32, to the carry-chain, in an external slice and breaks the timing that way. Both structures are probably caused by resource preservation. After several failed attempts to make the tool understand, the divider was partially manually instantiated with primitives according to the structure in figure 8.4. The shading shows the parts wanted in the same slice. Appendix A explains how the instantiation was done and the result is seen in figure 8.3b.

8.4.4 Results

A block diagram for the resulting design can be seen in figure 8.4. The underlying algorithm may not be that clear anymore, but hopefully the previous sections explain the steps taken to arrive at the resulting design. P MSW does not exist as a register on its own, it is either the current (named "P CURR") or the stored (named "P OLD") subtraction result and the MSB of P LSW must also be shifted

in to the stored result each iteration. The design has a maximum clock frequency of ~360 MHz and uses ~136 LUTs without external interfacing to the processor bus. No detection of zero-divisor exists and the result in such cases are undefined, this is partly due to the processor lacking a system for exceptions.

8.5 Related Designs

It has been somewhat difficult to make any good comparisons to other designs. Most of what are presented here will be of "apples to pears" type. Though together they may help to estimate where this design places itself in the ranks.

Most suitable is a comparison with the hardware used for division in Xilinx' soft CPU Microblaze. Maximum frequency for Microblaze in this FPGA is around 200 MHz[15] and the division instruction has a 32-34 clock cycle latency depending on configuration. There are no official numbers on the divider size and the source code is closed, but the hardware divider is an optional component and thus comparing the sizes of the CPU with and without it should give a hint of the size. The core was synthesized, with speed optimization selected, with and without the hardware divider in Xilinx Platform Studio 11 [35]. For full numbers synthesis numbers see appendix C. According to the synthesis numbers, the resulting divider causes 163 extra LUTs to used by the whole Microblaze CPU. It is assumed that the pipeline is stalled until the multi-cycle division is finished.[33] All-in-all, a design that is familiar.

Altera also offers an optional hardware divider, whose exact size is not known, in their CPU Nios II. In contrast to the dividers in Microblaze and xi2 it has a variable cycle latency, 6-68 clock cycles, and the ALU stalls until the division is done. This gives an advantage when divisions can be performed quickly. That a division can use up to 68 cycles is on the other hand peculiar and very punishing, as no other arithmetic operations can be performed meanwhile.[3]

Xilinx' own tool "Coregen" can generate various "pure" dividers, and some performance numbers are present in the documentation. Notable is that a Virtex-4 of speedgrade 10 is used as an example there, while the target speedgrade for this work was speedgrade 12 (faster). A generated 32-bit divider can run at a maximum frequency of 206 MHz, though it appears to be fully pipelined and able to produce one division result per clock cycle. It uses a huge amount of logic and flip-flop resources. It is not possible to generate a 32-bit divider which works in a similar way to ours (output one division result every 32nd clock cycle), however it is possible to generate an 8-bit divider which outputs a result every 8th clock cycle. That design has a maximum frequency of 304 MHz. Other possibilities in Coregen are to generate high radix dividers using DSP48 blocks. A 36-bit divider of such type has a maximum frequency of 301 MHz. While still using huge amounts of standard resources, it also uses thirteen DSP48 blocks.[32]

There is a parametrized, unsigned serial divider available at OpenCores, but it has not been possible to synthesize it correctly at the time of writing. Even so,

it is assumed to be a fair bit slower because for a full 32/32-bit divider it creates a 64-bit subtractor.[11] Another project at OpenCores offers a fully pipelined, parametrized divider, though it only does 2n/n-bit divisions. A 32/16-bit version of it reaches 346 MHz on a Virtex-4 speedgrade 12. Because of full pipelining it is heavy on resources.[19]

A table with the latency of some dividers in older (ASIC) CPUs was found in [18]. Most of these run at similar clock frequencies and have similar latency as the design presented in this chapter. Of course these are at the time of writing 16 years older designs and may also be more pipelined.

8.6 Testing

The module was first tested on its own, rather manually with the help of Modelsim, checking that signals behaved according to expectations. A Modelsim "do-file" was then constructed with some variations in operands, containing some thought out corner cases. Verification of results were still done manually. When the module later on was integrated with the system a larger and more convenient assembler test program was used. This contains some expected corners cases as well as other combinations of operands which had exposed peculiar bugs during the development, and comes together with a reference file with expected results.

8.7 Discussion & Conclusions

The Microblaze numbers are very close, except that the 160 MHz lower max speed. It does use more resources, but that number includes any additional logic needed when interfacing to the CPU. Division is a relatively rarely used operation in CPUs, which justifies using a low resource-approach. The pure dividers have only one purpose, which justifies high resource usage. What can be noted is that our design keeps up in clock frequency to the pure dividers, it's mostly the brute force from a fully pipelined high radix divider that is missing.

The non-restoring variant may have been possible to redo in a similar way to how the restoring was implemented, removing the normalization steps. The divisor (and carry in to the adder) would still have to be conditionally inverted, resulting in one more 32-bit register used and a mux in to the right side of the adder in figure 8.4, though the 32-bit register partial_old and the mux on the left side would be saved. It was said in 8.4.2 that a switching adder at the targeted speed and context seemed impossible, but with a clearer view of exactly were the problems lie and the same instantiation tactics used for the second implementation, it might not be.

The current implementation needs external handling of signed operands, which could possibly be handled internally. It is also based on a 32-bit adder, which means some combinations of operands produce invalid results. Using a larger adder

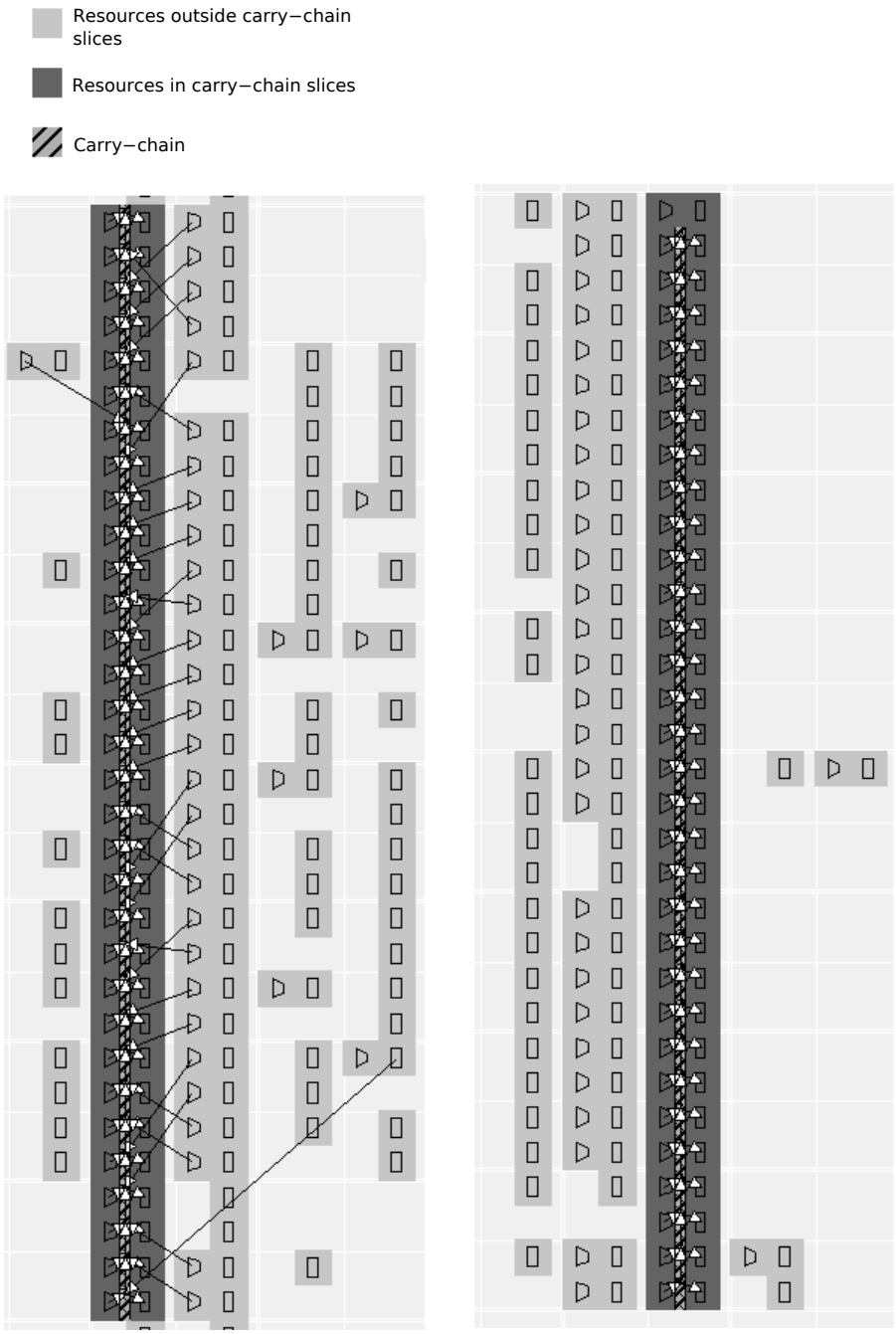
should mitigate that problem. An example of an error producing combination is one divided by $2^{32} + 1$ (or 2147483649, or 80000001 in hexadecimal notation), which should produce a quotient which is zero, but produces a quotient which is 4294967292 (or FFFFFFFC in hexadecimal notation.)

It can be argued that division where e.g. the divisor is a power of two can be done much faster. But in the end, advantages of detecting this and related "fast cases" and exploit them in xi2 will be few. Why was further explained in section 8.4.1.

8.8 Future Work

Possible future improvements to try are:

- In case exceptions are implemented, zero detection on the divisor should be added.
- Use a higher radix.
- Increase pipelining, which in turns requires an increased amount of adders.
- Using enhanced versions of basic division, like SRT.
- Implementing a totally different algorithm, like Newton-Raphson.
- Handle signed operands in a better way.
- Extend the adder one bit in order to handle all 32-bit operands.



(a) (b)

Figure 8.3: Snapshots from Floorplanner showing slice external and internal generation of dynamic signals to divider carry-chain.

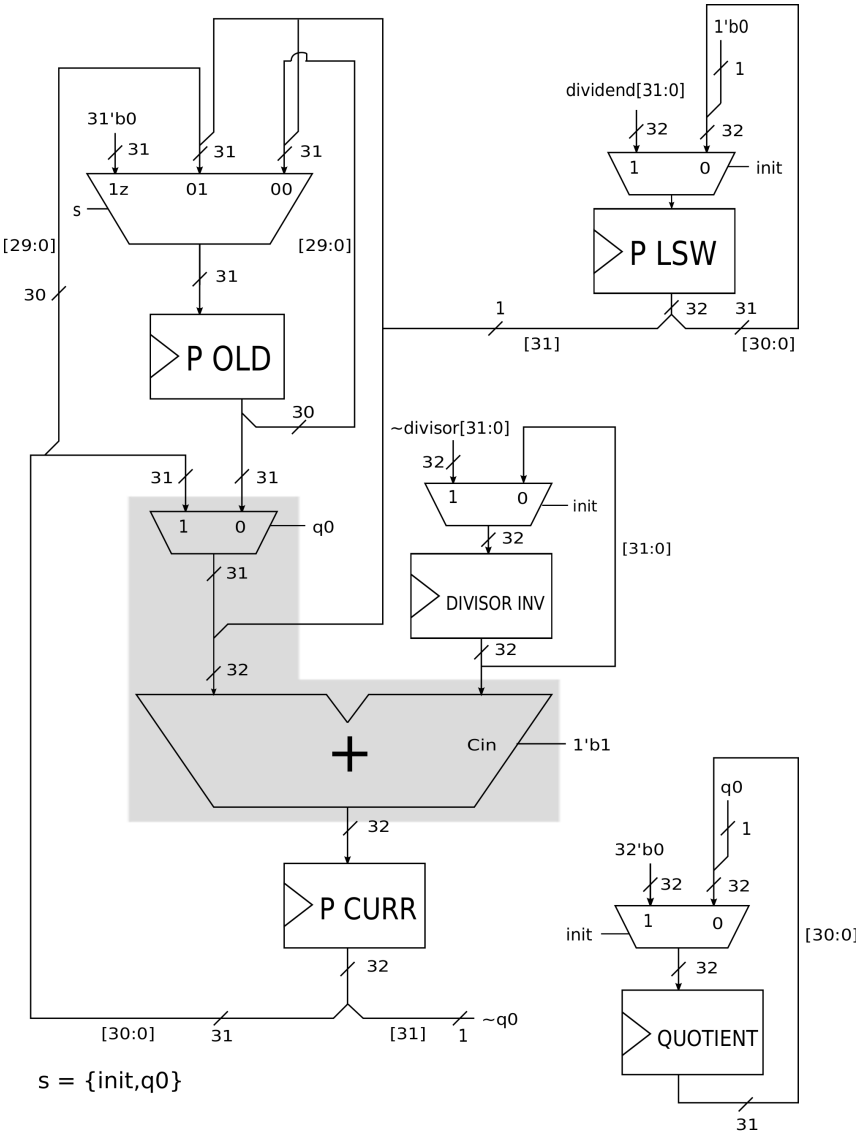


Figure 8.4: Block diagram of restoring division hardware. The shaded area marks functionality which should be realized with one LUT4 per bit for optimal performance.

Chapter 9

Miscellaneous Work

There have been some smaller modifications, additions and enhancements done around xi2. This chapter presents them.

9.1 Sign Extending Instruction

About the first work done was the implementation of a sign extending instruction. It takes an 8-bit or 16-bit value and copies the sign-bit to remaining upper bits. It was implemented in the logic unit according to listing 9.1.

Listing 9.1: Sign extension for 8- and 16-bit operands.

```
result[31:0] = {24{op[7]}, op[7:0]};
result[31:0] = {16{op[15]}, op[15:0]};
```

9.2 Port modifications

The in and out port modules have been modified for various reasons.

Several modules are communicated with using the outport. An identifier value in operand A decides which port the data found in operand B is destined for. This leads to several comparisons with, as it was, the eight first bits in operand A. The routing of operands from the forwarder is strained and keeping these comparisons as small as possible is beneficial. Before changes, the amount of bits in port identifiers and which port that had which identifier were hardwired throughout the CPU. These things are now handled by defined constants in single file. This simplifies port handling and comparisons can easily be kept just as large as necessary. The old outport was operating in EX 1 only, while the new one also operates in EX 2, for reasons presented in chapter 4. The new way also eases the strain on operand routing, as the loading of operand B to another register is always done and not dependent on operand A anymore. Instead, operand B and the result of

the port comparison are stored in EX 1, and the decision to output or not is taken in EX 2. A similar technique is used in the divider port handler, in that case only for performance reasons.

Apart from usage of constants for port names, the inport has not seen subject to any modifications, and still operates in a single execution stage.

9.3 Program Address Space Extended

The program address space has been extended to support larger programs. Previously, programs could only address up to 2^{16} instructions (256 kB). After the extension programs can now contain up to 2^{27} instructions (512 MB). The actual width of the address space is controlled by the configuration file. Any value from 16 to 27 bits is valid, but anything higher than 16 bits might affect the performance.

Using the maximum possible address space requires the largest set of instruction cache (18-bit tag + 7-bit index/set + 2-bit word select = 27 bits). It should also be noted that unconditional jump instructions still uses 16-bit addresses. This is due to the instruction format which has not changed to accommodate the larger address space.

9.4 External Work

Some modifications to the original xi2, listed below, were done by other thesis workers and integrated to this thesis' version of the CPU.[6]
Registers Increased number of internal general registers, from 16 to 32.
Instructions Instructions for counting leading ones and zeros were implemented, to be used as help-functions for floating point operands.

Chapter 10

Benchmarks

While xi2 can be synthesized to a very high frequency, little is known about the actual performance. This chapter will present some benchmark test that were executed and compared in order to get a rough idea on how well xi2 performs.

10.1 2D-DCT

10.1.1 Overview

The first benchmark is a program that performs a two dimensional Discrete Cosine Transform (2D-DCT or DCT-II). DCT is often used in signal and image processing, for instance JPEG compression.

Assembly code was translated and hand optimized from a 8x8 2D-DCT C-program written by Thomas G. Lane [22]. The C-program was based on a slightly modified version of an effective algorithm created by C. Loeffler, A. Ligtenberg and G. Moschytz [25]. It should be noted that the assembly code for OpenRISC 1200 was based on the same C-program but written and optimized by Johan Eilert. That code also included quantization.

10.1.2 Results

Xi2 was able to perform the 2D-DCT in 1223 clock cycles. OpenRISC 1200 needed somewhere near 2200 clock cycles to do the same work according to Johan Eilert. The value for OpenRISC is an estimated value after clock cycles for quantization have been drawn from the end result. Hence, there might be some uncertainty regarding the benchmark value for OpenRISC. Since the goal behind this benchmark was to give a rough idea of the performance of xi2 the estimated value was deemed sufficient enough. Figure 10.1 shows a comparison graph with cycle efficiency and total execution time for respective CPU.

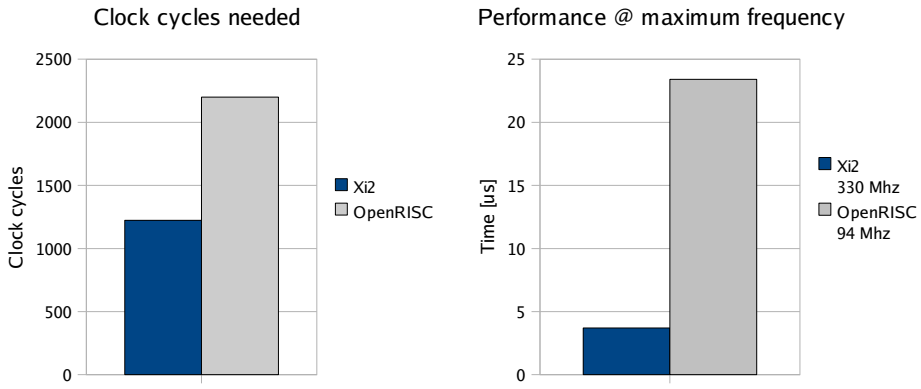


Figure 10.1: Benchmark charts for the 2D-DCT. Lower is better.

10.1.3 Conclusions

Although the accurateness of this benchmark can be questioned, there is no doubt about xi2 being much faster than OpenRISC when it comes to performing the chosen 2D-DCT algorithm. This is mainly due to the fact that OpenRISC stalls the pipeline on multi-cycle instructions. Multiplication instruction takes three cycles to complete (two stall cycles), and memory instructions take two cycles (one stall cycle).

10.2 Following a Linked List

10.2.1 Overview

A benchmark test that we believed would result in poor performance for xi2 was to follow a linked list to count the number of elements in it. This assumption was based on the load instruction, whose result is currently not forwarded. The length of the linked list was 16 elements in the test.

10.2.2 Results

Xi2 was able to follow the linked list in 167 clock cycles. The value for OpenRISC is unknown at the time of writing.

Chapter 11

Results and Discussion

Results of the thesis as a whole and a discussion on the basis of them can be found in this chapter. Individual modules' results have been presented in corresponding chapters.

11.1 Results

11.1.1 Final Pipeline

The resulting CPU pipeline with modifications and additions is illustrated in figure 11.1. The new instruction cache can be seen before the decode stage, replacing the program memory block, and the data cache in the execution stages, replacing the data memory blocks. The modified program counter is illustrated with a new block at the top of the pipeline. The new AGU is in the same place as the old one. If it is enabled, the MAC unit will begin execution in the third execution stage, not in the first, in contrast to what is illustrated. If the AGU is disabled, the MAC unit will as in the original xi2 begin in execution stage one. The new multiplier and its use of a three-stage execution pipeline can clearly be seen. Outport operations are extended to two stages. As the divider itself is situated largely outside the pipeline, its presence consists of a mere special result register in execution stage two. The sign extending instructions are executed by the logic unit and so their addition is not visible. Also added is the CL unit.

11.1.2 Performance

With all new features except AGU and instruction cache enabled, the CPU at the time of writing synthesizes to ~324 MHz. Figure 11.2 shows how the maximum frequency varies with diff rent timing constraints. With the AGU enabled it changes to ~298 MHz. Table 11.1 lists all modules of the CPU, together with their approximate f_{max} and area. Not all modules have been substantially modified or modified at all in the thesis, but all are included in order to provide a complete picture of the situation. The total area can *not* be calculated by summation of

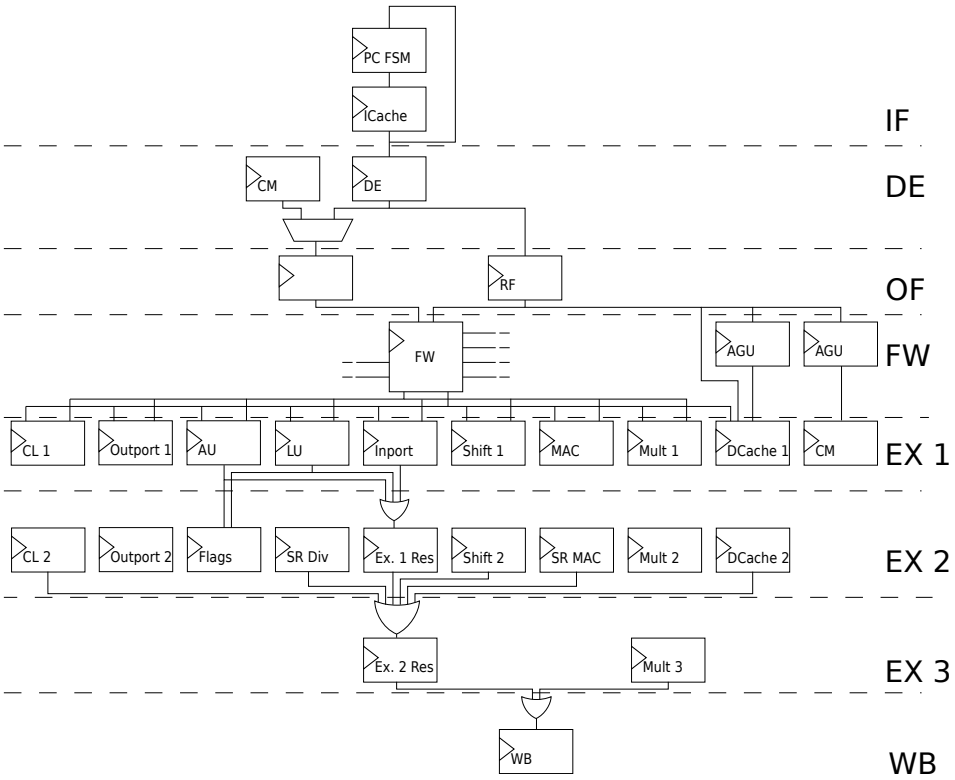


Figure 11.1: Xi2 with long pipeline.

the area column, as some modules instantiates others as sub-modules, e.g. the decoder instantiates several matcher blocks. Some modules have very high f_{max}, due to no combinational paths inside the modules themselves. The modules were synthesized for the non-AGU version of the CPU (except the AGU itself) with ISE 11.

At the time of writing, the instruction cache could not be enabled without a serious performance hit, even in direct-mapped mode. As it has previously been successfully integrated, the problems could be caused by some type of configuration bug.

11.1.3 Resource Usage

Resource usages for the original and extended versions of xi2 are found in appendix B. Those numbers are however including resources used by a wrapper used in synthesis, which are approximately 40 LUTs and 70 flip-flops.

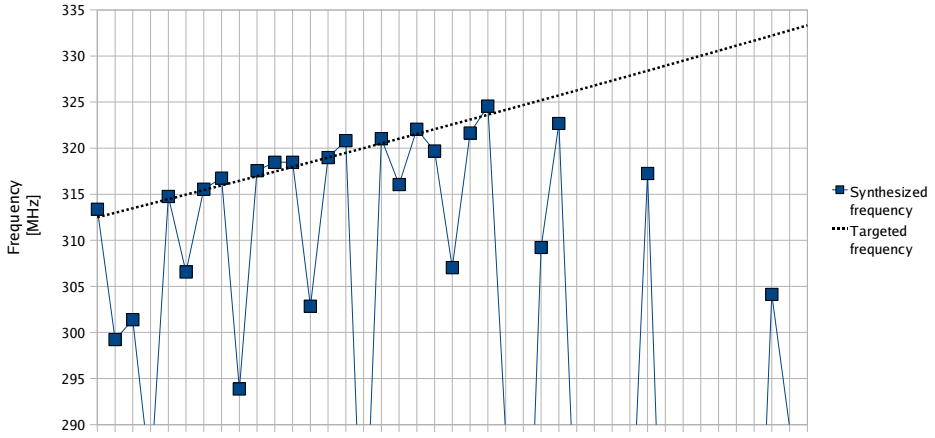


Figure 11.2: Maximum frequencies for different timing constraints.

11.2 Discussion

The CPU has a lower f_{max} after the additions. The new value is most likely not because of any individual block setting a definitive limit, as all of them, except the current AGU, have an f_{max} of 340 MHz or higher. When synthesizing a single module the tools have to optimize routing and placement of that module only. When synthesizing the whole CPU, several blocks need fast access to the same signals and resources in order to reach high speeds, problematic modules can need fast access to several signals, and modules are interdependent. The ability to achieve the best case for every module is, as expected, not possible and the tools have to make trade-offs. Similarly, modules that are much faster than needed offers room for worse placement and routing. For these reasons, merely adding more modules makes it more difficult to reach high frequencies.

A separate note on the new AGU: it has already on its own an unacceptably low f_{max}, but it causes larger problems than expected with the routing of the whole CPU, as the system frequency drops almost 40 MHz below the AGU f_{max}. This can partially be explained by the fact that both AGU and data cache are "black sheep", they are the two slowest modules in the system, and are heavily dependent on one another.

Are the additions worth the lower CPU f_{max}? While the somewhat low f_{max} of the caches guarantees that the whole CPU will achieve that frequency or less, a CPU without any memory caches is seriously hampered. The question is rather what organization of the caches to use. Since the 2-way set associative instruction cache might lower f_{max} too much (see chapter 4), the direct mapped organization is suggested. It is however hard to determine the best alternative, a lower cache miss rate can be of more value to the overall performance than a higher clock

Module	f_{max}	Area [#LUTs]	Comments
CPU	324	2962	Without insn. cache & AGU
AGU	333	98	New
AU	388	32	Modified
Const. Mem.	374	88	Modified
Data Mem.	457	49	Modified, Uses two BlockRAMs
Decoder	445	229	Modified
Data Cache	340	579	New, 4 kB two-way associative
Data Cache	341	497	New, 2 kB direct mapped
Divider	360	137	New
EX 1 Res.	665	32	Modified
EX 2 Res.	782	32	New
Forwarder	385	332	Modified
Inport	717	34	Modified
Insn. Cache	340	152	New, 4 kB two-way associative
Insn. Cache	351	135	New, 2 kB direct mapped
Interrupt Handler	426	37	New
LU	514	140	Modified
MAC Unit	500	115	Modified, Uses four DSP48s
Matcher	897	1	Modified
Mult. 16x16	500	2	Uses one DSP48
Outport	821	52	Modified
PC FSM	347	348	Modified
Register File	449	444	Modified, 256 LUTs as DP RAM
Shifter	439	381	Unmodified

Table 11.1: Estimated f_{max} and resource usage for the CPU and its modules.

frequency. Especially if the main memory latency is high. In the data cache case, using the direct mapped variant will most likely not make the new CPU f_{max} jump back up to around 345 MHz anyway, hence the associative variant might be the better choice, but as seen above the situation for the integrated data cache module appears to be more unforgiving than first thought, and the direct mapped organization should be considered. The added multiplier and divider do not seem to restrict overall CPU f_{max} themselves as they are largely decoupled from the system and run more than fast enough, giving extra routing freedom. The pipeline extension needed for the multiplier does however complicate things in the forwarder, but as can be seen in table 11.1 it should still be fast enough. The PC FSM has become quite complicated because of the new partial stalling functionality and support for interrupts. The execution flow is not so easy to follow anymore, especially not if many PC-changing events occurs at the same time (jump invalidate, conditional jumps, direct/unconditional jumps, instruction cache miss, data cache miss, interrupts). While a working solution with a small priority queue has been implemented, it is not fully verified. Some untested conditions can possibly cause erroneous execution. The new PC FSM is still considered to be a

better solution than to stall the whole CPU, which would had seriously lowered f_{max}.

Last but not least, even 300 MHz (which is a point the CPU f_{max} seems to hover around at the time of writing) is still a 50% higher clock frequency than what Xilinx' own CPU can manage in the same FPGA, and it should also be mentioned that the design currently contains very little floorplanning. Careful placement of certain parts have in the past proved to be helping performance and could probably still be utilized.

Chapter 12

Conclusions and Future Work

This chapter contains the primary conclusions drawn from the thesis work and ideas for future work.

12.1 Conclusions

Xi2 has been greatly improved throughout the thesis. In addition to numerous smaller improvements, xi2 now have configurable instruction and data caches, pipelined multiplications, fast divisions, support for interrupts and a new AGU. All of these features have, with some exceptions, been added without any significant impact on the maximum frequency. While there is no doubt much left to be done, xi2 has at least taken one step in the right direction.

Much experience has been gained. Some of the more concrete parts are listed below.

Insignificant changes can make a major difference to synthesis results. Even extremely minor changes appears to be able to change the tools' approach to how things should be synthesized. Sometimes that different approach makes a major difference for the end result.

Limited routing resources in an FPGA creates unusual limits. This becomes really apparent when adding modules to a design like this, where many modules need to communicate with each other.

Keep the resulting hardware simple. This could in turn make the RTL code less obvious and sometimes appear weird, but in order to get the simplest and fastest hardware solutions the synthesis tools appear to sometimes need really simplified or peculiar code structures.

It is not straight-forward to design for high FPGA speed.

To get the maximum performance you may have to get your hands dirty. Very model-specific adjustments for a certain FPGA will probably have to made in certain parts.

The trading of more hardware for more speed is not always possible. In fact, it can have the opposite effect.

12.2 Future Work

Much of the possible future work have been presented in previous chapters belonging to respective module. There are however some global CPU future work ideas worth mentioning.

- 32 instead of 16 registers makes the CPU larger in several areas, and in turn more difficult to route. The additions made in this thesis should be evaluated on a 16 registers variant of the CPU.
- The CL unit often shows up as problematic in timing reports, which could be a sign of a trouble-making module. It could ease the routing strain if the CL unit was removed.
- Fix the AGU according to what is stated under "Future Work" in AGU chapter, (5.7). This can hopefully increase the system performance as well, for reasons mentioned in section 11.2 and that chapter.
- Subroutines have a high minimal length (seven clock cycles), which should be lowered. Preferably to no more than the return instruction.
- The interrupt support is very basic and should be expanded.
- The cache modules does not support Virtex-II BlockRAMs and does not take advantage of the higher BlockRAM sizes in Virtex-5 and newer architectures.
- If any kind of operating system is to run on the CPU, exception support must be implemented.
- The CPU and its sub-modules are not thoroughly verified, which is something that definitely should be done. Especially the PC FSM needs to be verified.
- A compiler for at least one high-level language would be useful both for more rigorous testing and benchmarking as well as users.
- Resources could be saved by integrating the short multiplier with MAC unit hardware, thereby saving one DSP48 block.

Bibliography

[1] Altera. Annual Report, 2009. http://phx.corporate-ir.net/External.File?item=UGFyZW50SUQ9Mzc0ODV8Q2hpbGRJRD0tMXxUeXBlPTM=&t=1.

[2] Altera. Nios II core implementation details v.9.1.0, 2009. http://www.altera.com/literature/hb/nios2/n2cpu_nii51015.pdf.

[3] Altera. Nios II processor reference handbook. ftp://ftp.altera.com/outgoing/download/support/ip/processors/nios2/niosII_docs_9_0.zip, 2009.

[4] Altera. Device comparison, 2010. http://www.altera.com/cgi-bin/device_compare.pl.

[5] Ian D. L. Anderson, Mohammed A. S. Khalid, and Jason G. Tong. Soft-core processors for embedded systems. In *The 18th International Conference on Microelectronics (ICM)*, 2006.

[6] Olof Andersson and Karl Bengtsson. Adapting an FPGA-optimized microprocessor to the MIPS32 instruction set. Master's thesis, Linköping University, 2010.

[7] ARM. ARM and Thumb-2 instruction set quick reference card. http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf, 2010.

[8] ARM. Cortex-M1 processor, 2010. http://www.arm.com/products/processors/cortex-m/cortex-m1.php.

[9] Jason F. Cantin and Mark D. Hill. Cache performance for SPEC CPU2000 benchmarks, 2010. http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data.

[10] Marcus Christensson and Daniel Mattson. Evaluation of synthesizable CPU cores. Master's thesis, Chalmers University of Technology, 2005.

[11] John Clayton. Unsigned serial divider, 2003. http://www.opencores.org/projects.

[12] Alan Clements. *The Principles of Computer Hardware*. Oxford, 3 edition, 2006. ISBN 0-19-856453-8.

[13] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. Semantic Data Caching and Replacement, 1996. `http://www.vldb.org/conf/1996/P330.PDF`.

[14] Andreas Ehliar. *Aspects of System-on-Chip Design for FPGAs*. PhD thesis, Linköping University, 2008. ISBN 978-91-7393-848-8.

[15] Andreas Ehliar. *Performance driven FPGA design with an ASIC perspective*. PhD thesis, Linköping University, 2009. ISBN 978-91-7393-702-3.

[16] Andreas Ehliar. OR1200 performance. private communication, 2010.

[17] Michael J. Flynn. *Computer Architecture, Pipelined and Parallel Processor Design*. Jones and Bartlett Publishers, 1 edition, 1995. ISBN 0-86720-204-1.

[18] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. *ACM SIGPLAN*, 29(6):61–72, june 1994.

[19] Richard Herveille. Hardware division units, 2002. `http://www.opencores.org/projects`.

[20] Mark D. Hill and Alan Jay Smith. Evaluating associativity in CPU caches, 1989. `IEEEXplore`.

[21] Damjan Lampret. OpenRISC 1200 IP core specification, rev. 0.7, 2001. `http://opencores.org/websvn,filedetails?repname=openrisc&path=/openrisc/trunk/docs/openrisc1200_spec.pdf`.

[22] Thomas G. Lane. Implementation of 2D DCT algorithm described in [25], jfdctint.c, 2009. `http://www.ijg.org/`.

[23] Lattice. LatticeMico32 processor reference manual. `www.latticesemi.com/documents/doc20890x45.pdf`, 2007.

[24] Dake Liu. *Embedded DSP Processor Design*. Morgan Kaufmann, 1 edition, 2008. ISBN 978-0-12-374123-3.

[25] C. Loeffler, A. Ligtenberg, and G. Moschytz. Practical fast 1-D DCT algorithms with 11 multiplications. In *Proc. Int'l. Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 988–991, 1989.

[26] Grant Martin, Luciano Lavagno, and Louis Scheffer. *EDA for IC Implementation, Circuit Design, and Process Technology*. CRC Press, 1 edition, 2006. ISBN 978-0-8493-7924-6.

[27] Amos R. Omondi. *Computer Arithmetic Systems*. Prentice Hall, 1 edition, 1994. ISBN 0-13-334301-4.

[28] OpenCores. Wishbone System-on-Chip (SoC) interconnection architecture for portable IP cores, 2002. `http://opencores.org/downloads/wbspec_b3.pdf`.

[29] OpenCores. Opencores: Misson, 2010. `http://opencores.org/opencores,mission`.

[30] Steven Przybylski, Mark Horowitz, and John Hennessy. Characteristics Of Performance-Optimal Multi-level Cache Hierarchies. In *Proc. The 16th Annual International Symposium on Computer Architecture*, pages 114–121, 1989.

[31] William Stallings. *Computer Organization & Architecture, Designing for Performance*. Pearson Prentice Hall, 7 edition, 2006. ISBN 0-13-185644-8.

[32] Xilinx. Divider v2.0. `http://www.xilinx.com/support/documentation/ip_documentation/div_ds530.pdf`, 2008.

[33] Xilinx. Microblaze processor reference guide. `http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf`, 2008.

[34] Xilinx. Virtex-4 FPGA user guide, v.2.6, 2008. `http://www.xilinx.com/support/documentation/user_guides/ug070.pdf`.

[35] Xilinx. Platform studio and EDK, 2010. `http://www.xilinx.com/ise/embedded/edk_pstudio.htm`.

Appendix A

Instantiated Two's Complement Subtractor

Below are the ideas used when instantiating the subtractor used in the divider. "S" is the carry-mux select signal, generated in the LUT. S = 1 propagates the incoming carry, S = 0 selects the DI input, which can be an external input, one of the LUT inputs or a hardwired 1/0. Table A.1 lists the desired Carry out behavior, and figure A.1 shows the signals in a slice.

Opa	OpB	Cin	Cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table A.1: Carry out dependency on operands.

OpA	OpB	S
0	0	0
0	1	1
1	0	1
1	1	0

Table A.2: Carry propagation dependency on operands.

From table A.1 we get the propagation behavior in table A.2. When not propagating carry, OpA or OpB can be used. From this reasoning we get S = OpA$\oplus$OpB,

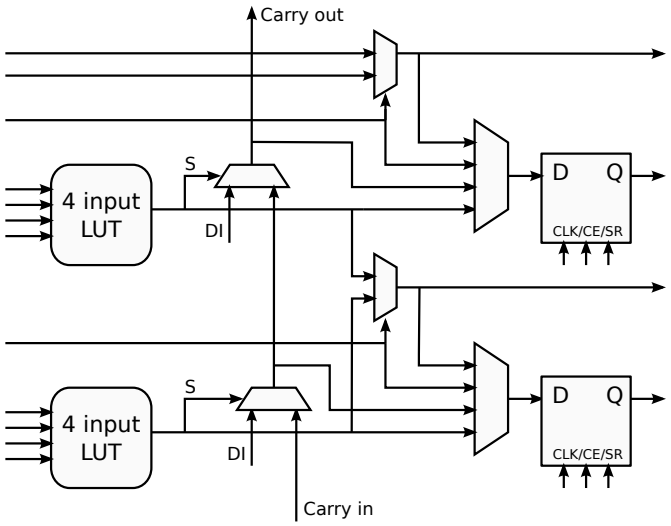


Figure A.1: Instantiated subtractor slice.

DI = OpB. S = OpA⊕OpB⊕Cin is the current result bit of OpA + OpB + Cin and is therefore input to the following flip-flop (the extra XOR gate is not shown in the figure). OpB is in the divider case always the divisor, which is static during a division run and the first input to the LUT. The MSB in the result of the most recent subtraction is the second input to the LUT. This signal decides if the most recent or the old subtraction result, which are the last two LUT inputs, is OpA.

Appendix B

Xi2 Synthesis Reports

All numbers taken from map reports.

B.1 Original

No floorplanning.

```
Design Summary
--------------
Number of errors:      0
Number of warnings: 1419
Logic Utilization:
  Number of Slice Flip Flops:         1,419 out of  71,680    1%
  Number of 4 input LUTs:             1,812 out of  71,680    2%
Logic Distribution:
  Number of occupied Slices:          1,280 out of  35,840    3%
    Number of Slices containing only related logic:   1,280 out of   1,280 100%
    Number of Slices containing unrelated logic:          0 out of   1,280   0%
      *See NOTES below for an explanation of the effects of unrelated logic.
  Total Number of 4 input LUTs:       1,814 out of  71,680    2%
    Number used as logic:             1,684
    Number used as a route-thru:          2
    Number used for Dual Port RAMs:     128
      (Two LUTs used per Dual Port RAM)

  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  Number of bonded IOBs:                  5 out of     768    1%
    IOB Flip Flops:                       4
  Number of BUFG/BUFGCTRLs:               1 out of      32    3%
    Number used as BUFGs:                 1
  Number of FIFO16/RAMB16s:               3 out of     200    1%
    Number used as RAMB16s:               3
  Number of DSP48s:                       4 out of      80    5%

Average Fanout of Non-Clock Nets:                3.18

Peak Memory Usage:  312 MB
```

```
Total REAL time to MAP completion:  10 secs
Total CPU time to MAP completion:   9 secs
```

B.2 Xi2 with Additions

No floorplanning, no AGU, no instruction cache.

```
Design Summary
--------------
Number of errors:      0
Number of warnings: 2125
Logic Utilization:
  Total Number Slice Registers:       2,126 out of  71,680    2%
    Number used as Flip Flops:        2,094
    Number used as Latches:              32
  Number of 4 input LUTs:             3,003 out of  71,680    4%
Logic Distribution:
  Number of occupied Slices:          2,371 out of  35,840    6%
    Number of Slices containing only related logic:   2,371 out of   2,371 100%
    Number of Slices containing unrelated logic:          0 out of   2,371   0%
      *See NOTES below for an explanation of the effects of unrelated logic.
  Total Number of 4 input LUTs:       3,005 out of  71,680    4%
    Number used as logic:             2,747
    Number used as a route-thru:          2
    Number used for Dual Port RAMs:     256
      (Two LUTs used per Dual Port RAM)

  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  Number of bonded IOBs:                  5 out of     768    1%
    IOB Flip Flops:                       4
  Number of BUFG/BUFGCTRLs:               2 out of      32    6%
    Number used as BUFGs:                 2
  Number of FIFO16/RAMB16s:               7 out of     200    3%
    Number used as RAMB16s:               7
  Number of DSP48s:                       5 out of      80    6%

  Number of RPM macros:            1
Average Fanout of Non-Clock Nets:                3.34

Peak Memory Usage:  419 MB
Total REAL time to MAP completion:  1 mins 35 secs
Total CPU time to MAP completion:   1 mins 33 secs
```

No floorplanning, no instruction cache, with AGU.

```
Design Summary
--------------
Number of errors:      0
Number of warnings: 2415
Logic Utilization:
  Total Number Slice Registers:       2,416 out of  71,680    3%
    Number used as Flip Flops:        2,384
```

```
    Number used as Latches:              32
  Number of 4 input LUTs:             3,168 out of  71,680    4%
Logic Distribution:
  Number of occupied Slices:          2,583 out of  35,840    7%
    Number of Slices containing only related logic:   2,583 out of   2,583 100%
    Number of Slices containing unrelated logic:          0 out of   2,583   0%
      *See NOTES below for an explanation of the effects of unrelated logic.
  Total Number of 4 input LUTs:       3,170 out of  71,680    4%
    Number used as logic:             2,912
    Number used as a route-thru:          2
    Number used for Dual Port RAMs:     256
      (Two LUTs used per Dual Port RAM)

  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  Number of bonded IOBs:                  5 out of     768    1%
    IOB Flip Flops:                       4
  Number of BUFG/BUFGCTRLs:               2 out of      32    6%
    Number used as BUFGs:                 2
  Number of FIFO16/RAMB16s:               8 out of     200    4%
    Number used as RAMB16s:               8
  Number of DSP48s:                       5 out of      80    6%

  Number of RPM macros:            2
Average Fanout of Non-Clock Nets:                3.22

Peak Memory Usage:  422 MB
Total REAL time to MAP completion:  1 mins 53 secs
Total CPU time to MAP completion:   1 mins 47 secs
```

No floorplanning, with instruction cache, with AGU.

```
 Design Summary
--------------
Number of errors:      0
Number of warnings: 2944
Logic Utilization:
  Total Number Slice Registers:       2,455 out of  71,680    3%
    Number used as Flip Flops:        2,423
    Number used as Latches:              32
  Number of 4 input LUTs:             3,296 out of  71,680    4%
Logic Distribution:
  Number of occupied Slices:          2,688 out of  35,840    7%
    Number of Slices containing only related logic:   2,688 out of   2,688 100%
    Number of Slices containing unrelated logic:          0 out of   2,688   0%
      *See NOTES below for an explanation of the effects of unrelated logic.
  Total Number of 4 input LUTs:       3,304 out of  71,680    4%
    Number used as logic:             3,040
    Number used as a route-thru:          8
    Number used for Dual Port RAMs:     256
      (Two LUTs used per Dual Port RAM)

  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  Number of bonded IOBs:                  5 out of     768    1%
    IOB Flip Flops:                       4
```

```
  Number of BUFG/BUFGCTRLs:               2 out of      32    6%
    Number used as BUFGs:                 2
  Number of FIFO16/RAMB16s:              10 out of     200    5%
    Number used as RAMB16s:              10
  Number of DSP48s:                       5 out of      80    6%

  Number of RPM macros:           3
Average Fanout of Non-Clock Nets:               3.28

Peak Memory Usage:  426 MB
Total REAL time to MAP completion:  2 mins 8 secs
Total CPU time to MAP completion:   2 mins 1 secs
```

Appendix C

Microblaze Synthesis Reports

The Microblaze cores were synthesized for Virtex-4 speedgrade 12 with speed optimization selected. The numbers below come from the map reports. Setup of environment and core were done rather quickly and the reports may not reflect a fully functioning Microblaze core. Compared to other known synthesis reports, the number on LUT usage in these report is, after all, highly probable.[10]

C.1 Microblaze with Hardware Divider

```
Design Summary
--------------
Number of errors:      0
Number of warnings:  477
Logic Utilization:
  Number of Slice Flip Flops:         1,196 out of  71,680    1%
  Number of 4 input LUTs:             2,109 out of  71,680    2%
Logic Distribution:
  Number of occupied Slices:          1,511 out of  35,840    4%
    Number of Slices containing only related logic:   1,511 out of   1,511 100%
    Number of Slices containing unrelated logic:          0 out of   1,511   0%
      *See NOTES below for an explanation of the effects of unrelated logic.
  Total Number of 4 input LUTs:       2,181 out of  71,680    3%
    Number used as logic:             1,696
    Number used as a route-thru:         72
    Number used for Dual Port RAMs:     384
      (Two LUTs used per Dual Port RAM)
    Number used as Shift registers:      29

  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  Number of bonded IOBs:                 34 out of     768    4%
  Number of BUFG/BUFGCTRLs:               3 out of      32    9%
    Number used as BUFGs:                 3
  Number of FIFO16/RAMB16s:               4 out of     200    2%
    Number used as RAMB16s:               4
```

```
  Number of DSP48s:                     3 out of      80    3%
  Number of DCM_ADVs:                   1 out of      12    8%
  Number of BSCAN_VIRTEX4s:             1 out of       4   25%

Average Fanout of Non-Clock Nets:               3.98

Peak Memory Usage:  409 MB
Total REAL time to MAP completion:  1 mins 26 secs
Total CPU time to MAP completion:   48 secs
```

C.2 Microblaze without Hardware Divider

```
Design Summary
--------------
Number of errors:      0
Number of warnings:  473
Logic Utilization:
  Number of Slice Flip Flops:         1,081 out of  71,680    1%
  Number of 4 input LUTs:             1,946 out of  71,680    2%
Logic Distribution:
  Number of occupied Slices:          1,373 out of  35,840    3%
    Number of Slices containing only related logic:   1,373 out of   1,373 100%
    Number of Slices containing unrelated logic:          0 out of   1,373   0%
      *See NOTES below for an explanation of the effects of unrelated logic.
  Total Number of 4 input LUTs:       2,019 out of  71,680    2%
    Number used as logic:             1,535
    Number used as a route-thru:         73
    Number used for Dual Port RAMs:     384
      (Two LUTs used per Dual Port RAM)
    Number used as Shift registers:      27

  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  Number of bonded IOBs:                34 out of     768    4%
  Number of BUFG/BUFGCTRLs:              3 out of      32    9%
    Number used as BUFGs:                3
  Number of FIFO16/RAMB16s:              4 out of     200    2%
    Number used as RAMB16s:              4
  Number of DSP48s:                      3 out of      80    3%
  Number of DCM_ADVs:                    1 out of      12    8%
  Number of BSCAN_VIRTEX4s:              1 out of       4   25%

Average Fanout of Non-Clock Nets:               3.93

Peak Memory Usage:  408 MB
Total REAL time to MAP completion:  1 mins 15 secs
Total CPU time to MAP completion:   44 secs
```

Upphovsrätt

Copyright