Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Adapting an FPGA-optimized microprocessor to the MIPS32 instruction set

Examensarbete utfört i Datorteknik
vid Tekniska högskolan i Linköping
av

Karl Bengtson, Olof Andersson

LiTH-ISY-EX--10/4323--SE

Linköping 2010

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

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

Adapting an FPGA-optimized microprocessor to the MIPS32 instruction set

Examensarbete utfört i Datorteknik
vid Tekniska högskolan i Linköping
av

Karl Bengtson, Olof Andersson

LiTH-ISY-EX--10/4323--SE

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

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

Linköping, 1 April, 2010

Avdelning, Institution
Division, Department

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

Datum
Date

2010-04-01

Språk Language	**Rapporttyp** Report category	**ISBN** —
☐ Svenska/Swedish	☐ Licentiatavhandling	**ISRN** LiTH-ISY-EX--10/4323--SE
☒ Engelska/English	☒ Examensarbete	
	☐ C-uppsats	**Serietitel och serienummer** Title of series, numbering
	☐ D-uppsats	**ISSN** —
☐ ________	☐ Övrig rapport	
	☐ ________	

URL för elektronisk version

http://www.da.isy.liu.se

http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-54680

Titel
Title

Anpassning av en FPGA-optimerad processor till instruktionsuppsättningen MIPS32

Adapting an FPGA-optimized microprocessor to the MIPS32 instruction set

Författare
Author

Karl Bengtson, Olof Andersson

Sammanfattning
Abstract

Nowadays, FPGAs are large enough to host entire system-on-chip designs, wherein a soft core processor is often an integral part. High performance of the processor is always desirable, so there is an interest in finding faster solutions.

This report aims to describe the work and results performed by Karl Bengtson and Olof Andersson at ISY. The task was to continue the development of a soft core microprocessor, originally created by Andreas Ehliar. The first step was to decide a more widely adopted instruction set for the processor. The choice fell upon the MIPS32 instruction set. The main work of the project has been focused on implementing support for MIPS32, allowing the processor to execute MIPS assembly language programs.

The development has been done with speed optimization in mind. For every new function, the effects on the maximum frequency has been considered, and solutions not satisfying the speed requirements has been abandoned or revised.

The performance has been measured by running a benchmark program — Coremark. Comparison has also been made to the main competitors among soft core processors. The results were positive, and reported a higher Coremark score than the other processors in the study.

The processor described herein still lacks many essential features. Nevertheless, the conclusion is that it may be possible to create a competitive alternative to established soft processors.

Nyckelord
Keywords

FPGA, Xilinx, Virtex-4, MicroBlaze, soft core processor, optimized, MIPS, MIPS32, Coremark

Abstract

Nowadays, FPGAs are large enough to host entire system-on-chip designs, wherein a soft core processor is often an integral part. High performance of the processor is always desirable, so there is an interest in finding faster solutions.

This report aims to describe the work and results performed by Karl Bengtson and Olof Andersson at ISY. The task was to continue the development of a soft core microprocessor, originally created by Andreas Ehliar. The first step was to decide a more widely adopted instruction set for the processor. The choice fell upon the MIPS32 instruction set. The main work of the project has been focused on implementing support for MIPS32, allowing the processor to execute MIPS assembly language programs.

The development has been done with speed optimization in mind. For every new function, the effects on the maximum frequency has been considered, and solutions not satisfying the speed requirements has been abandoned or revised.

The performance has been measured by running a benchmark program — Coremark. Comparison has also been made to the main competitors among soft core processors. The results were positive, and reported a higher Coremark score than the other processors in the study.

The processor described herein still lacks many essential features. Nevertheless, the conclusion is that it may be possible to create a competitive alternative to established soft processors.

Sammanfattning

FPGAer används idag ofta för stora inbyggda system, i vilka en mjuk processor ofta spelar en viktig roll. Hög prestanda hos processorn är alltid önskvärt, så det finns ett intresse i att hitta snabbare lösningar.

Denna rapport skall beskriva det arbete och de resultat som uppnåtts av Karl Bengtson och Olof Andersson på ISY. Uppgiften var att fortsätta utvecklandet av en mjuk processor, som ursprungligen skapats av Andreas Ehliar. Första steget var att välja ut en mer allmänt använd instruktionsuppsättning för processorn. Valet föll på instruktionsuppsättningsarkitekturen MIPS32. Projektets huvutarbete har varit fokuserat på att implementera stöd för MIPS32, vilket ger processorn möjlighet att köra assemblerprogram för MIPS.

Utvecklingen har gjorts med hastighetsoptimering i beaktning. För varje ny funktion har dess effekter på maxfrekvensen undersökts, och lösningar som inte uppfyllt hastighetskraven har förkastats eller reviderats.

Prestandan har mätts med programmet Coremark. Det har också gjorts jämförelser med huvudkonkurrenterna bland mjuka processorer. Resultaten var positiva, och rapporterade ett högre Coremark-poäng än de andra processorerna i studien. Slutsatsen är att det är möjligt att skapa ett alternativ till de etablerade mjuka processorerna, men att denna processor fortfarande saknar väsentliga funktioner som behövs för att utgöra en mogen produkt.

Acknowledgements

We would like to thank our supervisor and examiner Anderas Ehliar for letting us do this master thesis project. Without his help, technical expertise and curious anecdotes, the work would have been a lot harder, and certainly less interesting. Secondly, we would like to thank all the helpful people at ISY. Among the appreciable are Johan Eilert for his enlightening insights, and Anders Nilsson and Thomas Johansson for their helpfulness on technical matters. The members of our sister project — xi2dsp — Daniel Källming and Kristoffer Hultenius, also deserves recognition for cooperation and discussions. Last but not least we want to thank ISY for supplying the coffee (about 12kg in total) required to keep us going.

Contents

Chapter 1

Introduction

Synthesizeable processor cores provides interesting opportunities for digital designers. Combined with other blocks and custom logic, complete tailor-made SOC designs can be designed and implemented with a minimum of development time.

1.1 Problem description

A highly optimized soft core processor, capable of operating at frequencies well above the competition, has been designed by Dr. Andreas Ehliar. The processor is speed optimized for the Xilinx' Virtex 4 FPGA, by exploiting the properties of its hardware. Although the clock speed is impressive — 357 MHz, the lack of a compiler and the custom instruction set makes benchmarking of the processor difficult. Our initial task was to evaluate a few different instruction sets to find a suitable alternative to the one implemented by Dr. Ehliar.

The choice fell upon the MIPS32 ISA. The main task of the project has been to increase the usability of the processor by allowing code compiled by GCC to run as well as to evaluate the performance of the architecture itself by running benchmarks.

1.2 Assumed prior knowledge

This document assumes that the reader has a grasp of basic digital electronics and computer engineering. Familiarity with a Hardware Description Language (HDL) is also preferable. For engineering students at LiTH, at the time of writing, this roughly translates into students from the Y- and D-programmes having completed the mandatory courses, *Digitalteknik*, *Datorteknik* and *Konstruktion med mikrodatorer* or *Elektronikprojekt Y*.

Given these basic prerequisites, all additional background needed to fully appreciate this document is provided in Chapter 2. Students or researchers specializing in the field of computer engineering are encouraged to skip sections of Chapter 2 as appropriate.

1.3 Disposition

To aid the reader and facilitate an easier understanding of the structure and content of the document, a brief overview of each chapter follows.

- **Chapter 1: Introduction** Gives an introduction as well as providing an outline of the problems and tasks to be accomplished.
- **Chapter 2: Background** Provides background information necessary for the rest of the report.
- **Chapter 3: The XIPS Processor** Gives an architectural overview of the XIPS processor as well as describing interesting implementation details. This chapter can be considered both as a manual of the processor and documentation of the work carried out in this thesis project.
- **Chapter 4: Results** Contains performance statistics and comparisons. Also included are performance estimates for several features not yet implemented.
- **Chapter 5: Conclusions** Presents the conclusions of the thesis, and reflection on what could have been better.

- **Chapter 6: Future Work** Lists areas in need of further work.
- **Appendix A: DDR controller** Describes a memory controller that was created in an early stage of the project.
- **Appendix B: Performance Diagrams** Contains diagrams with comparisons with a larger number of processors.
- **Appendix C: xi2 instruction set** A listing of the xi2 instruction set.
- **Appendix D: XIPS instruction set** A listing of the MIPS32 instructions supported by the XIPS

1.4 Abbreviations

AGU Address generation unit

ASIC Application specific integrated circuit

AU Arithmetic unit

CLB Configurable logic block

CPU Central processing unit

CRC Cyclic redundancy check

DDR Double data rate

DSP Digital signal processor/processing

FPGA Field programmable gate array

GPU Graphics processing unit

GPR General purpose register

HDL Hardware description language

ISA Instruction set architecture

LU	Logic unit
LUT	Lookup table
MAC	Multiply and accumulate
MIPS	Several meanings. In this thesis the name of our ISA
MMU	Memory management unit
RAM	Random Access Memory
RISC	Reduced instruction set computer
SOC	System on chip

Chapter 2

Background

This chapter contains the necessary background to understand the rest of the thesis. The most important part is the section covering the xi2 processor. The work carried out in this thesis is entirely based on the general architecture of the xi2 and the previous work of Dr. Ehliar. In chapter 3 the XIPS processor is mainly covered in terms of differences compared to the xi2, making the xi2 section of this chapter invaluable for proper understanding. Readers who are well versed in the areas covered are encouraged to skip other sections as they see fit.

2.1 FPGAs

An FPGA (Field Programmable Gate Array) is an integrated circuit chip, which is built in such way that the logic behavior can be modified. It is designed to be programmed by the user, to implement desired functionality. The main idea is to fill the chip with a "sea" of various useful logic components, such as multiplexers, flip-flops, adders, and so forth. Practically, it consists of an array of so called CLBs, that can be connected to each other in the system of wires that run along the CLBs. A CLB consists of one or several small LUTs, which is nothing but small read-only-memories, and some sequential parts like flip-flops and possibly bigger blocks of RAM.

To make use of the FPGA, the desired behavior is expressed in a hardware description language (HDL). The most used HDLs are VHDL and Verilog, of which we use the latter. The functionality can then be tested using a simulation tool. Once satisfied with the behaviour in simulation, this is — in theory — where the user effort ends. The computer tools take over the job of making the FPGA resemble the HDL code. This process is called synthesis, and consists of three parts: mapping, placing and routing. Mapping will map the behavioral HDL code into the building blocks of the FPGA. The placing part will portion out the logic function into CLBs. The routing part then connects the CLBs to each other, and to some physical pins on the chip.

Thanks to having become cheaper, faster and denser throughout the years, an FPGA can now be a competitive alternative to an ASIC in many cases. Some examples follow:

Development: When developing a new product, the pre-mature product can be tested in real life, using an FPGA as a stand-in for the real chip.

First delivery: When time to market is a critical variable, a chip can be replaced by an FPGA, if the circuit in question to be used is not yet manufactured.

Small volumes: Due to the high one-time cost, ASIC usage is not always motivated. So if a company develops a small series of specialized units, the higher part-cost of the FPGA is a minor problem.

2.1.1 Development platform

The development in this thesis is targeted at the Virtex-4 [6] FPGA from Xilinx. The development board from Avnet [16] (product number ADS-XLX-V4SX-EVL35), includes several different useful chips, like memories, and different physical interfaces. For simulation we used Mentor Graphics ModelSim. The software tools we have used for logic synthesis are from Xilinx.

Virtex-4

Virtex-4 is a versatile FPGA for general use. Figure 2.1 shows the general architecture. A CLB consists of four so called slices. All four contains two lookup tables, two flip-flops, some muxes and some arithmetic logic. Two of the slices (Slice 0 and Slice 2 in the figure) can alternatively be used as shift registers or distributed RAM. The LUTs are of 4-to-1 type, and the block RAMs size is 16 kbits (18 kbits if counting the parity bits).

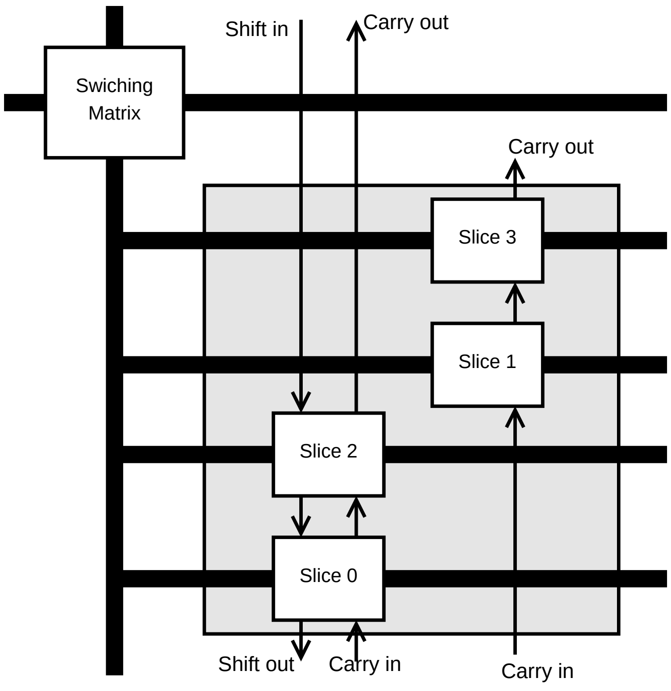


Figure 2.1. One CLB (the gray area) of The Virtex-4, in its surrounding environment

2.2 Softcore microprocessors

2.2.1 What is a soft processor?

The expression "soft" refers to the fact that the processor in full can be implemented using logic synthesis. This means that it can be integrated on the same chip as other logic designs, leaving a great flexibility compared to using a "hard" processor, which inexorably will demand its space on the circuit board. There are a lot of soft processors available for purchasing, as well as as open source alternatives. A soft processor can be synthesized for FPGA or ASIC, depending on the needs and purposes.

Due to the high performance of todays FPGAs, many classic CPUs (such as Zilog Z80, Intel 8080 and Motorola 68000) can be — and have been — reimplemented as softcore versions. This is often done as open source projects with merely esoteric purposes, but can also have practically interesting applications, such as porting an old system, that makes use of an old fashioned processor to an FPGA based platform.

2.2.2 Why use them?

As hinted, the possibility to integrate the processor on the same chip as other units has several advantages. The most obvious is that the need for an extra chip for the CPU vanishes. The dense design can allow for lower wire delays. Using a soft core processor also opens up for reconfigurability, allowing e.g. in-system bug fixing. Even if the processor is alone on the FPGA, there could still be some advantages over both ASIC solutions (such as time to market, and one-time cost) and over general purpose processor solutions (like performance and flexibility).

2.2.3 Notable softcore microprocessors

Some soft processors that are common in FPGA related projects will now be presented.

Microblaze

Microblaze is the name of Xilinx' own softcore processor. Its pipeline depth is configurable between 3 and 5 stages. Other things that can be configured include cache size, optional peripherals, MMU, and more. When configured for maximum speed, by using maximum pipeline, and logic partitioning optimized for low latency, the clock frequency can reach 235 MHz on a Virtex-5 [1]. The architecture is since 2009 included in the Linux kernel source tree.

OR1200

Perhaps the most well known open source processor and flagship project of the OpenCores initiative. It is an open source implementation of an architecture specification called OpenRISC 1000.

While fully open source and synthesizeable to an FPGA, it has not been optimized for such usage, and performance in FPGAs is lacking. OpenRISC 1200 can be found at the OpenCores homepage [3].

LEON

The European Space Research and Technology Centre designed the 32-bit LEON CPU, based on the SPARC-V8 architecture. It is written in VHDL, and available under LGPL, or as a purchasable product for commercial use. The current version, LEON4, is maintained by Gaisler research. The processor core uses a 7-stage pipeline and is very configurable. [10].

2.3 MIPS

The MIPS architecture was originally developed at Stanford University and is one of the first RISC architectures. The acronym stands for Microprocessor without Interlocking Pipeline Stages. The basic idea was to allow each sub-phase of the instruction to complete in a single clock cycle. This was a big departure from earlier designs, where different instructions required different amount of clock cycles to execute. By requiring all stages to take the same amount of time, the hardware could be better utilized and higher performance achieved.

The MIPS architecture has enjoyed great success in many different markets since the 1980s, but today it is mainly used in embedded devices. The architecture is a good example of a simple, clean RISC instruction set and the availability of many good simulators makes it a good choice for education purposes.

Several revisions of the instruction set exist. The first one being MIPS I and the latest ones being MIPS32 and MIPS64 (for 32 and 64-bit implementations respectively). The differences are minor and MIPS32 is basically a superset of MIPS I. For a detailed view of the MIPS architecture see [12].

2.4 Measuring processor performance

We measure performance in an attempt to determine fitness for a particular purpose. A processor can be exceptionally fast at performing a certain kind of computation but offer insufficient performance for a different task. For example, the main processor of a desktop computer is not specialized for any particular type of programs and tries to perform all tasks equally well, while excelling at none. A GPU on the other hand, is specialized towards the graphics related operations needed for advanced 3D graphics.

To really know how well a processor performs a certain task, one would ideally have to implement the specific algorithm on that specific processor. Naturally, this is an unfeasible approach for processor evaluation. Simple metrics such as clock speed provide a hint of performance, but is almost useless by itself. Average amount of cycles per instruction reveals a bit more. However, to really get an idea we must put the processor in motion — we must run a program on it.

By executing a mix of instructions corresponding to a real program we can get an estimate of the number of average instructions per clock cycle. However, a simple mix of instructions may not accurately model dependencies between instructions, which may or may not cause the processor to stall, leading to an optimistic performance estimate.

Benchmarks[1] are programs designed to measure the performance of an entire computer system or a part thereof. Compared to simple instruction mixes they better model inter-instruction dependencies and more accurately estimates performance. A synthetic benchmark performs no real work, but tries to mimic the operations performed by a real program, while an application benchmark performs a real, application specific, task.

Naturally, one benchmark does not fit all. As previously stated, performance is application dependent and benchmarking programs must take this into account. Choosing the right benchmark is an important first step. Designers of embedded systems need a different benchmark than the PC gamer looking for bragging rights among his peers.

In this thesis we are looking to measure general purpose integer performance. So we need a well established benchmark to measure that, preferably with clear reporting rules and a central source of scores.

2.4.1 Coremark

Coremark is a benchmark for testing and comparing processor cores, released and maintained by the EEMBC[2] [7]. The aim is to test the very core of the processor, i.e. regardless cache size, and so on. The size is supposed to fit in the cache memory, by being less than 16 kB for the program part, and less than 2 kB for the data part. Coremark is a synthetic benchmark and performs no real work. However, the individual parts use real algorithms. Hopefully, the use of

[1]The term originates from the marks on permanent objects land surveyors made to indicate the elevation at that point. They were used as references in further surveys.[20]

[2]Embedded Microprocessor Benchmark Consortium is a non-profit corporation who publishes general and application specific benchmarks for the embedded market. Member companies include many of the top vendors of processors for the embedded markets, such as ARM, Intel and IBM among others

common algorithms improves the benchmarks ability to correctly predict performance. A sequence of iterations makes one coremark test: Each iteration, four different tasks are performed; list finding/sorting, matrix manipulations, state machine processing, and CRC calculation.

Coremark scores

Coremark scores are reported as the number of Coremark iterations per second. This number can be directly compared with other processors to determine relative performance. Another interesting number is Coremark iterations per MHz, roughly equivalent to "amount of work carried out in a cycle" or a measure of how efficient the architecture is. While this number does not represent absolute performance it may be of interest for SOC designers whose maximum clock frequency is not limited by the processor but by some other component.

Reporting Coremark scores requires full disclosure of compiler version and compilation options used as well as compliance to a set of rules concerning run length etc. The EEMBC provides a central repository for submitting and comparing scores.

Figure 2.2 shows Coremark output when run on a desktop computer equipped with an Intel P4 at 3.0 GHz. Coremark is self-verifying, which means that the desired output for some specific seeds are known in advance, so the the program can check itself for correct output. This shows in the figure as the CRC check sums, followed by "Correct operation validated".

Coremark vs. Dhrystone

Dhrystone is another benchmarking program worthy of mention. It is a simple benchmark targeting the integer core of a processor, much like Coremark does. Like Coremark, it can also be made to run on almost any platform and is therefore widely used in the embedded systems world. Common though it may be, benchmarking using Dhrystone is not without pitfalls. The benchmark is highly susceptible to compiler optimization, allowing newer compilers to

```
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 14117
Total time (secs): 14.117000
Iterations/Sec   : 4250.194801
Iterations       : 60000
Compiler version : GCC3.4.6 20060404 (Red Hat 3.4.6-11)
Compiler flags   : -O2 -DPERFORMANCE_RUN=1  -lrt
Memory location  : Please put data memory location here
                   (e.g. code in flash, data on heap etc)
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0xbd59
Correct operation validated. See readme.txt for run and
reporting rules.
CoreMark 1.0 : 4250.194801 / GCC3.4.6 20060404
(Red Hat 3.4.6-11) -O2 -DPERFORMANCE_RUN=1  -lrt / Heap
```

Figure 2.2. Coremark output on a desktop PC (P4 3.0GHz)

optimize away large portions of work. While one would expect newer compilers to do a better job and increase performance for the Coremark benchmark as well, Coremark is designed in such a way that the compiler can never avoid actually doing the work. This fact, paired with the lack of reporting rules, makes comparing different Dhrystone scores difficult and of questionable value. Coremark was designed to be a successor to Dhrystone, providing the same benefits but without the well known problems.

All benchmarking done in this thesis uses Coremark. No effort has been made to run the Dhrystone benchmark nor exists any reason exert any. For more information on the problems with Dhrystone, please refer to [13]

2.5 Pipeline hazards

In a pipelined processor several instructions are processed simultaneously. For example, a new instruction is fetched at the same time as a second one is executed and a third ones result is written to the registers etc. A problem related to the pipeline architecture and the fact that several instructions are executed at once is called a "hazard".

2.5.1 Data hazards

A data hazard is caused by data dependencies between instructions. If the first instruction writes to the same register that the second reads from, the write may not complete before the read, resulting in incorrect data being read.

2.5.2 Control hazards

Control hazards are related to branching instructions. Branches are used to alter program flow and thus may or may not change the program counter. More specifically, before a branch instruction is completely executed it is hard to know from where to fetch the next instruction.

2.5.3 Structural hazards

This type of hazard occurs when two different instructions need to use the same hardware at the same time. For example, if different instructions uses execution pipelines of different length both instructions could theoretically try to write to the register file in the same cycle.

2.6 The xi2 processor

The xi2 processor was developed by Dr. Andreas Ehliar as part of his doctorate thesis. Initially meant as a high speed DSP optimized for Virtex-4 FPGAs, the focus shifted towards general purpose computing, after realizing that the performance could challenge established softcore offerings from major FPGA vendors.

A simple instruction set, with an instruction width of 27 bits flowed through a seven stage pipeline (a large number in the context — the Microblaze has three or five). The typical RISC instructions (ADD, SUB, J, etc..) were accompanied by some DSP instructions, like MAC. One notable property was the constant memory that allowed instructions to make use of 32-bit constants in an easy way. Refer to Appendix C for a complete listing of the xi2 instruction set.

The different units were manually optimized for high frequency, with some modules making extensive use of instantiated FPGA primitives in place of behavioral HDL code. This resulted in a design that can be clocked in 334 MHz without floorplanning and 357 MHz when floorplanning is used [18].

While this clock speed is highly impressive, the xi2 lacks many essential tools and features to make it useful in general. Most notable is the lack of a compiler. Stall functionality to deal with data dependencies between instructions is also missing.

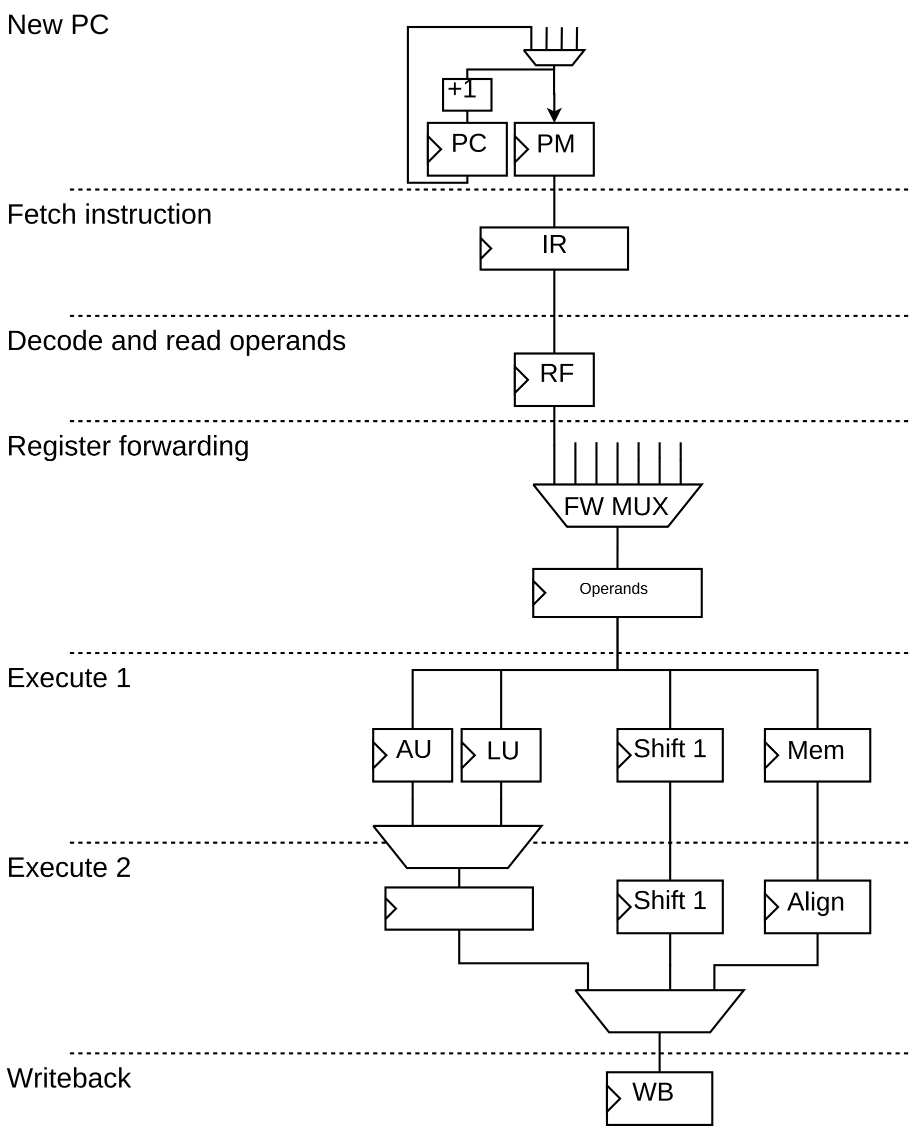


Figure 2.3. Overview of the xi2 pipeline

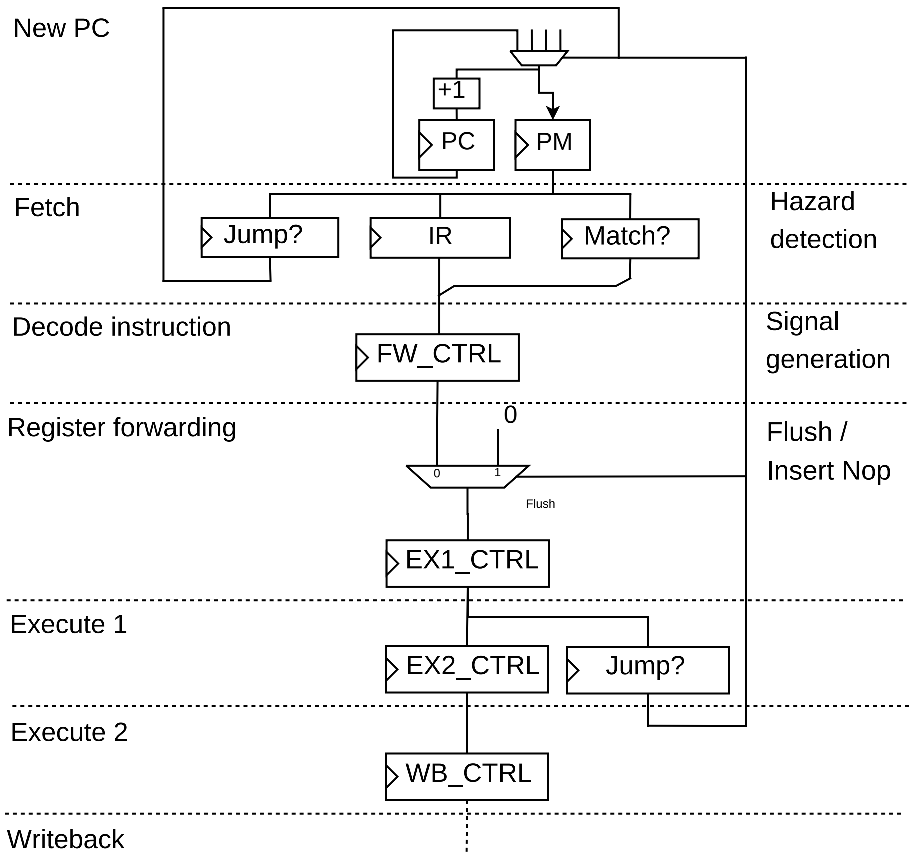


Figure 2.4. Overview of the xi2 control path

2.6.1 Pipeline architecture

Figure 2.3 shows a simplified view of the pipeline and 2.4 shows an overview of the control path. The pipeline stages are:

1. Calculate new PC (PC)
2. Instruction fetch (FE)
3. Instruction decode, read operands (DE)
4. Register forwarding (FW)
5. Execute 1 (EX1)
6. Execute 2 (EX2)
7. Writeback (WB)

Forwarding and loopback

To allow results to be used when they are ready, but not necessarily yet written to the register file, forwarding is required. An ADD instruction requires one cycle to complete and the results are ready in the EX1-stage. However, the results is not written until two cycles later, in the WB-stage. Without forwarding, an instruction following the ADD would have to wait until the result are written before the correct data is available. Forwarding takes advantage of the fact that the correct result does in fact already exist, and provides the means for the following instruction to receive the correct data. Forwarding can handle some of the data hazards that occur in the xi2 pipeline.

A product of the high degree of optimization is the FW-stage, which is necessary to keep the clock speed high. However, the use of a separate pipeline stage for forwarding means data will have to be ready at the beginning of the FW-stage instead of at the beginning of the EX-stage. This means that forwarding from an instruction with a latency of one cycle requires an additional instruction

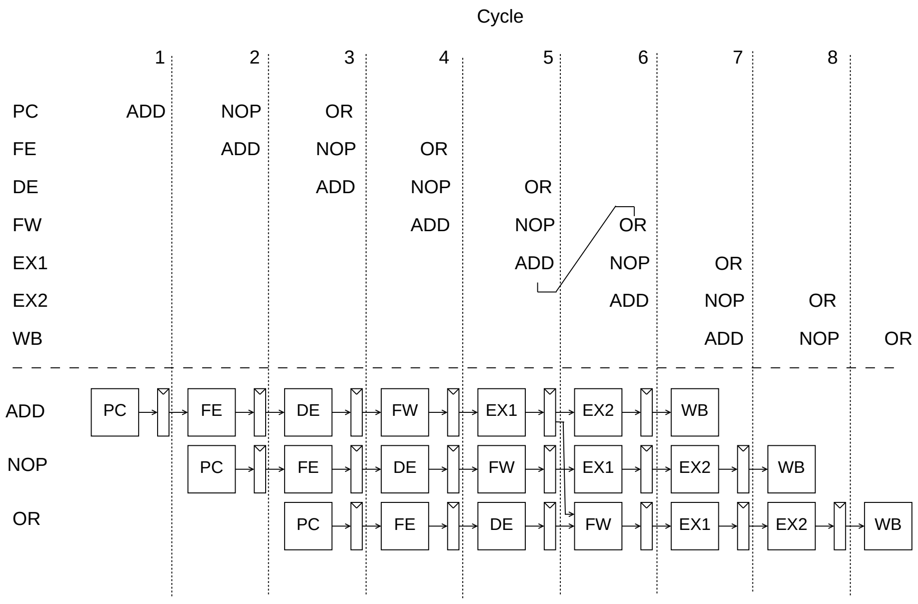


Figure 2.5. The OR requires the result of the ADD. The result is required in cycle 6, but will not be written to the register file until cycle 7. The processor will detect the data hazard and generate the correct signals in cycle 4 and 5. The forwarding mux will make the result from the EX1 stage available instead of the old value in the register file.

in between. Respectively, forwarding from an instruction with a latency of two cycles requires two instructions in between. Figure 2.5 shows an example of how forwarding works.

To alleviate this problem somewhat, a loopback mechanism is used. This allows results from the AU to be looped back and used as input for a following instruction also using the AU. Basically it is forwarding local to the execution unit. Loopback is also implemented for the LU, but is not possible between different execution units. See figure 2.6.

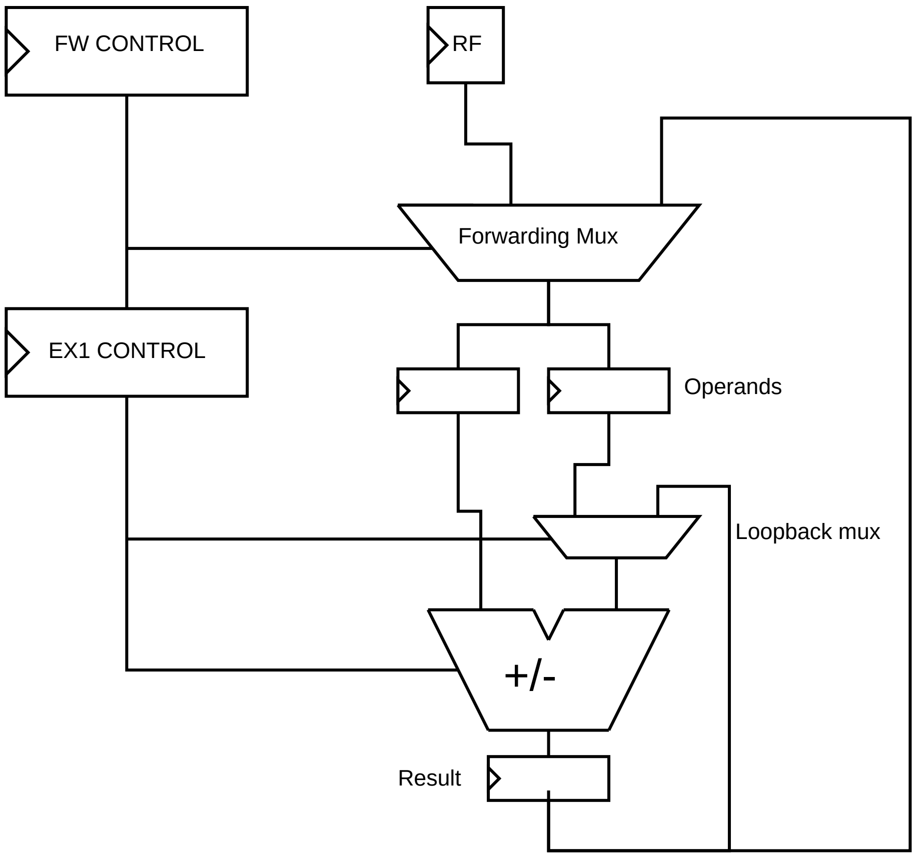


Figure 2.6. A simplified view of forwarding and loopback for the Arithmetic unit.

The result is that the xi2 processor requires the programmer (or the toolchain) to posses knowledge of the pipeline architecture. Instruction sequences not supported by forwarding or loopback will produce unexpected results as the processor will use the old register values. This is one of the main drawbacks of the xi2 processor and one that must be resolved to allow execution of compiled code.

2.6.2 Branches

The xi2 uses flags for jump decisions and a special bit in the instruction word for branch prediction (predict taken or predict not taken). To ensure that the previous instruction has time to modify the flags before a conditional jump reads them, jump decisions must be made in the EX1-stage. If the branch was wrongly predicted the pipeline will be filled with instructions that should not be executed. These instructions must be flushed to ensure they do not write incorrect data to the registers. Flushing takes place between the FW-stage and the EX1-stage and flushed instructions will be replaced with NOPs. The number of instructions to flush depends on whether the branch was predicted as taken or not and is either 3 or 4 cycles. Correctly predicted branches suffer no penalty.

Consider the example program below. In the example BNE is predicted as not taken, which will turn out to be incorrect, so instructions A1 to A4 will have to be flushed, and the program counter redirected to *label*, as illustrated in figure 2.7.

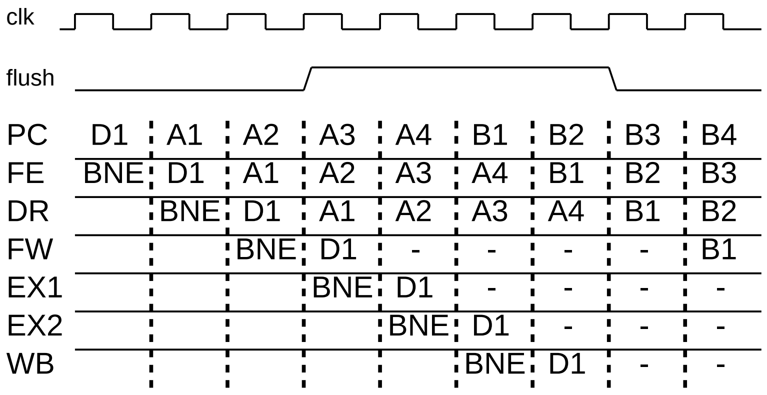
Delay slot

All branch instructions have a delay slot. This means the instruction directly following the branch will be executed, regardless of whether the branch is taken or not. This is necessary because the final branch decision is made late in the pipeline and the following instruction has already entered the EX1-stage.

2.6.3 Multiply and Accumulate

Performing fast multiplication requires the use of special DSP48-slices in the FPGA. These blocks provide acceleration for common DSP-tasks, such as multiply-and-accumulate.

```
          ...
          BNE label
          D1
          A1
          A2
          A3
          A4
          ...
label:    B1
          B2
          B3
          B4
          ...
```



PC	D1	A1	A2	A3	A4	B1	B2	B3	B4
FE	BNE	D1	A1	A2	A3	A4	B1	B2	B3
DR		BNE	D1	A1	A2	A3	A4	B1	B2
FW			BNE	D1	-	-	-	-	B1
EX1				BNE	D1	-	-	-	-
EX2					BNE	D1	-	-	-
WB						BNE	D1	-	-

Figure 2.7. Example of a flushed instruction sequence

One DSP48-slice contains one 18x18-bit multiplier, so one slice is insufficient for 32x32-bit multiplication. However, four of them can be combined to provide the functionality. Consider two 32-bit numbers that we wish to multiply:

$$r_{63..0} = a_{31..0} * b_{31..0}$$

If we consider

$$r_{lolo} = a_{15..0} * b_{15..0}$$

$$r_{lohi} = a_{15..0} * b_{31..16}$$

$$r_{hilo} = a_{31..16} * b_{15..0}$$

$$r_{hihi} = a_{31..16} * b_{31..16}$$

This can be written as:

$$r_{63..0} = r_{lolo} + r_{lohi} * 2^{16} + r_{hilo} * 2^{16} + r_{hihi} * 2^{32}$$

This approach requires 4 16x16-bit multipliers and 4 32-bit adders, and thus fits nicely into four DSP48-slices. 32x32-bit multiplication is not enough, the hardware must also be able to perform multiply-and-accumulate.

$ACC_i + a_i * b_i = ACC_{i+1}$
$ACC_0 = a_0 * b_0$
$ACC_1 = a_0 * b_0 + a_1 * b_1$

This can be achieved by accumulating the partial sums and finalizing the result when needed. So for one parital sum:

$$r_{lolo_i} = a_{15..0_i} * b_{15..0_i} + r_{lolo_{i-1}}$$

After accumulating the partial sums, we arrive at the final result by adding them together:

$$r_{63..0_i} = r_{lolo_i} + r_{lohi_i} * 2^{16} + r_{hilo_i} * 2^{16} + r_{hihi_i} * 2^{32}$$

Note that we can not accumulate and compute the final result at the same time without needing 3 more adders. The xi2 uses a special "Finalize" instruction, that the programmer can use to indicate when accumulation is done and the final result should be calculated.

Figure 2.8 illustrates how ordinary 32-bit multiplication, as well as multiply-and-accumulate, can be achieved by using four DSP48-slices.

2.6.4 Optimization

As evident by the high clock speed, the xi2 is highly optimized for speed. To achieve this, signal timing has been considered during all phases of development. This section covers specific optimization in the xi2 as well as performance optimizations for FPGAs in general.

To optimize for an FPGA one must possess knowledge of its inner workings and use its structure to ones advantage. The function is configurable, but the actual hardware is not. The LUT is there and area can not be saved by only using half of it. This is one of the main things to consider when optimizing for FPGAs, the hardware is fixed so it is up to the designer to make do.

The development tools will try to optimize the design. If this is insufficient one must intervene at the appropriate place in the process. Optimizing a design for FPGAs will generally go trough these steps:

- Algorithm optimization — The first step is to choose an efficient way to perform the desired calculation.
- Pipelining — This includes splitting the logic into several clock cycles, while trying to balance the amount of computation between pipeline stages. The tools has functionality to help with this, but the rough partitioning is usually done by hand.
- Logic synthesis — This stage maps the constructs of the HDL into the building blocks of FPGA. Normally performed by software tools.
- Placement — Chooses appropriate places for the synthesized logic. Normally performed by software tools.

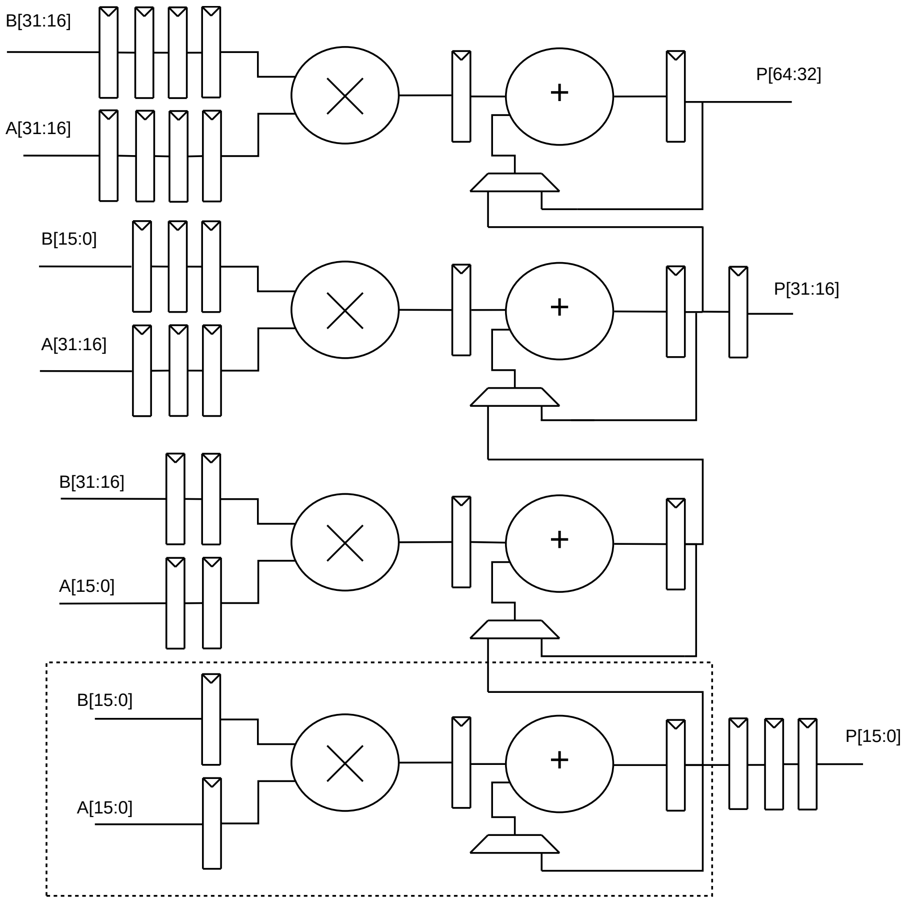


Figure 2.8. In multiply mode the muxes will choose the input from the DSP48-slice directly beneath it, and will finalize the sum. By choosing the input from the output of the adder, accumulation can be achieved. In the multiply-and-accumulate case, the "Finalize" instruction will perform the final summation, to compute the final result. The outline shows what is basically one DSP48-slice. The number of registers on the inputs and outputs as well as pipeline registers inside is configurable.

- Routing — Connects the building blocks to form the complete function. Normally performed by software tools.

This list also indicates relative importance. A poor choice of algorithm cannot be compensated for by optimal logic synthesis and perfect placement and routing.

Pipelining

The xi2 owes much of its speed to heavy use of pipelining. By dividing instructions into smaller and smaller chunks of logic the clock speed can be increased. Pipelining too much, however, will lead to other problems that will limit performance, for example data hazards, control hazards, and growing size of the design. A longer pipeline will generally lead to higher penalties for mispredicted jumps as well as a larger amount of stalls due to inter-instruction dependencies. Naturally, this is subject to application specific variations and the trade-off is not trivial.

Logic synthesis - FPGA primitives

Modern logic synthesis tools generally do a fairly good job at translating behavioral HDL code into the logical elements of an FPGA. For a highly optimized design though, this might not be sufficient.

To achieve the highest performance, it is sometimes necessary to "synthesize the logic function by hand" and explicitly instantiate the actual building blocks of the FPGA. A good analogy of this would be that of a software programmer, who is unhappy with the way the compiler handles his code. In an attempt to improve performance he rewrites parts of his code using inline assembly. Both the FPGA designer and the programmer have to leave the comfort of their high level language and delve into the realm of hardware specific details. Just as the assembly language differs from processor to processor, the logical building blocks differ from FPGA to FPGA.

Several parts of the xi2 contains large number of instantiated components. Two notable examples include the arithmetic unit and the Forwarding mux.

Placement - Floorplanning

Another compelling reason to instantiate FPGA primitives is to facilitate floorplanning. While floorplanning of a design synthesized by the tools is possible, it is extremely cumbersome during development. An update in the HDL code may change the way synthesis is done, causing previous floorplanning efforts to be useless.

Generally, one should only floorplan manually instantiated primitives. This is largely the case in the xi2 processor.

Floorplanning can be done in several ways. Components can be floorplanned relative to other components or be given fixed position in the FPGA. Usually, the first method is the preferred one, specifying that related logic should be packed closely is usually enough.

Routing

Finally, it is possible to route the design manually. This has not been done in the xi2 processor as the benefits for doing so is, in general, not worth the extra effort required [18].

2.6.5 xi2-dsp

Parallel to our project a related project has been undertaken by Kristoffer Hultenius and Daniel Källming. The xi2-dsp project aims to further develop the DSP side of the xi2, providing enhanced performance for signal processing tasks. More specifically the project has provided the xi2 with:

1. 32-bit Division Instruction
2. Low latency 16x16-bit Multiplication
3. Two-way associative instruction cache
4. Two-way associative data cache
5. Improvements to the AGU
6. Interrupts

These features, among many smaller tweaks and improvements has been implemented while maintaining a high clock speed of about 320 MHz. Our project uses the division unit from the xi2-dsp project and some discussion is based on results from their cache implementations.

Chapter 3

The XIPS processor

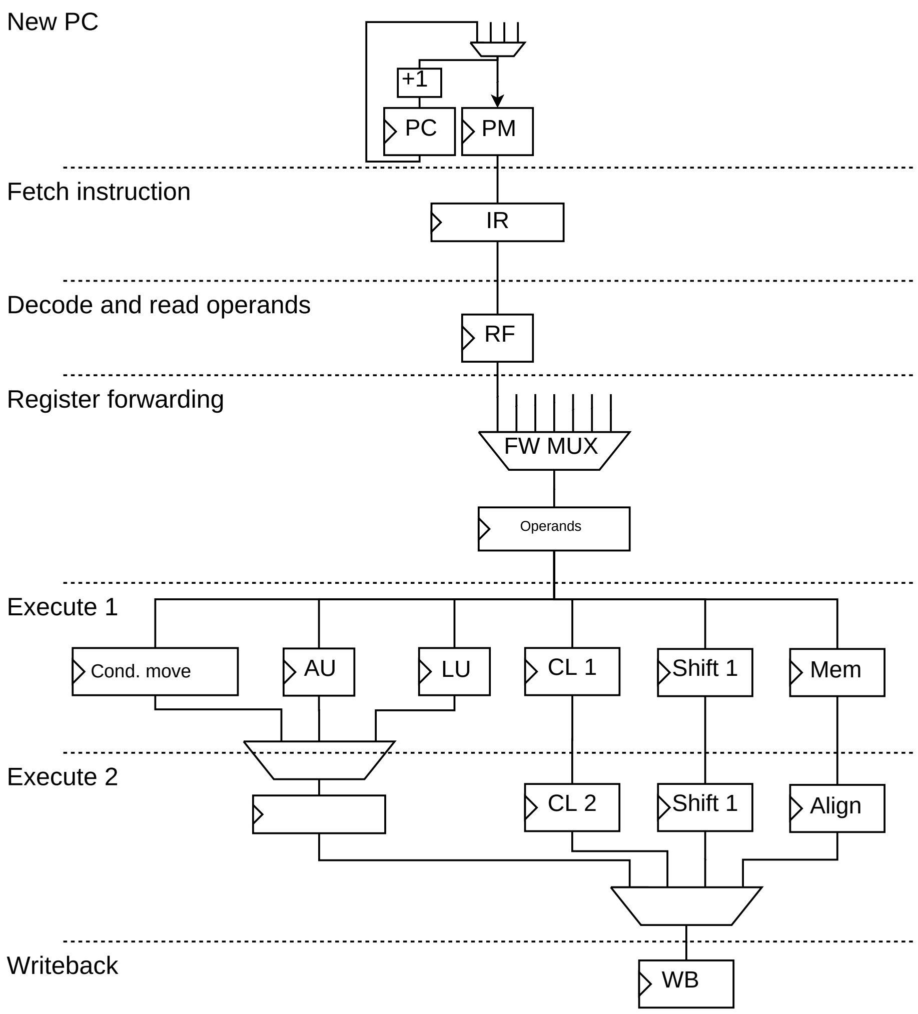
The XIPS processor represents the bulk of work carried out in this thesis. It is basically a MIPS32 version of the xi2 processor sharing the same basic architecture. To accommodate the ISA, a lot of changes has been made. The data path has been extended to support additional instructions and branching and forwarding logic have been revised. Stall functionality has been added to hide the pipeline from the programmer and supporting all instruction sequences allowed by the MIPS32 ISA to be executed correctly.

Only a subset of the ISA has been implemented, but enough work has been done to allow running software compiled by GCC. This allows us to measure performance by running benchmark applications and comparing the results with other processors.

This chapter serves to document both our work and the resulting hardware and software. What follows is an account of the new functionality implemented as well as the effort to maintain as high clock speed as possible.

3.1 Differences from the xi2 processor

While the basic architecture remains the same, there are quite a few important differences between the xi2 and the XIPS processor. Certain features of the xi2 are not used in the XIPS and the corresponding hardware units were removed. This includes the AGUs used for


New PC
+1
PC
PM
Fetch instruction
IR
Decode and read operands
RF
Register forwarding
FW MUX
Operands
Execute 1
Cond. move
AU
LU
CL 1
Shift 1
Mem
Execute 2
CL 2
Shift 1
Align
Writeback
WB

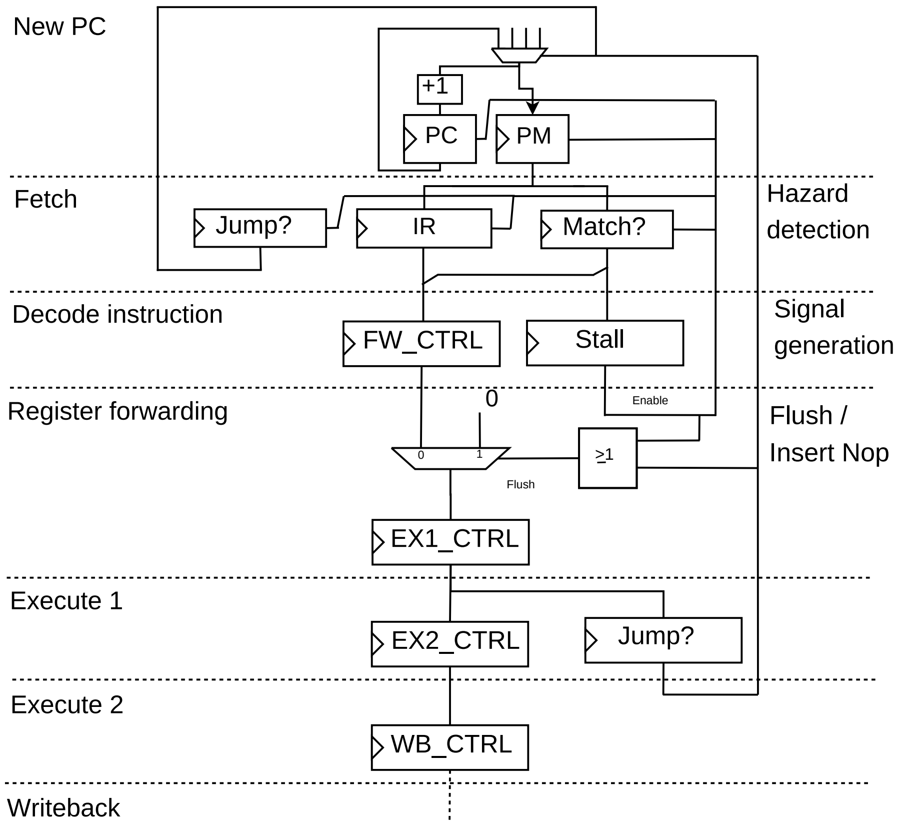


Figure 3.2. Overview of the XIPS control path

easy data access in DSP applications and the constant memory used for 32-bit constants. The important differences and new features are detailed below.

3.1.1 Instruction set architecture

The instruction set of the xi2 is a relatively close match with the MIPS32 ISA, both being fairly standard RISC instruction sets. Though most of the instructions existed in both sets, MIPS32 is a bit more extensive and contains a number of additional instructions. This section will detail some of these new instructions and the hardware required to execute them. For a complete listing of all supported instructions, please refer to Appendix D.

The encoding of instructions are of course totally different so the decoding stage had to be completely rewritten.

Simplified usage of multiply-and-accumulate instructions

The xi2 already had multiply and MAC instruction, but their implementation was not sufficiently hidden from the programmer, requiring the use of a "Finalize" instruction. The MIPS32 ISA does not make use of such an instruction, so the same functionality would have to be achieved otherwise.

One solution would be to do the finalization right before the value is read from the special registers. Unfortunately, due to the pipelined nature of the MAC unit, this would introduce an additional latency of several cycles.

The solution is to allow the MAC unit to detect when it is possible to do a finalize on its own, and perform it accordingly. This causes the second problem. When the result is finalized, the partial sums are destroyed and should the programmer wish to continue accumulation, the result will incorrect. So the hardware must do these things:

1. Detect that it is possible to do finalization and perform it as soon as possible.

2. Restore the partial sum so that further accumulation can be done.

Figure 3.3 shows a revised version of the MAC unit, that supports restoration of the partial sums. The idea is to correct the partial sums through a special input to the DSP48-slices. The finalized result is divided into approriate parital sums and then fed back into the DSP48-slices through this input.

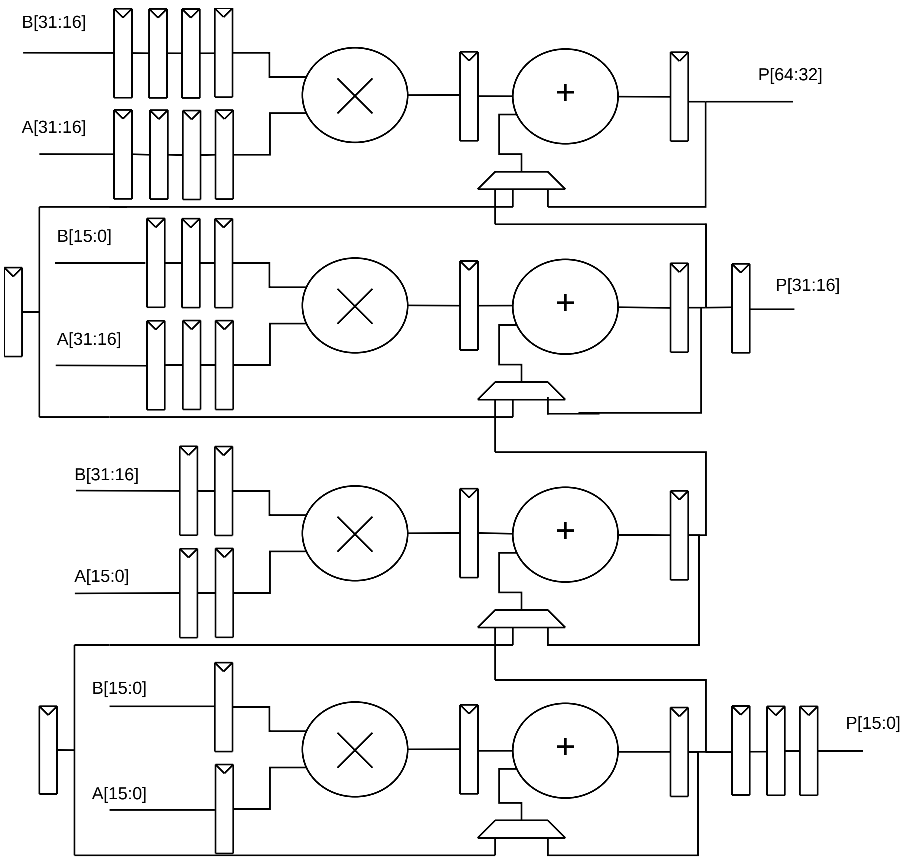


Figure 3.3. Revised version of the MAC-unit. Note the extra inputs to the muxes. These are used to restore the partial sums after finalization. Control logic is not included

Division instruction

MIPS32 has signed and unsigned 32-bit division resulting in a 32-bit quotient and a 32-bit remainder. These results can be accessed via special instructions in the same manner as the results of a multiplications. To accommodate this, a division unit has been integrated. This division unit was implemented by Daniel Källming as part of the xi2-dsp project. The original version used in xi2-dsp did not support signed division, so a slight adaptation to our needs was necessary.

Other new instructions

A few other less common instructions had to be implemented. Implementation details are beyond the scope of this document, but they are nonetheless worthy of mention.

- Count leading ones/zeroes (CLO, CLZ)
- Set on less than (SLT, SLTI, SLTU, SLTIU)
- Move from/to HI/LO (MFHI, MFLO, MTHI, MTLO)
- Conditional moves (MOVN, MOVZ)
- Various branch/jump instructions (BNE, BEQ, BEZ etc.)

3.1.2 Forwarding and Stall

Requiring the programmer or compiler to handle cases not supported by forwarding (as in the xi2 processor) is not an acceptable solution if MIPS32 code is to be executed. Additional hardware to detect these cases and take the appropriate action is required. The XIPS processor implements stalling of the fetching and decoding stages, allowing execution of issued instructions to continue long enough to produce the results needed. To maintain a high clock frequency, the logic is spread over several pipeline stages. Figure 3.4 illustrates a sequence where required data is not available and a stall is needed. Recall figure 2.5 for a case when forwarding is possible.

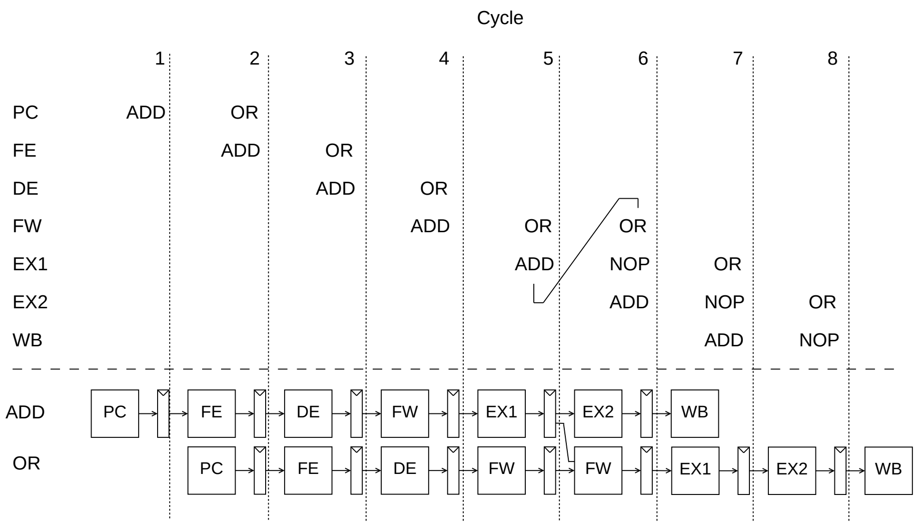


Figure 3.4. The data is required in cycle 5. Since it is not available the processor will stall, keeping the OR in the FW-stage and inserting a NOP in its place. Everything in the top of the pipeline, stage 1 to 4 is naturally stalled as well. In cycle 6 the data is available, the stall is lifted and the data forwarded to the OR instruction as usual.

Data hazard detection

Detecting data hazards basically consists of comparing the source register addresses of the correct instruction with the destination register address of recently issued instructions. In the xi2 as well as in the XIPS processor this matching is done in the DE-stage before we have fully decoded the instruction. This introduces some problems but is necessary for timing reasons.

The most important issue stems from the MIPS32 instruction format. Different instructions indicate destination register address in different fields. Since we do not yet know which instruction we are dealing with we must check for all possible combinations. Additionally, some instructions write to registers using implicit destination addresses (mainly branching instructions) and do not explicitly indicate their destination at all. The relative complexity of the instruction format, combined with a few special cases, leads to increased complexity in properly detecting hazards. Figure 3.2 shows how the detection and the generation fit in the control pipeline.

Forwarding and stall signal generation

The detection stage merely detects possible hazards and does little more than matching certain bits between the current and recently issued instructions. To generate appropriate forwarding and stall signals we must make sure that the hazard is real and a problem, as well as determining the proper cause of action.

Stall handling

When a hazard that can not be resolved by forwarding has been detected, the processor stalls the first four pipeline stages until the data is available. This seems simple enough, but the pipelined implementation of hazard detection and signal generation results in some additional problems.

Hazard detection and signal generation for both operands is done independently and in parallel. This poses a problem when stalling is required for one operand while the other merely requires forwarding. As the stall will insert NOPs in the pipeline, the forwarding decision will no longer be valid. Luckily, the stall itself gives us an additional cycle in which to correct this.

Special cases

Apart from data hazards there are a few other situations that require the processor to stall. These special cases are:

- **Multiplication to GPR**: Multiplication to a general purpose register does not fit in the ordinary pipeline. The latency of the multiplication is too high. The processor will stall to handle this.

- **Move from HI or LO**: Multiplication and division results are stored in two special registers: HI and LO. These registers can be read from or written to using special instructions. Multiplication and division does not fit in the ordinary pipeline. If a

MFHI is issued too soon after a MUL, before the result is ready, the processor will stall.

- **Move to HI or LO**: The MAC-unit does not support a MTHI directly followed by a MTLO, as this would introduce a structural hazard. The reason for this is not obvious but stems from how the DSP48 slices are built. Basically two adjacent DSP48-slices share an input. This input is required in the same cycle by both slices and in this case the processor will stall for one cycle to avoid the structural hazard.

- **Conditional Move instructions, MOVZ and MOVN**: These instructions require the WB-stage to be optional. If the condition is not met the instruction does nothing. This poses a problem for forwarding to work. The value to be written must exist somewhere in the pipeline. If the condition is not met, the old value of the destination register should be forwarded, but that value is not available in the pipeline since only the source registers are read during the DE-stage. One possible solution to this would be to add an additional read-port to the register file. However the performance gain of doing this is probably minimal. Instead, the processor will stall until the data is written.

Loopback

As discussed in the previous chapter, the xi2 uses loopback within the AU and LU. This allows, for example, an ADD instruction to use the result of an earlier ADD instruction even if there is no instruction in between, excluding forwarding. The loopback feature of xi2 is turned off in the XIPS processor. Early estimates concluded that it would be difficult to maintain high clock speed with loopback in conjunction with the more complex forwarding and stall generation. Chapter 4 will discuss the implications in terms of performance and area of this decision.

3.1.3 Branch handling

While xi2 used flags for jump decisions, MIPS does not make use of flags. Instead, registers are checked for certain conditions. When a jump instruction is issued, a jump prediction is made. This prediction can, of course, be incorrect, and has therefore to be reverted if the real decision was different from the prediction. Instructions on the incorrect path are now in the pipeline, and have to be flushed. To reduce the penalty of an incorrect prediction, MIPS makes use of one delay slot. As specified in the MIPS standard, the program counter is 32 bits wide, allowing a program data space of 4 GByte. The complexity of the PC computation increases as new types of jump instructions are added to the processor. The address incrementation, combined with a big mux showed to be time critical, and was tweaked and optimized to meet the timing requirements. For every instruction fetched, the input of the program memory is to be one of several choices. For example, the next instruction can be a prediction of a conditional jump, an offset, the value of a register, or an absolute value.

The advantage of using flags (like xi2 did) is that the branch decision hardware gets simpler, since the branches just have to test for a certain flag, and not for a whole condition. Also, the flags are set by the instruction before the jump, so the hardware has more time to calculate the decision. On the other hand, the benefit of using branches that do their own calculations is the increased freedom of choice for the assembly language programmer or compiler. The branches do no longer depend on the previous instruction.

3.1.4 Known issues

MAC instruction after a division instruction

Both ordinary multiplication, multiply-and-accumulate and division write to the HI and LO registers. MAC will add the result of the multiplication to what is already in the HI and LO registers. If one was to execute a division instruction followed by a MAC-instruction, one would expect the result of the multiplication to be added to the result of the division. However, due to implementation details this is not possible and will not work as expected.

This is only a minor issue since the sequence resulting in the error does not really do anything meaningful. Adding the 64-bit result of a multiplication to the 32-bit remainder and 32-bit quotient result of a division makes little sense.

Program counter

Since the plan is to extend the processor with a cache memory, the program counter has some flaws that will have effect if a larger memory than addressable with 16 bits is used. See section 3.2.2 about the absolute jump address handling.

Incomplete instruction support

XIPS supports a subset of the complete MIPS32 standard [5]. Certain groups of instructions have been excluded, mostly because they require functions not supported by the current hardware, but also due to the limited time of the project.

Excluded parts are:

- Floating point instructions
- Cache related instructions
- Coprocessor instructions
- Obsolete branch instructions (a.k.a. "branch likely" instructions)
- Exception related instructions (SYSCALL, BREAK and trap instructions)
- The instructions LWL, LWR, SWL and SWR

Giving the appropriate settings to the compiler, emission of most of these instructions can be avoided, but some of them must however be added to be able to claim that the processor really supports the

MIPS32 standard. The concerned instructions are SYSCALL, BREAK, and trap functions. The last ones (LWL etc.) can probably be implemented with exceptions executing a small piece of code that emulates the instruction. Floating point, cache, and coprocessor instructions are not meaningful, since the corresponding hardware is lacking. They should not be generated by the compiler if not explicitly used.

3.2 Performance optimization

It has been our ambition to maintain as high clock speed as possible, and this section will cover some of the ways we have tried to achieve this.

3.2.1 Work flow

Figure 3.5 is intended to give an idea of how the work flow of developing a socware design may look like. Some parts require further explanation.

The task of determining whether the timing problems are solvable or not includes analyzing the critical paths in the current synthetization report. Furthermore one can examine what kinds of primitives that have been utilized by the synthesizer. Sometimes one needs to write more explicit code, for the tools to be able to synthesize in the desired way.

3.2.2 Examples in the XIPS processor

Retiming of program counter increment

To reduce the critical path in the program counter (figure 3.6), the high bits of the next-PC register were moved to before the adder. As we can see in figure 3.7, the longest logical path is changed from a 32-bit adder and a mux, to two separate logical paths, whereof none is longer than a 16-bit adder and a mux. Noteworthy about the retiming technique is that the outer function of the retimed circuit

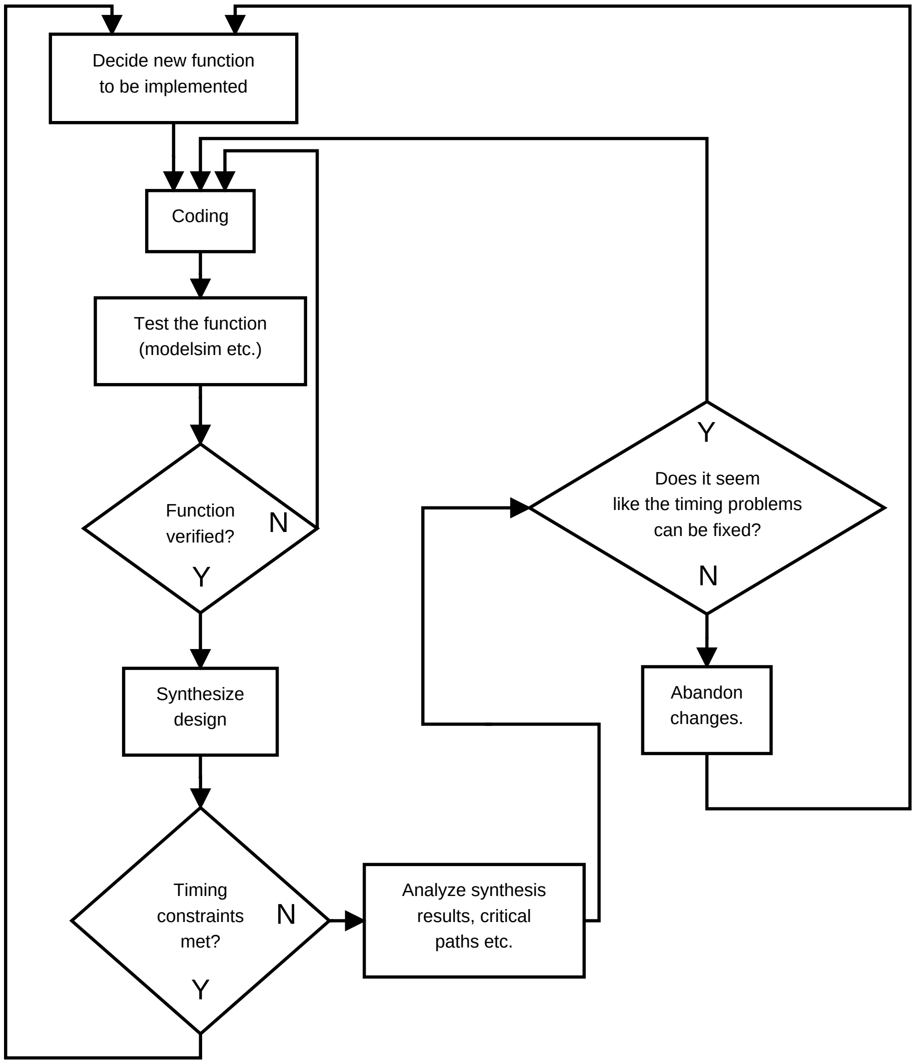

Decide new function to be implemented
Coding
Test the function (modelsim etc.)
Function verified?
N
Y
Synthesize design
Timing constraints met?
N
Y
Analyze synthesis results, critical paths etc.
Does it seem like the timing problems can be fixed?
Y
N
Abandon changes.

Figure 3.5. Flow chart describing general work flow

is not affected, so one can apply it without worries, as long as the retiming rules are followed. [8]

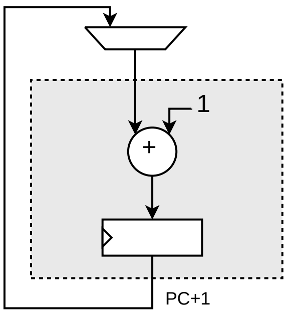


Figure 3.6. Original program counter

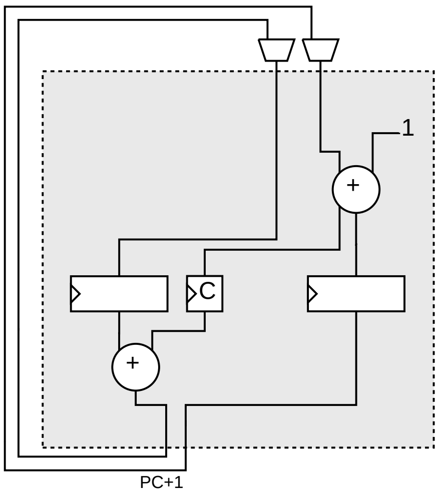


Figure 3.7. Program counter after retiming

Absolute jump addresses

Some of the jump instructions are relative, which means that the processor adds the jump offset to the current program counter, but since the current program counter is simply the address of the instruction, the jump destination can be known in advance. There is reason to exploit this, because an addition will then be saved in the design. So the toolchain (see section 3.3) recodes this kind of jump instructions, so that they contain the lowest 16 bits of the absolute jump destination, and a carry bit that is to be added to the upper 16 bits. In the future, the plan is that the cache memory is going to do this recoding, when fetching from the main memory, thus it is going to be completely invisible for the compiler. Now one could say that the 16 lower bits are absolute, and the upper 16 are relative.

With this in mind, notice that because most jumps are short (as in less than 64 kB), they will often not change high bits of the PC. This leads to the possibility of yet another optimization. We skip the addition to correct the high bits, and boldly use the old ones from the PC from the jump instruction. These will be correct in most cases, and when not, they could be corrected further down in the pipeline, where there is ample time for the addition. At present state, this correction is just for future compatibility, and has no effect, because the program memory uses the uncorrected PC, and there is no ability to correct for an instruction that is fetched using wrong upper bits. However, with the current size of the program memory, the memory area is smaller than 16 bits anyway, so function is not affected now. When later introducing a cache memory (see chapter 6 about future work) this problem could possibly be solved by regarding the erroneous PC as a cache miss. Another proposal is to use the functionality of flushing mispredicted jumps, and regard the faulty PC as an incorrect prediction.

To illustrate the work, we will follow the instruction sequence of BNE, ADD, SUB on their way though the first pipeline stages. In this example BNEs prediction bit is `1`. In figure 3.8, registers `P1`-`P3` are the prediction bit pipe. `C`, and `CC` are carry bits.

```
0x400000a0   BNE
0x400000a4   ADD  //delay slot
...
0x40000ccc   SUB
```

1. We start when the BNE is in the FE-stage, so `pc_piped=400000a0` and `P1 = 1`, and since the jump does not jump very far, the

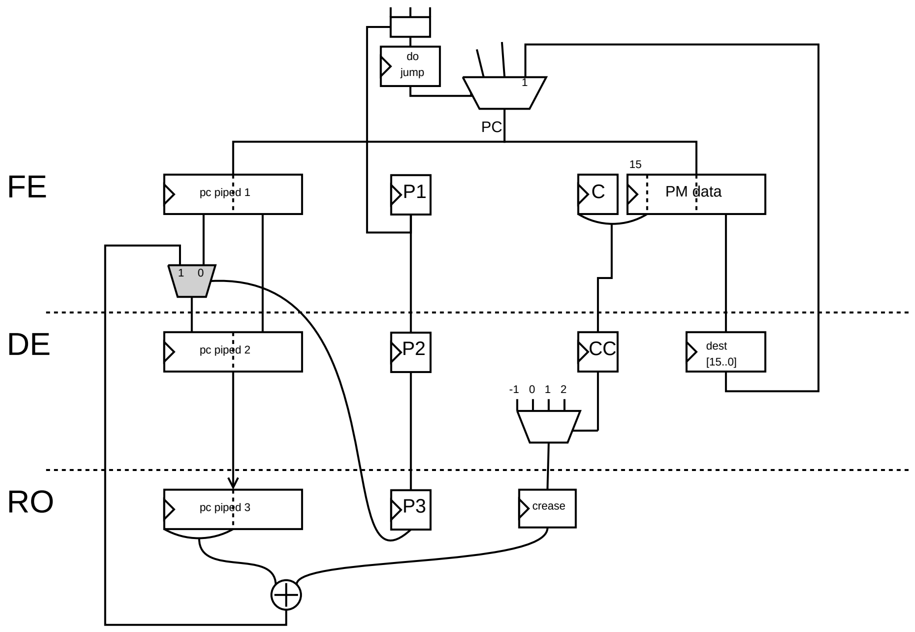


Figure 3.8. Illustration of offset jumps and how they are corrected

carry bits are zero. (C=0, PM_data[15]=0)

```
FE: pc_piped1=400000a0 (BNE), P1=1, C=0  PM_data="BNE"
DE: ...
RO: ...
```

2. Now, the P1 has made do_jump=1, and also P2=1. do_jump makes the PC mux select dest. BNE is now in the DE-stage, so dest=0ccc, and ADD is in the FE-stage.

```
FE: pc_piped1=400000a4 (ADD),          PM_data="ADD"
DE: pc_piped2=400000a0 (BNE), P2=1, CC=00, dest=0ccc
RO: ...
```

3. Now, dest is clocked into pc_piped1 and has also fetched the SUB instruction on PM_data.

```
FE: pc_piped1=40000ccc (SUB),           PM_data="SUB"
DE: pc_piped2=400000a4 (ADD),
RO: pc_piped3=400000a0 (BNE), P3=1, crease=00
```

4. Since P3 was 1 before, the high part of pc_piped3 was selected in the gray mux, so when SUBs address moves from pc_piped1 to pc_piped2, the high bits are corrected (in this case they happen to be corrected to the same value as before: 0x4000)

```
FE: pc_piped1=40000cd0 ...
DE: pc_piped2=40000ccc (SUB),
RO: pc_piped3=400000a4 (ADD)
```

Extra bits in the program memory

The Xilinx block RAMs contain so called parity bits. For every 32 data bits, there are four extra bits, originally intended to be parity bits in vulnerable applications. We, however, use them as well needed space for adding some extra width to the instruction word. Some information that normally would have had to be decoded from the instruction, can now explicitly be put in these bits. The information in the bits only depend on the instruction itself (and its position in the program), and has no runtime dependency, therefore they can be generated at compile time. At the moment, this is done by a Python script (See section 3.3 about the toolchain) but the plan is to calculate the extra bits in the cache memory, when fetching instructions after a cache miss, so neither the user nor the toolchain should be concerned with these bits.

1. The first bit provides information to be used in the forwarding unit. It decides how to interpret the *rd* and *rt* field of the instruction, or more precisely: which of them that points out the destination register. The choice is different for different instructions. Without this extra information, the forwarding unit would have had to decode the instruction completely before being able to forward correctly.

The reason why some instructions (the instructions with immediate operand to be precise) have this special coding is that the *rd* field overlaps with the immediate data field. An alternative solution to this could be to do some changes to the instruction coding. For example changing the order of *rs, rt, rd* to *rd,rt,rs,* so the destination bits do not have to be moved to *rt*.

2. The second bit is used as a carry bit for the previously mentioned PC-relative jumps. When the sixteen lowest bits of the target address are calculated, the result is an addition between the offset (which is sixteen bits) and the lower half of the instruction address (also sixteen bits), giving a seventeen bits result, i.e. one carry bit that has to be stored to be able to calculate the remaining upper bits.

3. The third bit decides whether the instruction is a register jump. It is used in the program counter unit as a selector for the new program counter value.

4. The forth bit is a prediction bit for the conditional jumps. The goal is to set this bit to the most probable jump decision for every specific jump instruction. (see section 3.3)

Comparator

The comparison between two 32-bit numbers showed to be a critical path. Originally, the comparison was implicitly expressed. After instantiating a manual solution, the latency of the comparison could be reduced. The idea is to do a partly parallel, partly serial comparison. Four bits are compared in one LUT, and uses every such part result (i.e. the output from a LUT) as input to a 2-to-1-multiplexer. Then the multiplexers are cascaded, and on the first one, a "1" is inserted. This "1" will now propagate through the multiplexers. If one part result fails, that multiplexer will select a "0" and thus zeroing the final result.

The reason why this method is faster is that it can make use of the carry chain that runs through the slices. The carry chain is "hard

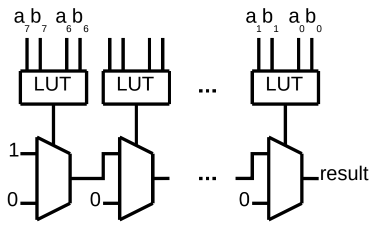


Figure 3.9. Illustration of the propagating comparator

wired", i.e. it is always there, and does not use the routing. Further on, it was improved even more, by splitting the carry chain into four parts, and using them as input to a LUT, to form a trade-off between carry chain delay and routing delay. This is a bit surprising, as the carry chain is very fast. In this particular situation, however, there was a small gain by using a shorter chain and the general routing.

3.3 Toolchain

The process of translating C code into machine code for our processor requires several steps. After the usual compiling and linking, the binary file is split into a program and a data part, to be placed in the respective block RAM. For the data part, this is just a matter of repacking data into a format that can be read by, and thus instantiated with, the Xilinx tools. For the program part however, the program is passed through a Python script, with the task to preprocess the instructions, to reduce the workload for the core at run time.

3.3.1 GCC

The first compiler that was used was gcc-4.2.4. A change to version 4.4.3 actually gave some improvements, just by changing the compiler version. The new compiler was then enhanced with a patch (written by Johan Eilert at ISY), tailor made for our architecture. The purpose of the patch is to describe the architecture, and define costs and benefits for different sequences of instructions. Thereby, the compiler optimization process can do a better job, rearranging the instructions in an efficient way.

3.3.2 Python script

The main part of the preprocessing is to transform the jump instruction addresses that are PC-relative, to absolute addresses. Since the address of the particular jump instruction, and also the offset of the jump, are known, the absolute destination can be calculated. As a result, the need for an offset adder in the PC unit is gone, and the critical path is reduced.

Another important part is the addition of extra bits to the instruction word. As mentioned in section 3.2.2, there are four extra bits per instruction. To set the value of the first bit (the one that decides which one of *rd* and *rt* to use as destination) it just has to decide if the instruction belongs to the group of instructions that uses the *rt* field as destination pointer.

The second bit is actually used together with the MSB of the absolute destination. They are coded to represent any of the numbers $\{-1, 0, 1, 2\} * 2^{17}$, to be added to the remaining bits of the jump address.

The third bit, the register jump bit, is just a check if the instruction happens to be any of the register jumps. The idea is to allow the processor to predict the destination of a register jump using a shadow register. At present time, this does not work optimally and register jumps will always incur a small penalty.

The fourth bit, the prediction bit, tells the program counter to — or not to — predict a conditional jump as taken. The choice of the prediction for a jump is of importance for the performance of the program. The first, and most simple approach is to predict every jump as taken. If our program uses a lot of loops that run for many iterations, this would seem like a good idea. For the same reason, taking *no* jumps, would be a bad idea. This is naturally very program dependent.

Another, slightly more sophisticated rule is to predict backward jumps as taken, and forward jumps as not taken (in this thesis named "smart" prediction). The reason for this is that loop jumps tend to jump backwards, and are taken every time in the loop except the last. And when seeing that the results when using "Always" and "Never" was almost as bad as randomly guessing, it could not get worse by trying to distinguish the loop jumps from other jumps. The fourth bit is also set on a register jump, as it tells the processor to instantly jump somewhere else then the ordinary sequential program flow.

See section 4.4.2 for the results of different branch prediction rules.

3.3.3 Testing and Verification

Functional verification of hardware is a tricky and time consuming problem. It is not uncommon for the verification process to require more time than the actual design and implementation, especially for an ASIC. For FPGAs, however, bugs can be fixed later on and verification does not need to be as strict.

In this thesis, verification has not been our foremost concern, but has been done out of necessity. We did not seek to "prove" functional correctness as such, but merely to arrive at something correct enough to run Coremark. As Coremark is self-verifying, we feel confident that obtained scores are trustworthy, which is enough for our purposes. Bugs can of course still occur, but at this point they are probably of obscure and rare nature, since they have not yet been revealed. Therefore, at least one can regard them as not affecting our conclusions significantly.

Late in the project, a bug occurred when compiling for MIPS I, with gcc-4.4.3, using custom tuning. This bug has not yet been analyzed, mainly due to lack of time, but since all other configurations are running fine, it should be possible to find and fix the error. In the end, the processor is correct enough for benchmarking purposes.

Test programs

Testing of the xi2 involved a series of test programs and a script to run and compare these with known results. We initially kept this basic idea, but wrote new test programs in MIPS32 assembly.

Our test programs generally targeted a specific feature, such as testing new instructions or branching and stalling logic. Initially, we had to deduce the expected results by hand, since no simulator was available. This proved extremely cumbersome, and made writing new test programs tedious and time consuming.

SPIM

SPIM (MIPS backwards) is a MIPS simulator created at the University of Wisconsin as a teaching aide for computer science. As such, it has been in use and development for over 20 years. SPIM provided us with a reference model and greatly simplified verification, as output from the simulated processor could now easily be compared to the output of the simulator.

To obtain SPIM for yourself and for more information, please refer to [14].

Random instruction generator

The SPIM simulator was used as a reference for verification and bug detection. Running a lot of randomized cases on the simulator, and on the core, and identifying differences in the results could help finding dependency cases that was not already taken into account. To be able to run a large amount of random instruction sequences in SPIM and XIPS respectively, a script for generating test programs, simulating them, and comparing the results, were written. The test programs are quite simple. First, a set of registers are filled with random data, followed by a sequence of randomly generated instructions performing arithmetic using these registers. The result of the program is simply the final contents of the register file.

Reference Platform

To allow us to run programs compiled for MIPS32 without having to use a simulator, and as an ASIC reference, we used a wireless router called WRT54GL. It is interesting not only because it contains an ASIC MIPS32 core but because it runs Linux. This made it easy to compile and run Coremark on it. When Coremark was ported, we did not make use of any standard C library. Instead the system was called directly. It was possible because Coremark does just use some text output, and time functions. As an extra benefit, the code will then look more like the XIPS version. The result chapter contains a performance comparison between the XIPS processor and the processor of the WRT54GL

For more information on the processor in the WRT54GL, please refer to [15].

Chapter 4

Results

In this chapter we present our results. We discuss the performance numbers obtained and compare them with those of the competition. Appendix B provides performance charts for a variety of processors, while the text below includes a closer look at the competition. We also try to do some estimations on the performance impact of certain interesting improvements or changes, to see if there is room for improvement.

4.1 Synthesis Results

The XIPS processor was synthesized with speed optimization. Many factors will influence the maximum possible frequency. Below we present the assumptions made during synthesis, and the maximum frequency obtained.

Synthesis tool: Xilinx ISE 10.1 (xst)

FPGA: Virtex-4 Speedgrade 12 (Part xc4vlx80-12-ff1148)

Temperature: 85°C (default: 0-85°C)

Supply voltage: 1.140 (default: 1.140-1.260)

Clock jitter: None

Maximum frequency: 285 MHz

Optimizing a large design is not an easy problem. The synthesis tools use heuristics to be able to complete the task in a reasonable amount of time. Because of this, the synthesis results can sometimes seem a bit random when timing constraints are close to the lowest possible. When the tools fail to meet the constraints, they usually fail dramatically and better results can often be obtained by increasing the allowed delay a bit. To make sure we got the fastest design possible, the processor was synthesized several times with small variations in timing constraints.

4.2 Running Coremark

By its nature, Coremark is intended to be portable to new processor architectures. The main work of the porting was to provide Coremark with output and input functions, and rebuilding the memory modules to make Coremark fit. Rebuilding was needed because of the lack of a cache memory, so the whole program had to fit into block RAMs. Parts of the toolchain had to be special made to split the program into multiple block rams. The linker script and startup assembly files were also changed to get the stack and data allocation right. Some minor configuration regarding data types and the lack of floating point operations were also necessary. For the text output, a UART transmitter from Opencores, connected to a terminal program on the workstation was attached to the memory bus.

To measure its performance, Coremark needs functions to get the current time. This was done using a memory mapped clock cycle counter. To get some extra information, more counters for interesting signals, such as stall, were added and reported with the Coremark results.

The Coremark tests were run on our development platform (see section 2.1.1). To save work, a frequency of 100 MHz was used to run the tests. The main reason for this was the extended size of the program memory (see above). Another reason was that the version of the Virtex-4 on the development board had speedgrade 10, and not 12 as targeted in the project. Furthermore, it would have been a waste of time to optimize the UART controller and other peripherals, since they would not have been needed in the final solution anyway. Finally, an easy availible system clock at 100 MHz already existed on the board. The coremark results can be translated to what they would have been if run at 285 MHz.

Worth to mention is that we actually — against the rules for Coremark reports — made some changes in the *core_main.c* file. The changes are some definitions of memory mapped counters and control registers, a few debug outputs, and output for the additional measurements that we added. However, no changes were made to the timed parts, nor to the calculation of the result.

Another exclusion clause is that some of the Coremark scores in the tables, are not directly taken from the "Iterations/Sec" output line. The number has been manually calculated from "Iterations" and "Total ticks". The reason for this is that on processors not supporting floating point operations, the "Total time (secs)" is an integer (a divided and truncated version of "Total ticks"). This means that the "Total time (secs)" can become rounded downwards with almost one second, resulting in a too large value of "Iterations/Sec". These manual calculations has been done for XIPS and WRT54GL. The other scores come from external sources.

4.3 Performance and area

When the coremark results were measured, different compiler options were used. Table 4.1 shows the results, expressed in coremark score (iterations per second). Shown is the difference between different compiler versions, different versions of the instruction set, as well as the impact of the architectural description patch (the XIPS columns). The improvement of changing to the newer compiler showed to be around ten percent. The customized tuning improved another four percent. Noteworthy, but not a part of the table, is that when using the tuning intended for a superscalar MIPS32 architecture, we got a result of 143.66 iterations per second on the new compiler, compared to 141.21 using no tuning.

	gcc-4.2.4		gcc-4.4.3	
	No tune	XIPS	No tune	XIPS
MIPS I	126,62	NA	139,88	freezes[1]
MIPS32	129,62	NA	141,21	147,27

Table 4.1. Performance comparison using different compilers and settings

	LUTs	Flip-flops
Total	3275	2046
Decoder	578	461
Shift unit	351	130
Forwarding unit	279	65
Arithmetic unit	32	32
Logic unit	101	32
Register file	446	74
MAC unit	406	435
Divider	366	298

Table 4.2. Resource usage for the main parts of the processor

4.3.1 Resource usage

The usage of specific parts of the processor is presented in table 4.2. Apart from LUTs and flip-flops, the processor also uses four DSP48 blocks, in the MAC unit. The register file uses 256 LUTs as distributed RAM, and the program- and data memory use one block RAM each.

4.3.2 Comparisons with other soft core processors

Since the XIPS processor is optimized for FPGAs, comparison with other soft core processors synthesized to FPGAs is of most interest. Unfortunately, we have not been able to run the benchmarks ourselves in all cases and have had to rely on external sources.

[1]When using the custom tuning together with MIPS I standard, Coremark freezes. This problem is probably due to a bug in the processor, and not in the compiler, but unfortunately there was not enough time to analyze this bug.

	XIPS	Microblaze 7.20	Leon4	OR1200
Score	147 @100MHz	93 @62.5MHz	N/A	33 @25MHz
Score/MHz	1.47	1.658	2.1	1.32
Theor. max	419 @285MHz	332 @200MHz	263 @125Mhz	132 @100Mhz
Area (LUTs)	3164	1000 [22]	4000 [10]	5000 [21]

Table 4.3. Comparison of Coremark score with other soft core processors

The comparison is complicated by differences in compiler version, cache usage (all processors, except XIPS use some kind of cache memory) and clock speeds differing from their theoretical maximums. For these reasons we advise the reader to regard these values with a grain of salt.

We have done our best to fully disclose the source of data, as well as any assumptions made in the comparison.

XIPS

- **Compiler Version** gcc-4.4.3
- **Compiler flags** `-nostdlib -mips32 -g -EL -O2 -fomit-frame-pointer -Dxi2 -mtune=xi2mips`
- **Coremark run details** Virtex 4 FPGA. Program and data in block RAMs. No program or data cache. 100 MHz.
- **Motivation for theoretical max** The processor alone can be synthesized to 285 MHz on a Virtex-4 speed grade 12. The comparison figure is simply $147 * 2.85 = 419$. Using cache memories would probably lower this number somewhat, refer to Section 4.5.1 for a more in-depth discussion on this.

Microblaze

- **Compiler Version** gcc-4.1.1 (Xilinx Microblaze)
- **Compiler flags** `-O3`

- **Coremark run details** Xilinx Microblaze v7.20.d in Spartan XC3S700A FPGA, 5-stage pipeline, 4 kB instruction cache, 4 kB data cache, integer divider, barrel shifter. 62.5 MHz. Data is from [9]

- **Motivation for theoretical max** Microblaze, optimized for speed, can be synthesized to 235 MHz in Virtex-5 [1], however Virtex-5 is a more capable FPGA than Virtex-4. To make the comparison a bit more fair we have used the maximum frequency in Virtex-4 instead, which is 200 MHz [2]. Using this figure we get a comparison number as $1.658 * 200 = 331.6$.

- **Area comments** Area is for a Virtex-5 FPGA, which uses 6 input LUTs. It would probably require substanially more in 4 input technology.

LEON4

- **Compiler Version** N/A

- **Compiler flags** N/A

- **Coremark run details** N/A

- **Motivation for theoretical max** We have not been able to run Coremark ourselves. Gaisler Research reports a Coremark/MHz score of 2.1 and a maximum frequency in a Virtex-5 FPGA of 125 MHz [10]. No other data is available. The theoretical maximum is thus $2.1 * 125 = 265.5$. Noteworthy is that clock speed should be slower in Virtex-4.

- **Area comments** Area is for a Virtex-5 FPGA, which uses 6 input LUTs. It would probably require substanially more in 4 input technology.

OR1200

- **Compiler Version** gcc-4.2.2

- **Compiler flags** `-O2 -g -fomit-frame-pointer -mhard-mul -Msoft-div -msoft-float -nostdlib -g -nostdlib -O2 -g -fomit-frame-pointer -mhard-mul -msoft-div -msoft-float`
- **Coremark run details** Coremark was run on the system used in the course TSEA44 at LiTH [17].
- **Motivation for theoretical max** The OR1200 can be synthesized to about 100 MHz [18]. $1.32 * 100 = 132$

4.3.3 Comparisons with ASIC MIPS32 cores

To compare with some ASIC cores, we ported Coremark to a hard MIPS32 processor, located in a network router called WRT54GL. Results are shown it table 4.4. It also includes Coremark scores [9], for a 80 MHz PIC microcontroller from Microchip, running MIPS32 Release 2 instruction set. The comparison is maybe a bit unfair, since it compares FPGAs and ASICs. The intention is however to give a perspective on the differences in soft versus hard processor cores.

The score for WRT54GL is obtained with Coremark running under Linux, at priority level -19. It will impair the score a bit, but probably negligible. The reasons behind the low number of Coremark/MHz are unclear.

4.4 Statistics

As mentioned in section 4.2, some extra output was added to the Coremark program to collect information about stalls etc. The text below shows a sample of what the output looks like. The extra output added by us is marked with `(*)`.

	XIPS	WRT54GL	PIC32MX360F512L
Coremark score	147 @ 100 MHz	342 @ 200 MHz	163.234 @ 80 MHz
Coremark/MHz	1.47	1.71	2.040

Table 4.4. Comparison of Coremark score with ASIC MIPS32 cores

```
Running Coremark for XIPS.
Timed test starts now...
And ended now.
(*)stalls:  500145211
(*)repents: 51480355
(*)jumps:   171675266
2K performance run parameters for coremark.
CoreMark Size     : 666
Total ticks       : 1542929350
Total time (secs): 15
Iterations/Sec    : 133
Iterations        : 2000
Compiler version : GCC4.2.4
Compiler flags    : -nostdlib -mips32 -g -EL -O2
            -fomit-frame-pointer -Dxi2 -DPERFORMANCE_RUN=1
Memory location  : STACK
[0]crclist        : 0xe714
[0]crcmatrix      : 0x1fd7
[0]crcstate       : 0x8e3a
[0]crcfinal       : 0x4983
Correct operation validated. See readme.txt for run and
reporting rules.
```

In this output one can see the total number of clock cycles for the run (named "Total ticks"). One notices that the number of clock cycles in which the stall signal was high (called "stalls") is around one third of the total amount. The "repents" tells how many conditional jumps that was mispredicted and thus had to be flushed. One repent means that four cycles are wasted.

4.4.1 Stalling

To get more detailed knowledge of what causes stalls, a couple of simulations were run with exhaustive debug output. The simulations ran 110 Coremark iterations. For every stall, data was stored containing information about what kind of dependencies that caused it. To get some statistics, a script categorized the instructions and

	au	cl	lu	slt	geo	sh	br	ld	st	mhl
to au	36300	0	146430	0	320760	683760	0	658109	0	0
to cl	0	0	0	0	0	0	0	0	0	0
to lu	747843	0	824959	0	0	629967	0	99612	0	0
to slt	439230	0	431200	0	0	0	0	44	0	0
to geo	0	0	320760	0	0	0	0	356400	0	0
to sh	34100	0	25982	0	320760	178420	0	0	0	0
to br	130350	0	1033340	717882	0	22440	0	3222670	0	0
to ld	2629468	0	3	0	0	0	0	773520	0	0
to st	378230	0	56804	440	35640	0	0	5062	0	39600
to mhl	0	0	0	0	0	0	0	0	0	0

Table 4.5. Dependencies causing stall, using no tuning

	No tune	Custom for XIPS
Clock cycles	**77,901,181**	**74,691,642**
Mispredicted jumps	2,141,362	2,100,882
Stalls	15,279,913	12,879,293
Stall clock cycles	26,147,198	22,379,960

Table 4.6. Overall difference in stall amount, with and without custom tuning

summarized it. Table 4.5 shows how the different groups of instructions[1] depends on each other (in the "hazardous" cases).

To investigate how the custom patch affected the results, simulations were also run with the patch disabled. Table 4.6 shows a summary.

Corresponding detailed statistics for the hazards is seen in table 4.7. To make it more clear, a third table (4.8) is provided, showing the numerical difference between 4.5 and 4.7. A positive number in 4.8 means that there were less stalls for that type of dependency after applying the patch.

[1]"au" = arithmetic instructions, "cl" = count leading ones/zeros, "lu" = logic instructions, "slt" = "set on less than" instructions, "geo" = "geometric" unit (MUL, DIV, etc.), "sh" = shift instructions, "br" = branches, "ld" = load, "st" = store, "mhl" = move from/to low/high

	au	cl	lu	slt	geo	sh	br	ld	st	mhl
to au	108020	0	21670	0	320760	683760	0	635990	0	0
to cl	0	0	0	0	0	0	0	0	0	0
to lu	749163	0	805137	0	0	633047	0	74734	0	0
to slt	34870	0	431200	0	0	0	0	44	0	0
to geo	0	0	320760	0	0	0	0	0	0	0
to sh	35090	0	3080	0	0	33660	0	0	0	0
to br	126830	0	1030260	793562	0	22440	0	2549470	0	0
to ld	1927118	0	3	0	0	0	0	773520	0	0
to st	288029	0	36984	0	35640	0	0	5282	0	39600
to mhl	0	0	0	0	0	0	0	0	0	0

Table 4.7. Dependencies causing stall, using custom XIPS tune

	au	cl	lu	slt	geo	sh	br	ld	st	mhl
to au	-71720	0	124760	0	0	0	0	22119	0	0
to cl	0	0	0	0	0	0	0	0	0	0
to lu	-1320	0	19822	0	0	-3080	0	24878	0	0
to slt	404360	0	0	0	0	0	0	0	0	0
to geo	0	0	0	0	0	0	0	356400	0	0
to sh	-990	0	22902	0	320760	144760	0	0	0	0
to br	3520	0	3080	-75680	0	0	0	673200	0	0
to ld	702350	0	0	0	0	0	0	0	0	0
to st	90201	0	19820	440	0	0	0	-220	0	0
to mhl	0	0	0	0	0	0	0	0	0	0

Table 4.8. The difference in dependencies causing stall

	Always	Never	Random	Smart	Dynamic
Clock cycles	1565M	1575M	1562M	1416M	1396M
Mispredicted jumps	79.5M	86.5M	84.1M	38.9M	35.6M
Mispredicted jumps (relative number)	47.75%	52.25%	50.75%	20.95%	18.75%

Table 4.9. Different static prediction schemes, as well as 1-bit dynamic for comparison.

4.4.2 Branch prediction

At present, the processor only supports static branch prediction, that is the prediction is hard coded in the extra bits, and thus a particular jump will predict the same every time. Some different prediction policies were tested:

Always All prediction bits are 1.

Never All prediction bits are 0.

Random Each prediction bit is given a random (but still constant) value.

Smart Jumps that jump forward in the program are predicted as not taken, and those who jump backwards as always taken.

Table 4.9 is a summary of the outcome of the different prediction schedules. In all cases the number of executed jump instructions is 151,836,381.

The last column, “Dynamic” presents the results of tests when using dynamic branch prediction. Dynamic prediction is not a part of the final implementation, but was implemented as a test to get some statistics of how much better the performance would be if dynamic prediction was used. As we see, the gain of changing from smart to dynamic prediction is a reduction in mispredicted jumps by about ten percent.

	Direct	Two way
Hits	73661720	74672451
Misses	972271	91824
Cache effect	103%	1.9%

Table 4.10. Effects of instruction cache

	Direct	Two way
Hits	9417601	9417599
Misses	79	81
Cache effect	0%	0%

Table 4.11. Effects of data cache

4.5 Estimations

4.5.1 Cache memories

If Coremark is to be run at 285 MHz, cache memories is a required. Is it reasonable to linearly extrapolate performance at 285 MHz with cache memories from performance at 100 MHz without? In general this might not be the case. Cache misses have a real impact on performance. In the Coremark case, however, things may be different.

Cache misses

To research the impact of a cache memory, a dump of all program memory and data memory accesses during 110 Coremark iterations was made. This data was then fed into a simple cache simulator modeling different types of cache memories. The size of the modeled cache memory is two block RAMs, containing 512 instructions each, i.e. 4 kB of cache. The line size is 8 instructions. The only thing that is changed in the comparison between direct mapped and two-way, is the associativity. The size is preserved.

The total number of clock cycles with no cache was 74691642. The estimated penalty for a cache miss is 75 cycles (see below).

The results are shown in table 4.10 and 4.11. An estimation on the effects of the cache memories on the Coremark score, is to use two-way cache memories for the instruction cache, and direct mapped for the data cache. The effect will be that the number of cycles increases with 1.9%, as seen in table 4.10. Taking 1.9% from the Coremark score 419 (as used for comparison in table 4.3, gives a score of 411 instead, this is of course under the assumption that the introduction of cache memories gives no decrease in clock speed.

The "Cache effect" is calculated using the formula

$$\frac{misses * penalty\_per\_miss}{clock\_cycles\_without\_cache}$$

It will tell by how many cycles the running time would increase, if a cache of the type in question was used.

The number of penalty cycles per cache miss can differ a lot depending on the access time of the memory. So in this table we use numbers from the DDR ram on the development board. The access time to the memory is 26 (!) cycles in the DDR controllers clock domain, which means $26 * 2.85 = 75$ cycles in the processors clock domain (assuming that the processor still runs at 285 MHz). So an estimated value is 75 penalty cycles per cache miss, assuming that the address that caused the cache miss is fetched first.

Clock Frequency

The xi2-dsp project has provided xi2 with instruction and data caches. These cache memories can be configured to work as either direct mapped or two-way associative. Reading from a cache memory — especially a two-way associative one — is a bit more complex than just reading from a memory. If the readout from the memory is part of the critical path, then cache memories could negatively affect clock speed.

In the XIPS processor, the program memory is not part of the critical path. We therefore believe a direct mapped cache memory would be possible without too great effect on the clock speed. However, reading from the data memory is a bit more critical and the additional delay of a cache would not help.

4.5.2 Full forwarding

To achieve full forwarding, the forwarding unit has to provide the results from the different stages at the time they are ready. This is not the case at present. Instead the forwarded data goes through registers, or put in other words: the forwarding unit uses its own pipeline stage. The benefit of, and reason behind having a forwarding step in the pipeline is of course to reduce the timing delay through the forwarding. The drawback is on the other hand that all data hazards will demand an extra clock cycle to wait for the result to pass the forwarding flip-flops. At present, if an instruction depends on the result of the closest previous one, it will introduce a stall of one cycle. So by removing the pipeline stage, and thus introducing full forwarding, every stall period would become one cycle shorter. To get an estimation of how many cycles that could be saved by this, we ran a shorter simulation of Coremark, collecting statistics of the lengths of the stalls. The result was (with the custom tuning patch) 22,379,960 stall cycles with the present design, and 9,861,028 stall cycles if every stall period was shortened with one cycle — a difference of 12,518,932 cycles. The total number of clock cycles was 74,691,642, so the percentual gain is 12,518,932 / 74,691,642 = 16.8%.

As an estimation of how full forwarding would affect the clock speed, a synthesis without the forwarding registers was done (just as a test, of course resulting in a non-functional design). With that change, a clock frequency of 219 MHz was met. Compared to 286 MHz giving a percentage decrease of about 23%. The conclusion is that full forwarding is worth investigating. Especially if the processor is to be used in a design that will run at a lower frequency, since full forwarding will decrease the maximum frequency, but increase the number of instructions per megahertz.

4.5.3 Loopback

An alternative to removing the forward stage is to introduce so called loopback functionality. Loopback, i.e. a path from e.g. AU to itself, so that two consecutive arithmetic instructions does not cause a stall. One can also make a “loopback” from one unit to another, if it is likely that e.g. LU depends on AU or similar.

Studying table 4.7, one notices that implementing loopback for e.g. LD → BR (which is the largest number in the table) would reduce the number of clock cycles with about three percent (2,549,470 against 74,691,642), i.e. a very small improvement for quite a lot of hardware.

4.5.4 Critical paths

The critical path is the longest path between two registers in the design, and dictates the maximum clock frequency of the processor. To increase the clock speed of the processor, it is necessary to decrease the delay of the critical path. While the synthesis tool is deterministic — the same input will always yield the same output — small changes in the code or the target timing requirement may change which path is actually the longest one. To try to get a good overview of the paths that are the most problematic, we synthesized the design several times with small variations in target frequency. Two paths came up a lot:

The branch decision Deciding if a branch should be taken or not is a bit troublesome. This was easier in xi2, the processor just had to check the correct flag. The XIPS processor must perform at least a 32-bit comparison. Further investigation could decide if this can be optimized further, and this path may be a good candidate for manual floorplanning.

Reading from the data memory The problem here is really the amount of computation required to read from the data memory. First, the XIPS uses a 16-bit offset that must be added with a general register to form the address. Secondly, the data must be read from the memory and then masked and shifted to accommodate byte and half-word loads. Besides, when a cache memory is to be inserted, there will be even more logic for memory reading. Decreasing the critical delay here seems tricky unless an extra pipeline stage is introduced. Another possibility would be to use an instruction set using a simpler load operation with less options.

4.5.5 More advanced branch prediction

To get an estimation of how much dynamic branch prediction would improve the performance, we added some functionality to write the prediction bit back to the program memory, if a branch prediction would turn out to be wrong. As can be seen in table 4.9, the gain from changing from smart static prediction to dynamic is just beneath ten percents. Not much in the context, but the hardware for implementing it is fairly small, and when using a cache memory, writing in the program cache is needed anyway, so the hardware cost should be small as well. Technically it will mean to check for a mispredicted jump, and then write the jump decision back to the program (or cache) memory on the correct address, which is already available in the pipeline anyway.

4.5.6 Area reduction estimates

There are still a few unused pieces left from the xi2 that could be removed. Some area could be saved by removing the unused loopback feature (see section 3.1.2) of the arithmetic and logic unit. Additionally, small amounts of area could be saved by only supporting the MIPS I instruction set, this is due to a few instructions that are not present in MIPS I but exists in MIPS32. We have not had time to investigate this fully and it might be worthwhile to do so.

Chapter 5

Conclusions

Overall we consider the project a great success and the XIPS processor has shown impressive performance. Should the work be continued, we are confident that a future version of the XIPS could become a serious alternative to established soft core processors. Especially if Xilinx FPGAs are used.

Additionally, the XIPS processor implements a large enough subset of the MIPS32 ISA to run compiled code, making the processor a lot easier to use than the xi2.

However, the comparison performed in this thesis is not without problems and the lack of cache memories prevents the XIPS from really being useful.

5.1 What we could have done differently

In hindsight, one particular thing stands out as something we could have done differently. At the start of the project, getting the MIPS simulator up and running and establishing a good test structure was not our primary goal. Instead we relied on handwritten test programs for far too long.

This was a huge oversight and ended up costing us a lot of extra work. The simulator and the random instruction generator made finding bugs a lot easier and we should have put in the effort to get it up and running as early as possible.

In the end, we can do nothing but remember the lesson learned: The wrong kind of laziness will cost you in the end and proper testing will save you time.

Chapter 6

Future work

Investigations and further development could be done in several areas:

- Integrate a cache memory, and accordingly modify the program counter
- Exceptions and exception handling
- Complete support for the instruction set (see section 3.1.4 under "Incomplete instruction support")
- As mentioned in section 4.5.2, implementation of full forwarding can be worth investigating.
- To improve speed, one could remove the lingering functionality for loopback, that is still left from the xi2 design.
- MMU for program and data
- Investigate frequency versus functionality per clock cycle
- Investigate the "freeze bug" that occurs when running the custom tuning patch together with MIPS I (see note in 4.3).
- Perform a more thorough area evaluation and comparsion. Neither the xi2 nor the XIPS processor has been optimized for area, and it could be interesting to look into it a bit more.

The first things to start off with from this point would probably be the cache memory and exception handling. Bug fixing is also of high priority. Full forwarding is not a requirement, but if it is concluded that it is feasible, it is a good idea to implement it before other things, as it has to do with the core itself. The same goes for the bug fixing and removing of unneccessary loopback functions. The MMU is important if one wants to run a decent operating system, but is of lower priority, since many things can be done without an MMU, and the implementation of one it is a project itself.

Bibliography

[1] Xilinx "MicroBlaze Soft Processor Core", 2010. [Online] http://www.xilinx.com/tools/microblaze.htm

[2] EETimes "Xilinx raises soft processor clock frequency 25%", 2005. [Online] http://www.eetimes.com

[3] OpenCores, OR1200 project page, 2010. [Online] http://www.opencores.com

[4] Xilinx "Microblaze v7.20 FAQ", 2009. [Online pdf file] http://www.xilinx.com/

[5] MIPS Technologies "MIPS32®Architechture for Programmers Volume II: The MIPS32®Instruction Set", 2009. [Online pdf file, requires free registration] http://www.mips.com/products/architectures/mips32/

[6] MIPS Technologies "Virtex-4 FPGA Data Sheet: DC and Switching Characteristics", 2009. [Online pdf file] http://www.xilinx.com/support/documentation/virtex-4.htm

[7] EEMBC, CoreMark homepage [Online] http://www.coremark.org/

[8] Charles E. Leiserson and James B. Saxe, "Retming Synchronous Circuitry", Algorithmica 6(1), 1991.

[9] EEMBC, CoreMark scores page, 2010. [Online], http://www.coremark.org/benchmark/

[10] Aeroflex Gaisler, “LEON4 32-bit processor core”, 2010.
[Online, downloaded 5:th of March 2010]
http://www.gaisler.com/doc/LEON4_32-bit_processor_core.pdf

[11] ORSoC AB, OpenCores homepage
[Online] http://www.opencores.com/

[12] Dominic Sweetman, “See MIPS Run Linux”,
Morgan Kaufmann 2009. ISBN 13:978-0-12-088421-6

[13] ECL Dhrystone White Paper, 2002. [Online], original source:
http://www.ebenchmarks.com/download/
/ECLDhrystoneWhitePaper.pdf
seems to be down. Downloaded, 7:th of March 2010. Mirror:
http://www.johnloomis.org/NiosII/dhrystone/
/ECLDhrystoneWhitePaper.pdf

[14] SPIM - A MIPS32 Simulator, project homepage
[Online] http://pages.cs.wisc.edu/ larus/spim.html

[15] BCM5352EL SOC MIPS32 chip used in the WRT54GL
[Online] http://www.broadcom.com/products/
/Wireless-LAN/802.11-Wireless-LAN-Solutions/BCM5352EL

[16] Avnet Electronics Marketing “Xilinx®Virtex-4™Evaluation Kit”, 2007 [Online, requires free registration]
http://www.em.avnet.com/

[17] Computer hardware - a computer system on a chip.
TSEA44 Course webpage.
[Online] http://www.da.isy.liu.se/courses/tsea44/

[18] Andreas Ehliar, “Performance Driven FPGA design with an ASIC perspective.”, 2009. [Online]
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-16372

[19] Microchip “PIC32MX3XX/4XX”, 2009
[Online, pdf file] http://www.microchip.com/

[20] David J. Lilja, “Measuring Computer Performance - A practitioner′s guide”, Cambride University Press, 2000. ISBN 0 521 64105 5 hardback

[21] Olle Seger, Per Karlström, Olof Kraigher, “TSEA44: Computer hardware - a system on a chip”, Course slides for TSEA44 at Linköpings Universitet, 2009. [Online] http://www.da.isy.liu.se/courses/tsea44/coursemttrl_09/ /1-Intro.pdf

[22] Xilinx homepage, “MicroBlaze Processor Performance”, originally found at http://www.xilinx.com/products/design_resources/ /proc_central/microblaze_per.htm, which seems to bee down, it can however be found at: http://web.archive.org/web/20080714055954/http://www. .xilinx.com/products/design_resources/proc_central/ /microblaze_per.htm

Appendix A

DDR Controller

Early in the project work, the plans were to make a cache memory, connected to a common wishbone bus. A DDR SDRAM interface was to be applied "on the other side" allowing faster and bigger memory than available on the chip. However, time was not sufficient for a cache memory. Nonetheless, a wishbone memory controller for the development board was built.

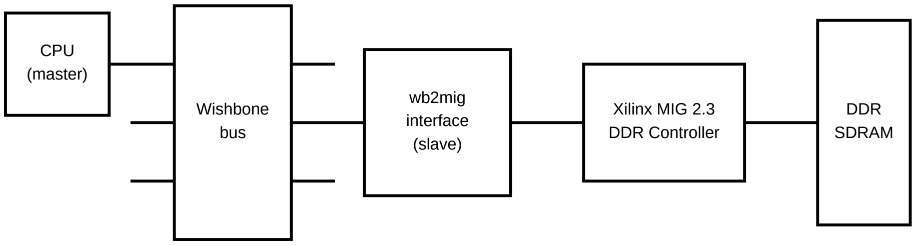


Figure A.1. System overview with memory controller

The controller consists of two main parts: A MC (memory controller) generated by the Xilinx Tool "MIG 2.3", and an interface between that controller and the wishbone bus. The main part of the work lies in the wishbone-to-MC interface ("wb2mig"). The MC has a huge latency of 26 clock cycles (when running the controller at 100 MHz) for fetching data, so at each read, a burst read of maximum length is desired, to put in the cache memory. Wb2mig is basically a state machine that translates and synchronizes the accesses from the wishbone master to the MC.

Short description of the funtion: (where *cycstb* means *wb_cyc & wb_stb*)

A write cycle:

The master requests the write by setting *cycstb* along with the *wb_we* signal. This triggers a write to the MCs address- and data-fifos. The *wb_ack* signal is immediately set, as the write starts directly. It also moves the FSM to the WRITING state, where it normally lasts for up to four clock cycles (if master keeps providing data), before moving back to an idle state, so a new read or write procedure can start. If the master stops providing data, FSM moves to wait in the TERMINATE WRITE state, since the MC side expects four cycles to pass.

A read cycle:

The master requests the read by rising *cycstb*. This triggers a write to the MCs address-fifo. Since the MC cannot immediately provide data, the FSM moves to a wait state. Here it remains waiting for the MC to assert its "data valid" signal. When that happens, the master gets responded with *wb_ack*, and the FSM moves on to the READING state. If master stops reading prematurely, a similar procedure as in the write cycle takes place.

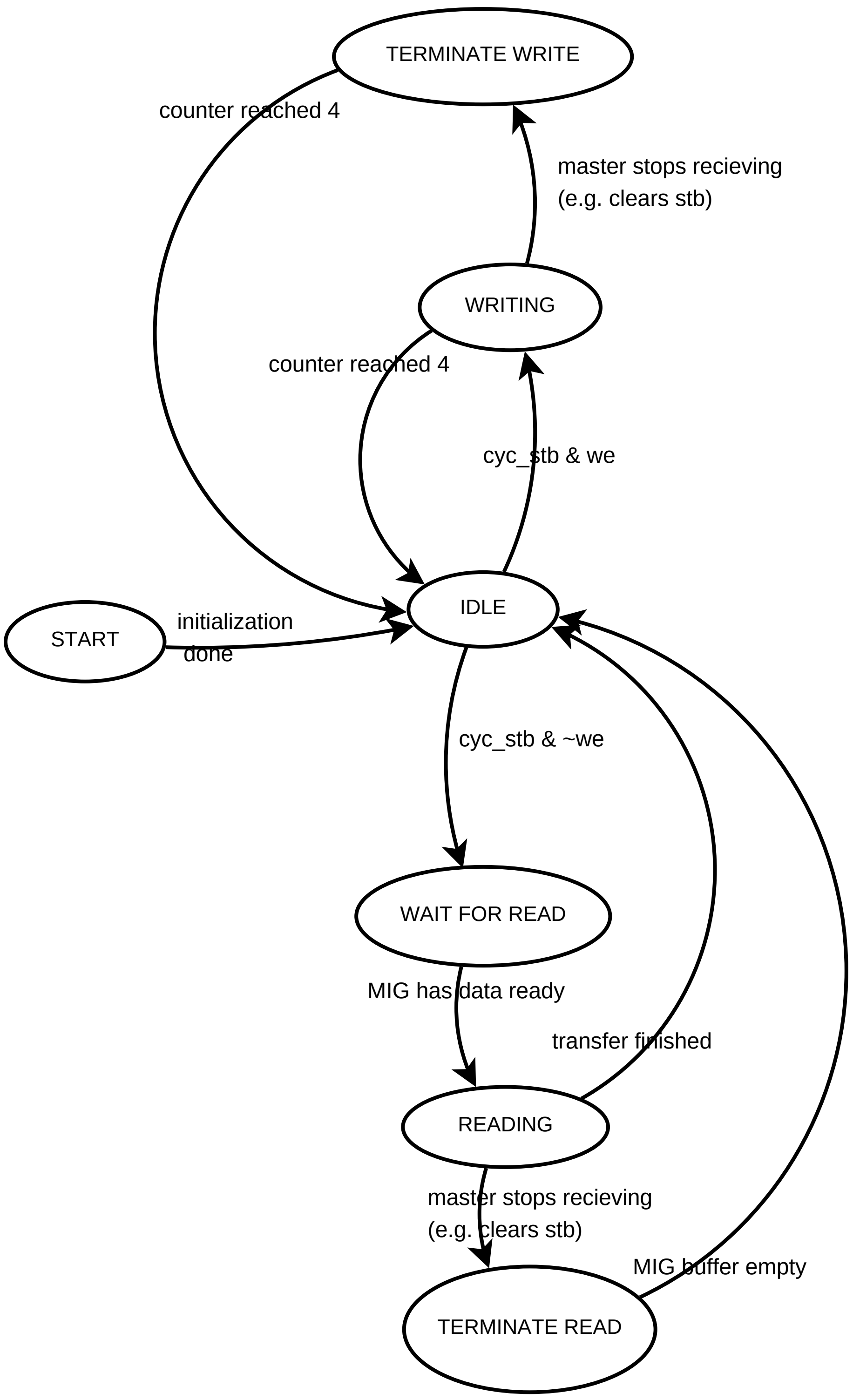


Figure A.2. wb2mig state diagram

Appendix B

Performance charts

The following pages includes charts comparing the XIPS processor to several other well known processors.

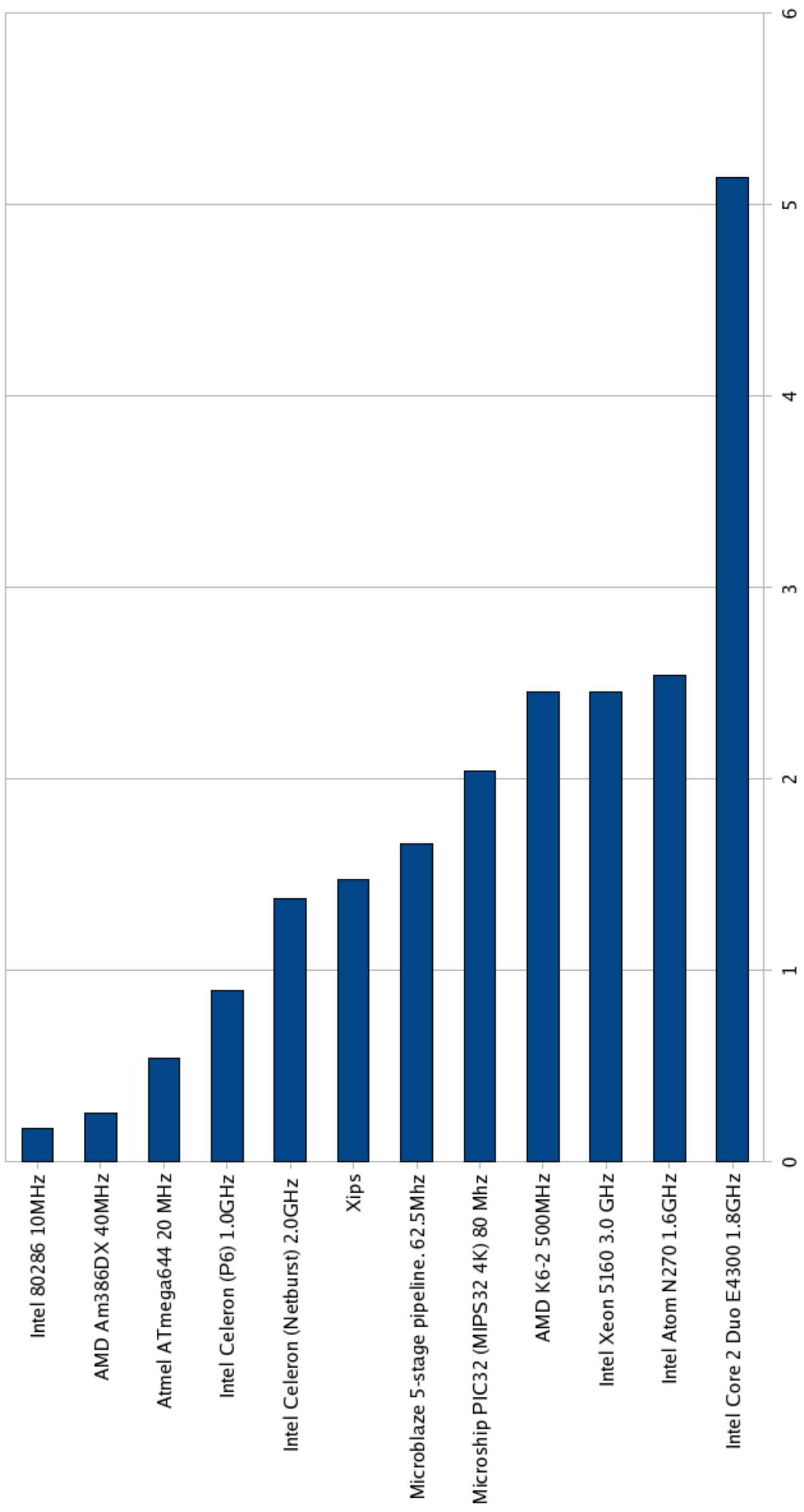


Figure B.1. Coremark/MHz for various processors

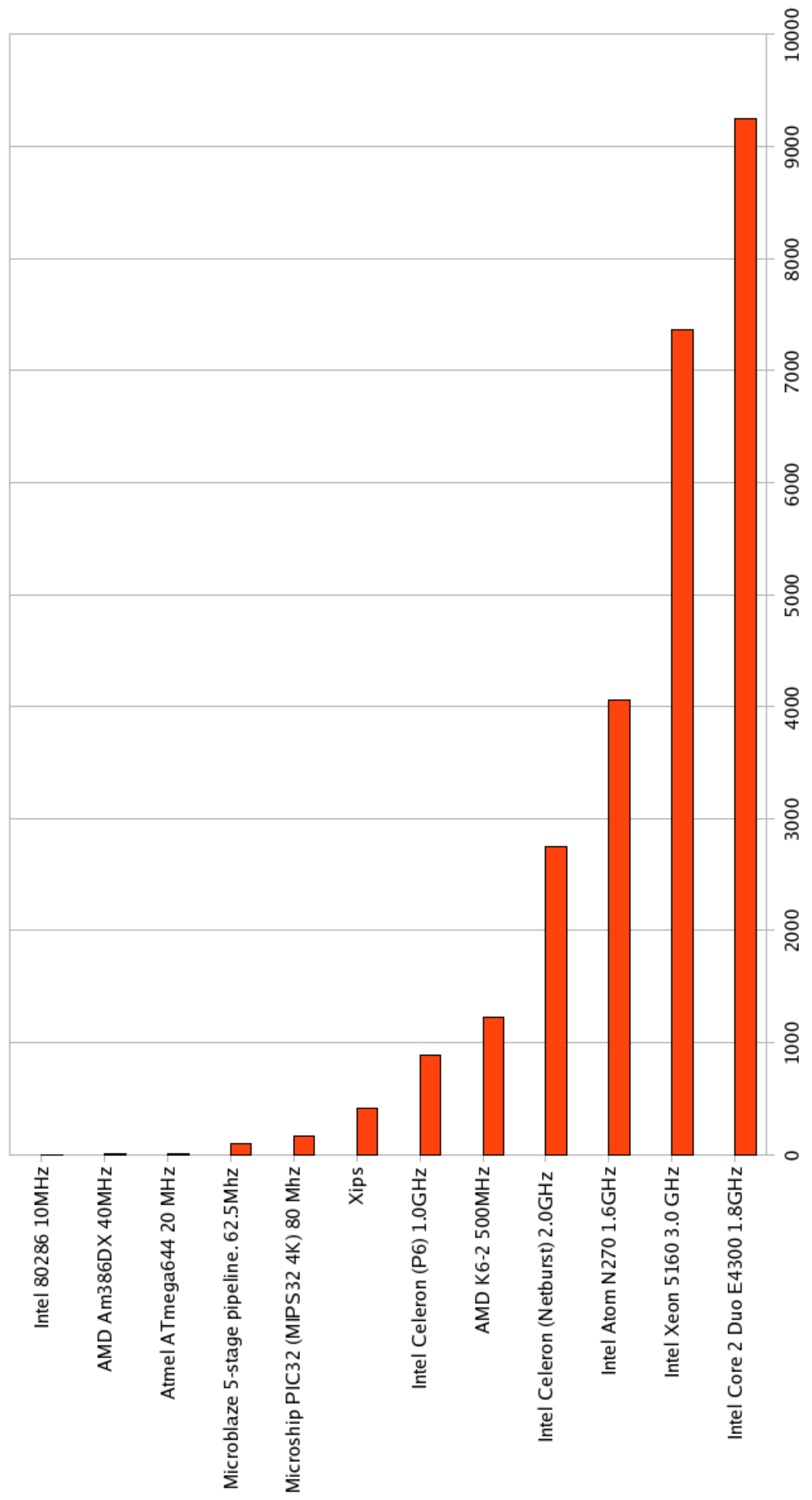


Figure B.2. Coremark score for various processors

Appendix C

xi2 instruction set

ADD `Rd = OpA + OpB`

AND `Rd = OpA or OpB`

ASR `Rd = OpB >>> OpA`

JUMP `conditional using flags, unconditional, subroutine`

LD `Rd = mem[OpB+OFF]`

LDB `Rd = mem[OpA+OFF] (byte)`

LDS `Rd = mem[OpA+OFF] (short)`

LOOP `OpA (high 16 bits, loop end minus 2, low bits`

LSL `Rd = OpB << OpA`

LSR `Rd = OpB >> OpA`

MAC `ACC_int += OpA*OpB`

MAC.F `ACC = (ACC_int += OpA*OpB)`

MUL `ACC = OpA*OpB`

NOP `No operation.`

OR `Rd = OpA or OpB`

SET `Rd = OpA`

SET `Rd = SR(OpA)`

Sign Extend `8->32 bits`

Sign Extend `16->32 bits`

ST `mem[OpB+OFF] = OpA`

STB `mem[OpA+OFF] = OpB`

STS `mem[OpA+OFF] = OpB`

SUB `Rd = OpB - OpA`

XOR `Rd = OpA xor OpB`

Appendix D

XIPS instruction set

This is a complete list of all supported instructions. For more detailed information, please see [5].

ADD Rd,Rs,Rt `Rd=Rs+Rt. Partially implemented, should throw exception on overflow.`

ADDI Rt,Rs,Imm `Rt=Rs+Immediate Partially implemented, should throw exception on overflow`

ADDU Rd,Rs,Rt `Rd=Rs+Rt`

ADDIU Rt,Rs,Imm `Rt=Rs+Immediate`

AND Rd,Rs,Rt `Rd=Rs AND Rt`

ANDI Rt,Rs,Imm `Rt = Rs AND zeroexted(Immediate)`

BEQ Rs,Rt,Offset `if Rs=Rt then branch to PC+Offset`

BGEZ Rs,Offset `if Rs>=0 then branch to PC+Offset`

BGEZAL Rs,Offset `if Rs>=0 then branch to PC+Offset and R31=PC+8`

BGTZ Rs,Offset `if Rs>0 then branch to PC+Offset`

BLEZ Rs,Offset `if Rs<=0 then branch to PC+Offset`

BLTZ Rs,Offset `if Rs<0 then branch to PC+Offset`

BLTZAL Rs,Offset `if Rs<0 then branch to PC+Offset and R31=PC+8`

BNE Rs,Rt,Offset `if Rs!=Rt then branch to PC+Offset`

CLO Rd,Rs `Rd=countleadingones(Rs)`

CLZ Rd,Rs `Rd=countleadingzeroes(Rs)`

DIV Rs,Rt `HI=Rs / Rt, LO = Rs mod Rt Signed.`

DIVU Rs,Rt `HI=Rs / Rt, LO = Rs mod Rt Unsigned.`

J,Addr `PC=PC_{31..28} || Addr`

JAL Addr `PC=PC_{31..28} || Addr and R31=PC+8`

JALR Rd,Rs `Rd=PC+8 and PC=Rs`

JR Rs `PC=Rs`

LB Rt,offset(base) `Rt=mem(base+offset) Load byte, signed.`

LBU Rt,offset(base) `Rt=mem(base+offset) Load byte, unsigned.`

LH Rt,offset(base) `Rt=mem(base+offset) Load halfword, signed.`

LHU Rt,offset(base) `Rt=mem(base+offset) Load halfword, unsigned.`

LUI Rt,Imm `Rt=immediate || 0^{16}`

LB Rt,offset(base) `Rt=mem(base+offset) Load byte, signed.`

LW Rt,offset(base) `Rt=mem(base+offset) Load word`

LB Rt,offset(base) `Rt=mem(base+offset) Load byte, signed.`

MADD Rs,Rt `{HI,LO} = {HI,LO} + Rs * Rt Signed multiplication`

MADDU Rs,Rt `{HI,LO} = {HI,LO} + Rs * Rt Unsigned multiplication`

MFHI Rd `Rd=HI`

MFLO Rd `Rd=LO`

MTHI Rs `HI=Rs`

MTLO Rs `LO=Rs`

MOVN Rd,Rs,Rt `Rd=Rs iff Rt!=0`

MOVZ Rd,Rs,Rt `Rd=Rs iff Rt=0`

MSUB Rs,Rt `{HI,LO} = {HI,LO} - Rs * Rt Signed multiplication`

MSUBU Rs,Rt `{HI,LO} = {HI,LO} - Rs * Rt Unsigned multiplication`

MUL Rd,Rs,Rt `Rd=Rs*Rt`

MULT Rs,Rt `{HI,LO}=Rs*Rt Signed.`

MULTU Rs,Rt `{HI,LO}=Rs*Rt Unsigned.`

NOR Rd,Rs,Rt `Rd=Rs NOR Rt`

OR Rd,Rs,Rt `Rd=Rs OR Rt`

XOR Rd,Rs,Rt `Rd=Rs XOR Rt`

ORI Rt,Rs,Imm `Rt=Rs or Immediate`

XORI Rt,Rs,Imm `Rt=Rs xor Immediate`

SB Rt,Offset(base) `mem(base+Offset) = Rt Store Byte`

SH Rt,Offset(base) `mem(base+Offset) = Rt Store Halfword`

SW Rt,Offset(base) `mem(base+Offset) = Rt Store Word`

SUB Rd,Rs,Rt `Rd=Rs-Rt. Partially implemented, should throw exception on overflow.`

SUBU Rd,Rs,Rt `Rd=Rs-Rt`

SLL Rd,Rt,sa `Rd=Rt << sa`

SLLV Rd,Rt,Rs `Rd=Rt << Rs`

SRA Rd,Rt,sa `Rd=Rt >> sa Arithmetic`

SRAV Rd,Rt,sa `Rd=Rt >> Rs Arithmetic`

SRL Rd,Rt,sa `Rd=Rt >> sa Logical`

SRLV Rd,Rt,sa `Rd=Rt >> Rs Logical`

SLT Rd,Rs,Rt `Rd =(Rs < Rt) Signed`

SLTU Rd,Rs,Rt `Rd =(Rs < Rt) Unsigned`

SLTI Rt,Rs,Imm `Rd =(Rs < Imm) Signed`

SLTIU Rt,Rs,Imm `Rd =(Rs < Imm) Unsigned`