Implementing the 32-bit MIPS-like RISC processor ECO32 in an FPGA

Rolf H. Viehmann

2006-07-30

Abstract

This diploma thesis deals with the implementation of the 32-bit RISC processor ECO32 ("**e**ducational **co**mputer with **32** bits") in an FPGA, using Verilog as the hardware description language.

The processor has been designed by Prof. Dr. Hellwig Geisse for research and teaching purposes, and has been available only as a simulator in the past, running the already ported software. One important part of my work was to find out if the design choices made were really well suited for a hardware implementation. Indeed, some design details were changed in the course of this work.

I will describe the architecture of the ECO32 system, the tools used, the implementation work done by me, as well as some possible future directions for the project. Because it isn't always easy to see if the implementation works as expected, I spent quite some time testing the code created, so I will describe how that was done, too.

Contents

Chapter 1

Introduction

The target of my project was to implement as much as possible of the ECO32 processor in a hardware description language (we used Verilog for reasons stated later), with the goal of running the processor in an FPGA with decent speed and great specification conformance. First, my work concentrated on the handling of interrupts and exceptions by the hardware, with the target of completing this important part of the architecture on time, but actually the time sufficed to work on several other aspects as well, leaving only one part (the translation lookaside buffer) unfinished.

The ECO32 is a 32 bit RISC processor that was designed by Prof. Dr. Geisse with the goal of creating a modern, clean, fairly easy to understand architecture to teach, research and experiment with. The general architecture of the CPU, that strongly resembles MIPS, is described in a separate chapter.

The whole project is active since 2001 (changing it's name from "EduComp" (for "educational computer") to "ECO32" in 2002), and has since grown an offspring, the ECO32e ('e' for embedded), a simpler, smaller version that lacks details like the user mode, interrupts and exceptions, and virtual memory. The ECO32e is used in teaching in the course "Hardware für Eingebettete Systeme" (hardware for embedded systems).

I had access to the whole source code that Mr. Geisse prepared for his course, so my work concentrated on the features that the ECO32e lacks (compared to the "full" ECO32). The tools used for the work are introduced in the next chapter, followed by an overview of the processor architecture. I will then concentrate on the parts I contributed, as well as the rationales behind some of the design decisions.

Chapter 2

Tools

For this ambitious project, several tools, most of them open source, proved helpful. Some of the tools may not seem to be very user friendly at first sight, but after investing some time to master them, they all work very well.

2.1 sim

The ECO32 simulator, developed by Mr. Geisse, simulates the CPU with some attached peripherals (RAM, up to 4 terminals, VGA graphics adapter, hard disk, etc.), and features numerous powerful debugging tools (register dump, single stepping, assemblation/disassemblation on the fly, etc.). Also the source code (ANSI C) of the simulator proved invaluable as a precise and detailed specification of the architecture in it's current form.

The focus of the simulator lies on simulating the hardware as precise as possible, down to the latencies of the disk, etc., instead of simply executing software written for the ECO32 ISA[1] as fast as possible.

2.2 asld

The ECO32 assembler is supplied with the ECO32 simulator package, and works like any assembler, it translates ECO32 assembler code to machine language that can be run on the simulator or on the real machine (that itself can be simulated by Icarus

[1] For a definition of "ISA" in this context see the glossary.

Verilog). There's no standalone disassembler, but the simulator is capable of that (but of course it wouldn't be difficult to write a litte stand-alone version, if needed).

2.3 decrypt

A litte helper written by me, that is useful to "decrypt" (explain) a word (32 bits), that can be interpreted as

- an ECO32 instruction word
- a PSW (processor status word)
- the current irqPnd register
- a simple integer (signed/unsigned)

That way, it's easy to see whether a PSW, the irqPnd register etc. is correctly updated. The functionality will be included into the simulator in the next version.

2.4 Unix tools

As with nearly any software project mainly developed on Unix/Linux, several "typical" Unix tools were used, for example gcc / g++, make (actually gmake[2]), shell scripts, grep, etc. I won't describe these tools here, because they are in no way specific to this project, and most of them are explained in depth in their manpages, Unix books, and on numerous web pages.

2.5 bc

The bash calculator provided very useful, because it can handle arbitrarily large numbers in any base without overflowing, so it can serve as a great learning tool while understanding algorithms. The (command line) user interface may have some quirks, but it's still a great software after all.

[2] "gmake" stands for "GNU make", and is the open source version of make. Gmake is the default on most GNU/Linux systems, and available for almost any UNIX system.

2.6 Verilog 2001

Also known as "IEEE Standard 1364-2001", the version of Verilog supported by all tools we used. It has recently been superseded by Verilog 2005, but the new version is not yet widespread, so tool support would have been lacking, making the choice an easy one. Verilog is sometimes also called "Verilog HDL" (but is **not** the same as VHDL).

2.7 Icarus Verilog

The processor was developed in Verilog, a hardware description language that can be synthesized (see below), but can also be simulated to analyze the behaviour of the described hardware, which is indispensable during the course of development.

We settled for Verilog mainly because of the availability of the free tool Icarus Verilog that simulates a design described in Verilog, and has nearly complete support of the constructs provided by Verilog 2001. What also provided helpful was the Icarus Verilog command line switch "-Wall" that causes the software to emit all supported warnings, making it quite easy to find erroneous code. The Xilinx-Tools seem to ignore some of the errors that can be found by Icarus, making life a bit harder sometimes.

2.8 GTKWave

Icarus Verilog generates waveforms in the widespread format "VCD" (value change dump), that is -in principle- human readable, but directly working with the VCD file would take enormous amounts of time and concentration, so GTKWave is used to display the VCD as nice waveform diagrams.

2.9 Xilinx-Tools

The only used closed source package is provided free of charge with the prototyping board that we used. It's a complete, comprehensive toolchain that can be used to analyze and optimize the timing behaviour of the Verilog code, as well as synthesize a bitstream that is ready to load onto the FPGA of our prototyping board. Xilinx is

the manufacturer of the FGPA chip and the CPLD chip on the prototyping board, so the generated files are Xilinx-specific (but the Verilog source code is not).

Three types of messages are emitted by the Xilinx-Tools: “errors”, “warnings” and “notices”, but the experience has shown that especially the “notices” can be far more important than the name implies, so the generated log file should at least be skimmed, and all “notices” be read.

2.10 Xess-Tools

Xess manufactures the prototyping board we used, and supplies some tools to test the board, load a bitstream, program the FlashROM on the board, etc. They also provide numerous application notes, explaining how to use the devices on the board.

2.11 XSA-3S1000

The prototyping board we used features an FPGA and a CPLD by Xilinx, 32 MBytes SDRAM, 2 MBytes FlashROM, two pushbuttons, a PS/2 port for either a keyboard or a mouse, a VGA Port, etc. It can run with up to 100 MHz, but it quickly became apparent to us that the ECO is too complex to run with the full 100 MHz on this hardware, so we aimed for 50 MHz main clock frequency. This may not sound too attractive, but as clock speed is not the highest priority on this project, and FPGAs are becoming increasingly more capable over time, it is satisfying for now.

2.12 LaTeX

LaTeX is a complete typesetting system, suited perfectly to layout documents like this one, without having to worry about the little typographic details that can cause a headache otherwise. The main strengt lies in the perfect typesetting of complex mathematical formulae, but the system is flexible and universal enough to be used for arbitrary documents.

2.13 OpenOffice.org Draw

I used OpenOffice.org Draw to draw the figures.

Chapter 3

ECO32 architecture

The architecture can (and quite possibly will) be expanded or modified in the future, so every detail I mention is (believed to be) accurate at the time of writing, but could be changed later. Our intention was not only to create a working hardware implementation of the architecture, but also to find out if the existing design is really practical and sensible for a hardware implementation, so we didn't shy away from making some small changes during the course of my work. Of course, all changes have to be made to the simulator as well, to keep the hardware implementation and the software implementation in sync.

The Software that is already ported to the architecture possibly has to be changed as well, but most of it is written in C, limiting changes to the compiler backend or the assembler in most cases.

3.1 Overall architecture

32-bit big endian RISC architecture, with emphasis on a clean, understandable, simple, yet realistic design. It is in many details very similar to the MIPS processor family.

3.2 Registers

32 32-bit general purpose registers, named $0 to $31. $0 always contains 0, $31 takes a procedure/function return address, $30 contains the interrupt return address. All other registers are equal from the perspective of a hardware designer, but they can be given separate roles by the software designers.

Writes to register $0 are ignored, but the value to be stored is calculated no less, to ensure that exceptions get raised[1] if necessary.

The 32 regular registers are accompanied by 5 32-bit special registers that can only be overwritten in kernel mode (see below). Special register 0 holds the PSW, the other 4 are used to control the TLB (Translation Lookaside Buffer).

3.3 Instructions

61 instructions, most of them working only on the registers, only the load and store instructions access main memory and peripherals. Every instruction is exactly 32 bits wide, with the NOP instruction being defined as 32 zeros[2]. Great emphasis lies on regularity of the instruction set, easing implementation and understanding.

3.4 Memory

Single bytes can be addressed arbitrarily, halfwords have to be halfword-aligned, words are always word-aligned. The TLB translates the 32-bit virtual addresses used to 32-bit physical addresses. The main memory (RAM) can be at most 512 MBytes large. The ROM has a maximum size of 256 MBytes. The highest half of the virtual memory address range is only accessible while in kernel mode.

The following figure illustrates the address ranges.

3.5 Peripherals

Peripherals are memory-mapped and only need to support word accesses (halfword- and byte-accesses are possible if they are supported by the peripheral). As soon as virtual memory is present (fully implemented), it will be possible to protect peripherals from being accessed by user-mode processes. There can be a maximum of 256 devices with 1 MByte address space each.

[1] The terms to "throw an exception" and to "raise an exception" are synonyms.

[2] The hardware will interpret this as "`add $0, $0, $0`", which clearly is a no-operation (NOP). There are several other instructions with no effect (like this one), but this is the "official" NOP.

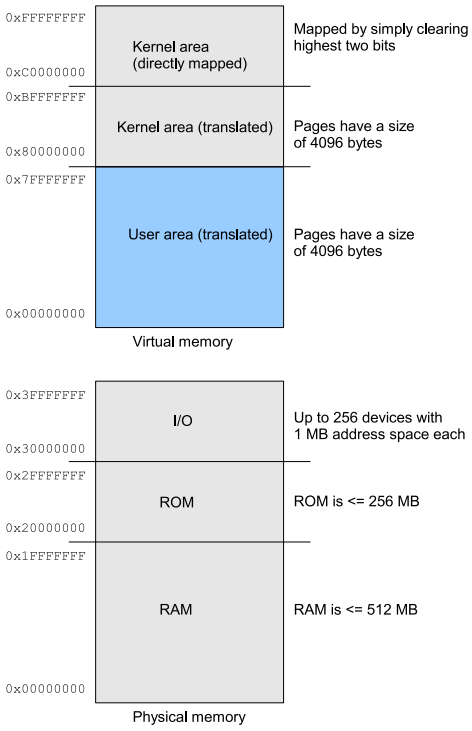

Figure 3.1: Memory architecture and address ranges of the ECO32 architecture. Copyright notice: This graphic was created by me.

3.6 Modes

2 modes, “Kernel mode” and “User mode”, the latter one being restricted to 56 of the 61 instructions. The goal is to run a version of Unix (an ‘ansified’[3] version of Unix v7) on the processor, so every limit on user mode processes that’s present in Unix has to be enforced by the hardware eventually. As soon as the TLB is finished, a write protection on memory pages has to be integrated, to protect the processes from each other.

[3]For a definition of “ansified” in this context see the glossary.

3.7 Supported software

No part of the hardware is targeted towards a specific software, but, as stated, everything that Unix and C need has to be integrated, because we want to be able to run (at least) all of the following software:

- Unix v7 OS kernel ('ansified' by Dennis Kuhn)
- Integer only C programs that can be compiled with lcc (a retargetable C compiler) - a backend for the ECO32 ISA is already present.
- Felix Grützmacher ported the C standard library to our architecture, and Norman Ulbrich ported many of the Unix commands, so all in all we have a fairly sophisticated software environment that right now can be run on the simulator (even though there are some problems left), but of course, eventually it has to run on the hardware, too.

3.8 Privileged instructions

Following a list of the privileged instructions (kernel mode only), and why they are privileged.

- `rfx` (return from exception) - restores (among other flags) the mode flag, should only be used by the handler (which is part of the OS (operating system) and described below). In previous versions of the specification, this wasn't privileged, but I recommended to change this, because a user mode process could periodically check the "previous mode" flag, and execute `rfx` if it was set to "kernel mode", elevating it's privileges.
- `mvts` (move to special register) - write access to the special registers allows arbitrary changes to the PSW, so any user mode process could simply switch to kernel mode if it could write to s.r. 0[4]. The other special regs. are used to control the TLB, which is also reserved to the OS.
- `tbs`, `tbwr`, `tbri`, `tbwi` (TLB handling instructions) - A user mode process that could change the TLB contents could effectively mess around with every bit of memory present in the system simply by giving itself access to it, which would (obviously) be a mayor security hole.

[4]Throughout this document, "s.r. x" stands for "special register x".

3.9 Interrupts/Exceptions

16 interrupts and 16 exceptions can be distinguished. The interrupts are individually maskable and can be switched on and off globally, but the exceptions always get handled (non-maskable). When the processor handles an interrupt/exception, it automatically switches to kernel mode with interrupts disabled.

3.10 Integer arithmetics

Signed and unsigned 32 bit integers can be processed by the hardware directly. Integer overflow does not raise an exception, division by zero does.

3.11 Floating point arithmetics

The processor doesn't contain a hardware floating point unit, and no floating point implementation at all is present, but possibly it will get added eventually.

Chapter 4

Implementation work I've done

There are three fundamentally different ways in which a CPU core can be implemented, single cycle, multicycle and pipelined. Pipelined implementations yield the highest possible instruction throughput, but are complicated to design and debug, so the choice for the first version of the ECO32e and ECO32 fell on the multicycle approach.

Single-cycle-operations are generally quite slow, making multicycle designs the second-best in speed terms, while being relatively easy to understand, expand and experiment with. Later, a pipelined version is envisioned, but only after the multicycled design is complete and working.

4.1 General structure of the CPU

The general structure had already been created by Mr. Geisse, but I had to understand it thoroughly first, before being able to make any noteworthy contributions.

As stated, it is a multicycle design, which means that every instruction executed will take several clock cycles to complete (with different instructions possibly taking a different number of cycles each). The amount of work done in each clock cycle is limited, because the cycle time would suffer too much if too much work had to be done at once. The highest possible clock rate for our FPGA chip would be 100 MHz, but not much work can be done per cycle when aiming for that speed, so we aimed for 50 MHz (every clock speed that can be expressed by $\frac{100MHz}{n}, \quad n \in \mathbb{N}$ can be used by simply dividing the clock signal, so 50 MHz is the second best).

The CPU is composed of several Verilog modules, each having defined inputs and outputs, as well as internal wires and registers. When compared to software

development, each module could be seen as a class, and indeed the modules are instantiated, with each instance being able to have it's own private state. In our CPU, each module ist instantiated only once, but that could change in the future.

The memory (both, RAM and ROM) as well as all peripherals are accessed via the bus with the bus address implicitly selecting the device ("memory mapped devices"). The bus as well as the peripherals are outside the scope of my work (that concentrates on getting the CPU right), but of course I have experimented with them, too, to get an understanding of their workings.

At present, the CPU is composed of the following modules:

- **cpu** (central processing unit)
  The top level module that instantiates each other module, it is connected to the bus (for reads/writes), has a 16-bit input for the interrupts, as well as a clock input.

- **ctrl** (control)
  By far the most complex module with connections to practically every other module inside the CPU. The control contains the state the CPU is in, on which most control output lines depend. Because it is very difficult to descibe the state transitions using words, please refer to the following figure for details.

- **pc** (program counter)
  One of the most simple modules, basically encapsulating the program counter and allowing write access to it only if the corresponding control output is set.

- **mar** (memory address register)
  Similar to the pc module, the mar holds a memory address that is fed to the bus (which is needed because the address for the bus access is guaranteed to stay constant until the device on the bus has finished the read or write).

- **mdor** (memory data out register)
  Just like the memory address, the data to write (on a write access) is defined to be constant until the data is written, so this unit holds it constant as long as needed.

- **mdir** (memory data in register)
  Because it is handy, the data read from the bus is kept inside the mdir until the next bus read.

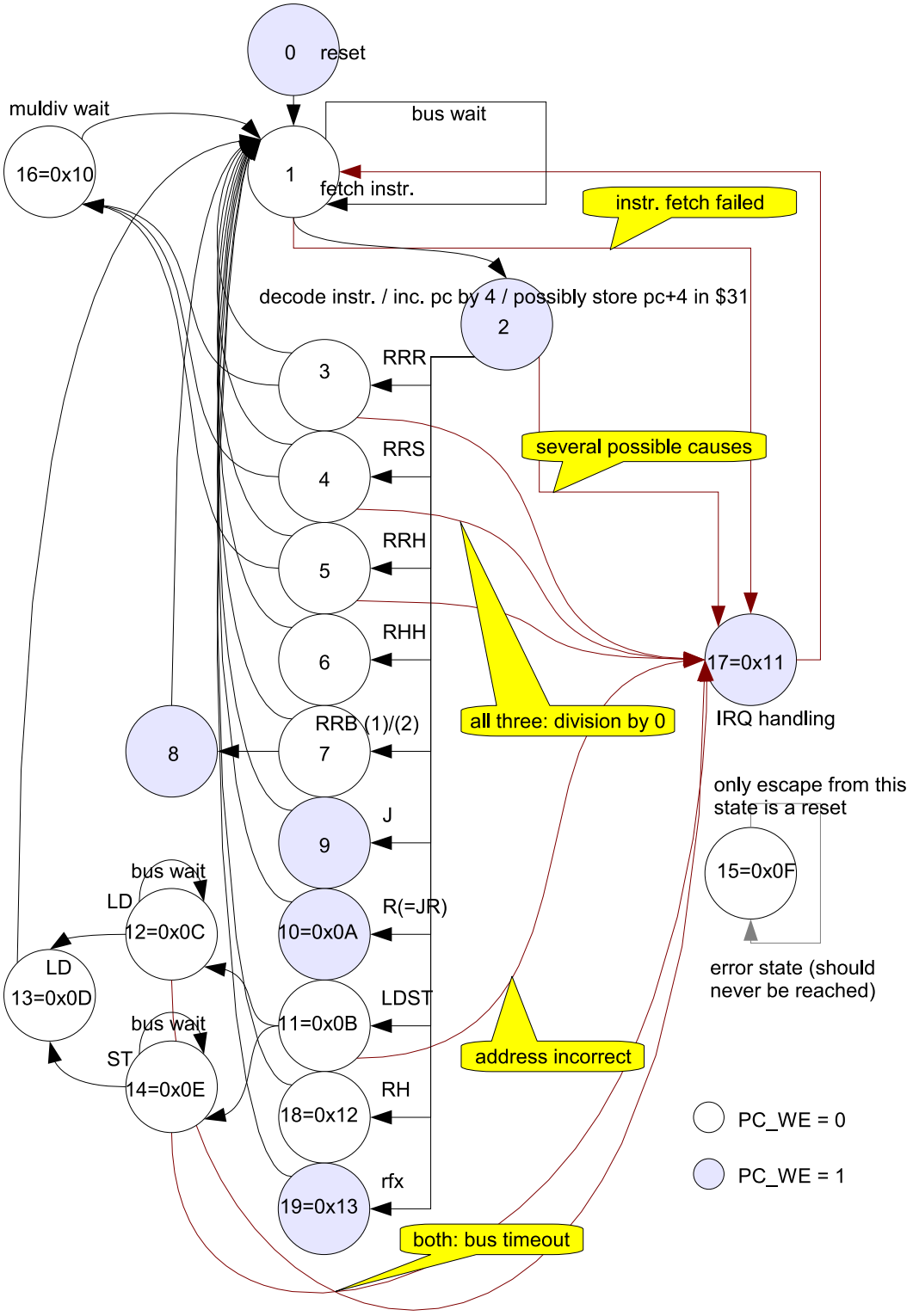

Figure 4.1: CPU states and transitions with their reasons. Copyright notice: This graphic was created by me.

- **ir** (instruction register and decoder)
  The instruction register holds the current instruction word, and decomposes it into it's components, zero-extending or sign-extending them as needed. Due to the relatively simple, regular instruction encoding on our platform, this module is quite compact as well.

- **regs** (register file for regular registers)
  The 32 regular registers are preserved by this module, which simply describes the behaviour of a register file, the Xilinx-Tools are powerful enough to infer[1] a BlockRAM[2] automatically, which is exactly what we needed anyway. If this wouldn't work, one would only have to change this module, instantiating a BlockRAM manually.

  This is actually a good example for clean encapsulation, because this module has just a few inputs/outputs and a defined behaviour, the implementation details are completely up to the module author. In contrast, the control is connected to nearly every other module, making it an inseparable part of the overall architecture, but in some cases (as with the control) this can't be avoided.

- **alu** (arithmetic/logic unit)
  The ALU is composed only of combinatorial logic, not containing any state, and is limited to functions that are quite fast, yielding a useful result in the same clock cycle.

- **shift** (shift unit)
  I've decided to give the shift operations their own unit, because they are more complex (and thus slower) that the other arithmetic/logic operators, so the result is ready one clock cycle after the input changed. What makes the hardware for the shifts more complex is that fact that both the value to shift, as well as the shift amount are variable, if the shift amount is fixed, the hardware needed for the shift becomes extremely simple.

- **muldiv** (`mul/div/rem` unit)
  I designed yet another unit for the multiplication / division operations, because they take numerous clock cycles to calculate, and I didn't want to mix faster and

[1] To "infer" means that the software is free to implement the structure (defined by the Verilog source files) in any possible way, depending on the capabilities of the target hardware.
[2] For a definition of "BlockRAM" see the glossary.

slower operators inside one unit, to simplify the usage of the modules. Because the module instantiating this one may not know how long exactly the calculation will take, an output line goes high as soon as the results are complete. This way, the internal implementation (and the speed achieved by it) may change any time.

- **sregs** (register file for special registers)
  The register file for the 5 special registers the ECO architecture contains is in principle similar to the register file for the regular registers, but there are some noteworthy differences. The Xilinx-Tools don't use a BlockRAM to store the contents of the special registers, normal flipflops are used instead, probably due to the odd number of special registers. This is no problem at all, indeed, it opens up some additional possibilities, for example, several (more than two) registers can be outputted at once, making the other modules a bit simpler (the PSW (special register 0) and the TLB bad address (s.r. 4) are permanently fed to the control, which uses the values whenever needed).

4.2 Multiplication/Division

Part of my work was to implement the multiplication and division instructions that operate on signed and unsigned integers, and are part of the current ECO32 instruction set. Verilog provides operators for multiplication, division and remainder calculations, but they are not synthesizable, or at least not with decent speed, so I didn't use them. Instead I successfully tried to implement two algorithms taken from Patterson/Hennessy [COD2e].
The ECO contains six instructions to work with multiplication and division, they are:

`mul`	signed multiplication
`mulu`	unsigned multiplication
`div`	signed division
`divu`	unsigned division
`rem`	remainder of a signed division
`remu`	remainder of an unsigned division

I designed a separate unit for the multiplication/division/remainder operations, because they take several clock cycles to complete, in contrast to the "normal" ALU that only tackles fast operations that provide their result in the same clock cycle. The unit

signals it's state (running or complete) to the control unit that may only use the result when it's complete.

Twos-complements (that are extensively used in this unit) can be calculated "on the fly" (on the same clock cycle) using combinatorial logic, so they don't slow down the calculations. Shifts with a fixed shift amount are equally fast.

4.2.1 Multiplication

The first algorithm (simply named "third version of the multiplication hardware") describes integer multiplication using a loop that runs for 32 iterations, and is generally quite simple. Each iteration completes in one clock cycle, but it may also be possible to calculate more than one iteration per clock cycle. That way, calculations would complete faster, with the disadvantage of longer, harder to understand code and probably higher area usage on the FPGA. Because of limited time, I didn't try to implement the algorithm in this faster way, so there's room for future experiments.

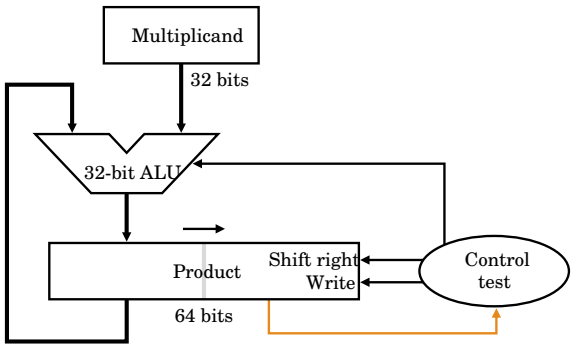

Figure 4.2: Hardware necessary for the multiplication. Copyright notice: This graphic is taken from [COD2e], see chapter "List of Figures" for details.

My implementation deals with signed integers in a simple, straightforward way: If the signs differ, that fact is stored for later reference, afterwards the absolute value of both arguments is used to calculate the result. If the signs did differ, the twos-complement is returned as the result of the calculation, leaving the main loop simple and efficient.

An alternative would have been to use Booth's Algorithm (also described in [COD2e]), but it is more difficult, and not necessarily faster (on nowadays hardware).

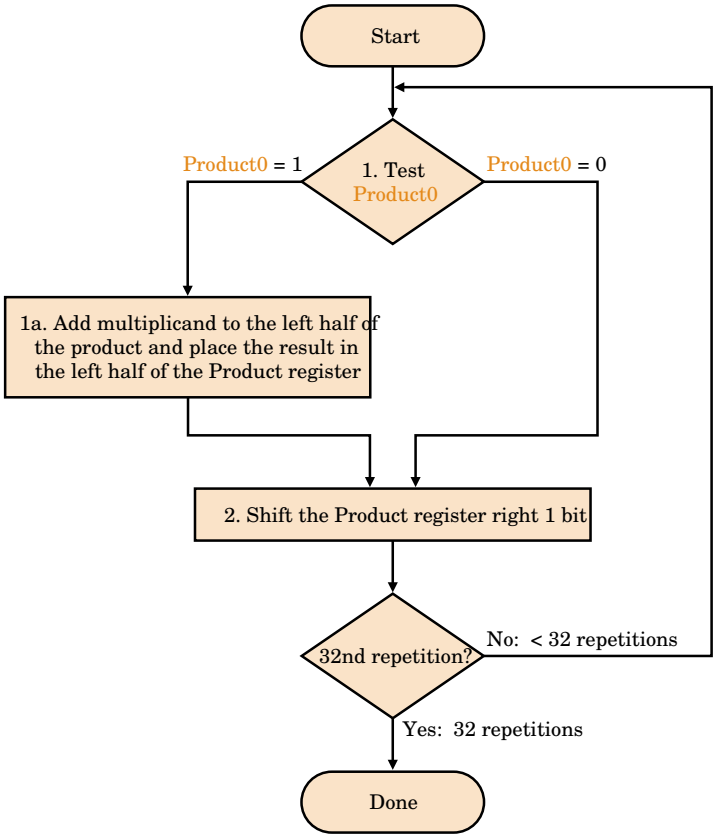

Figure 4.3: Flowchart describing the algorithm for the multiplication. Copyright notice: This graphic is taken from [COD2e], see chapter "List of Figures" for details.

4.2.2 Division

The hardware that is needed for the multiplications (at least most parts) can also be used for the division/remainder instructions, especially the 65 bit register. For the multiplications, 64 bits are sufficient, but when dividing unsigned 32 bit values, 65 bits are needed, a fact that isn't stated by Patterson/Hennessy, but became clear during testing the unit. The algorithm used for divisions is also taken from Patterson/Hennessy, again without a melodic name, simply called "the third division algorithm". Being very similar to the multiplication algorithm, it also takes 32 clock cycles to complete.

The handling of signed values is also similar to multiplication, the signs of the operands are used to generate two flags:

neg_quotient: is true if the signs of the operands differ, and specifies that the twos-complement should be returned for the `div` instruction.

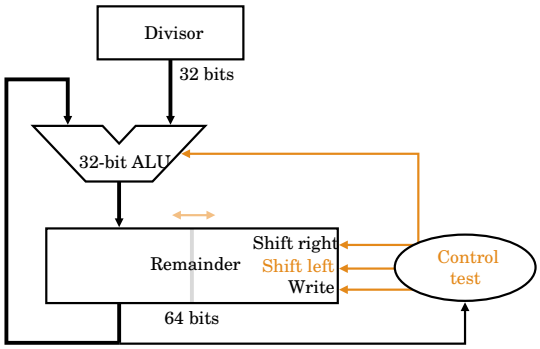

Figure 4.4: Hardware necessary for the division. Copyright notice: This graphic is taken from [COD2e], see chapter "List of Figures" for details.

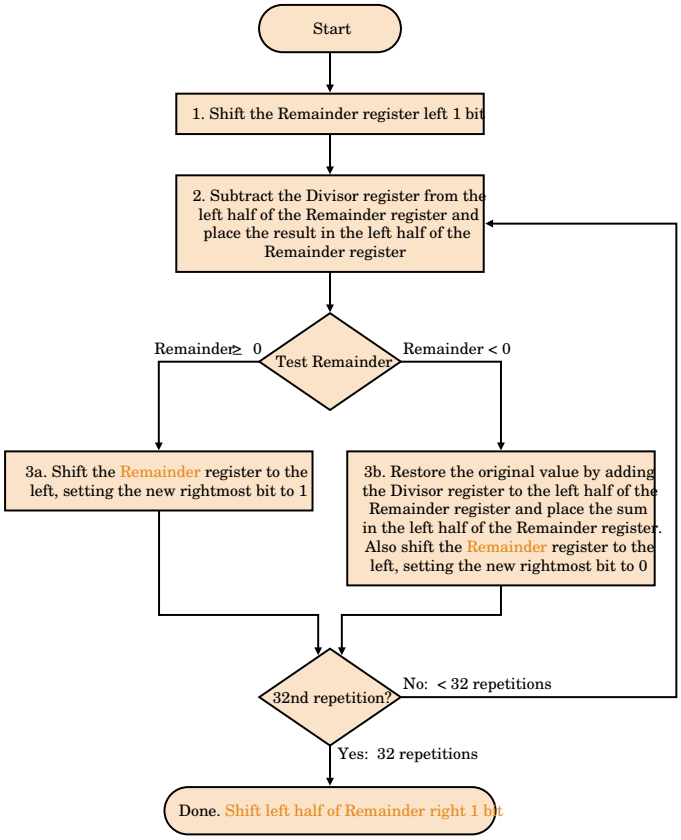

Figure 4.5: Flowchart describing the algorithm for the division. Copyright notice: This graphic is taken from [COD2e], see chapter "List of Figures" for details.

neg_remain: matches the sign of the dividend, and specifies to return the twos-complement for the `rem` instr.

A quotient or remainder can be calculated in 32 clock cycles, just like a product, but with a difference: The quotient and remainder are calculated in parallel (simply because the algorithm is defined that way). It's not possible for the user (of the processor) to take advantage of this fact, yet, but it would be possible (and maybe a good idea) to create a special register that holds the other value (rem. when dividing, quotient when calculating the modulus). After a multiplication, this special register could hold the upper 32 bits of the product. Writing the desired result to the regular register file, and writing the "additional result" to the special registers file could be done simultaneously, not slowing down the instruction.

A division by 0 is caught by the control already, overflow is not caught (our architecture is designed that way), so this unit does not contain any facilities to signal invalid operands, overflow or other problems.

4.3 Shift operators

The shift operators were relatively easy, because Verilog already provides unsigned shifts (logical shifts) that are synthesizable. The signed shift (arithmetic shift right) was built using two logical right shifts, not and or:

```
if (mode==2) begin // sar
   if (in[31] == 1) begin
      // replicate sign bit
      out <= (~(32'hFFFFFFFF >> shamt)) | (in >> shamt);
   end else begin
      out <= in >> shamt;
   end
```

The shift-amount is defined to be 5 bits wide on the ECO32, so only the lowest 5 bits of the second operand are used. While testing the Verilog implementation, a hard-to-find error in the simulator sim was revealed, here is the original code:

```
case OP_SAR:
   scnt = RR(src2) & 0x1F;
   smsk = (RR(src1) & 0x80000000 ?
      0xFFFFFFFF << (32 - scnt) :
```

```
    0x00000000);
WR(dst, smsk | (RR(src1) >> scnt));
break;
```

Seems to be absolutely correct, but C can make things difficult by featuring incomplete definitions. To quote from the original C manual:

> The result of a shift is undefined if the right operand is negative, or greater than or equal to the number of bits in the left expression's type.
> **K&R, Appendix A7.8, Shift Operators [KRC]**

In most cases, the above code is correct, but I found one case where it errs: if the shift-amount is 0, the subtraction (32 - scnt) is equal to 32, which is -in turn- equal to the width of the left operand. The shift result is not defined in this case, but the C compiler does not print a warning and emits code that may fail under rare occasions. In my opinion, the decision to **not** define the results with any possible operands was a bad one, making life unnecessarily harder for C programmers, but C is generally not exactly known for programmer friendliness.

4.4 Signed branches

Mr. Geisse already had implemented the unsigned branches, so I contributed the signed branches. The difference lies (as the names imply) in the way the operands are interpreted, as a 32 bit signed vs. 32 bit unsigned quantity. The ALU has three output lines that are important for the branch logic:

alu_ult high if (unsigned)a < (unsigned)b
alu_lt high if (signed)a < (signed)b
alu_eq high if a==b

The signed branches only observe alu_lt and alu_eq, the unsigned branches only (you guessed it) alu_ult and alu_eq. I generate the alu_lt signal simply by subtracting b from a, but first they both have to be sign-extended to 33 bits, yielding a 33 bit result. That way, I can be sure that the result never overflows, and I can simply check the highest bit of the result. If that bit is set, the result is negative, so a is lower than b.

It's very fortunate that registers, wires etc. in Verilog can have a bit width that's **not** a power of two, because that way, results of internal calculations can be wide enough to prevent any overflows. Both alu_ult and alu_lt are calculated via a 33 bit

register, with the operands sign extended for the (signed) 'lower than' signal, zero extended for the 'unsigned lower than'.

4.5 `mvts`/`mvfs` instructions

The `mvts` and `mvfs` instructions were straightforward to implement due to the design of the module for the special registers (that mimics the module for the regular registers). Both register files have connections from one data output of one register file to the data input of the other, making transfers between them a fast operation.

Because `mvts` is a privileged instruction, the transfer is not initiated at all by the control if it is not allowed, both register files know nothing about privileged instructions, making the design simpler. This holds true for most other modules as well, only the control cares about the two modes and prevents unwanted execution of privileged instructions.

4.6 `jalr` instruction

The only jump left to implement was the `jalr` (jump to register and link) that basically does two things at once: The PC is replaced by the value of the specified register, and the (old) value of the PC is saved to register $31.

We had a little discussion over the desired semantics if the specified register is also $31, but came to the conclusion that it makes little sense to do a `jalr` to register $31 in any case, so we decided to define that it's discouraged to to so. We didn't make "`jalr $31`" an illegal instruction, because we were unsure about the implications on a future pipelined version, so it's only discouraged, not raising an exception. This is actually in conformance to the "archetype", the MIPS processor family, that acts exactly alike in this regard.

Implementing the instruction was really simple, because the other three jumps were already present, so there was little left to invent, but the work on this instruction had a positive side aspect, because I stumbled over an error I had made while integrating the exceptions: The instruction fetch is (of course) a normal word-wide read from memory that has to raise the same exceptions as every other read, but it didn't. Fortunately, this was fixed quickly.

Chapter 5

The role of speed

As previously mentioned, highest possible speed was not the single most important consideration, in contrast to commercially available hardware (where clock speed is often the most prominently displayed feature). Very high speed is often only achievable by taking much effort and complicating the design considerably, but we aimed for a clean, expandable, relatively easy to understand design mainly for teaching and research purposes. On the other hand, faster is always better, so we had an eye on the speed of our design all the time.

On a multicycled design, execution speed depends on two factors: the **clock speed** and the **amount of work done per cycle**. If the complexity of a circuitry is too high, the results of a computation may not be ready when the next cycle begins (that uses the results of the previous one as it's input).

If a certain clock speed has to be possible (we wanted to hold the 50 MHz mark, as stated), the most complex paths may have to be split up to decrease complexity, preserving the values inside registers[1] between cycles. That actually has been done in some parts, for example when accessing the bus.

But not every complex circuitry is a problem, because the Xilinx-tools are more powerful than one can see at first sight, especially helpful is the "effort level" setting that can be turned up by the hardware designer. It could be compared directly to the optimization settings of most modern compilers, where a higher level can bring notable speed improvements, at the cost of a prolonged compilation.

[1] Here, "registers" are not to be mixed up with registers a assembler programmer is used to. When designing hardware, a register simply is a location to store values (of a specified width), without necessarily making these registers visible to any software running on the designed hardware. The tools used are free to replace a register that is never changed with simple logic, if they see fit.

Indeed, the steps needed to generate a programming file take much more time when the effort level is increased, but only by utilizing the higher level, our speed goal is met. The advantages over modifying the design manually are twofold. On one hand, it's faster to let the computer work a few minutes instead of spending hours experimenting with the design, on the other hand, it's smarter because the tools still implement the design exactly like stated in the source files, so all the work is still done in one cycle, not split up into several cycles.

A big difference between hardware- and software-design also lies in the fact that software is traditionally specified in a sequential way (one instruction after the other), but hardware is different. Each part of the circuit can do some amount of work in each cycle, working in parallel, which can (and should) be utilized by the designer to speed up the design. This especially holds true when, like in our case, the multicycle approach is used, because most of the units are used only once per instruction, so they are available in the meantime.

A great example for a parallel design will be the TLB, that searches all it's entries simultaneously, whereas the simulator has to search them sequentially, at least nowadays[2].

Depending on the capacity of the FPGA, it may even be a good idea to instantiate units more that once, to seize their capabilities in different parts of the design. The tools may actually do that automatically when optimizing for speed[3].

[2]With multicore-architectures becoming popular recently, it's likely that future compilers and programming languages will also enjoy the benefits of parallel processing more and more, but when designing hardware, parallelism is nothing new.

[3]Just like a compiler, the two possible optimization goals are **speed** or **size**, the former being the more relevant in practice.

Chapter 6

Semiautomated testing

Every hardware or software can, and most of the time does, contain bugs, so I spend quite some time chasing bugs until I had the impression that I had found them all.

6.1 Arithmetic and logic operators

Clearly it's important to test the implementation of any algorithm or operator. My first, naïve thought was to test every possible bit combination for the two operands in combination with every operator, but that's clearly impractical on today's hardware. For example, the possible number of bit combinations for the operands is $(2^{32})^2 = 2^{64} \approx 1.84 * 10^{19}$. Combined with the number of arithmetic and logic operators present in the ECO architecture, the automated test would take far too long (as in "a few millennia too long").

So I aimed a bit lower, testing only a few dozen operand pairs at once, generating VCD files that are still usable in combination with GTKWave, even though loading GTKWave takes about half a minute on my computer. I've written a short C program "autotest" that automatically generates an ECO32 assembler file that roughly works as follows: First of all, a few "known good" instruction have to be present and flawless, because they are needed to run the test. They are (at a minimum): `add` (both variants, three registers and two registers/one immediate), `ldhi`, `or`, `beq`. For each test, the operands ('a' and 'b') are selected from a list or randomly created. They are then placed in two registers of the processor, $9 and $10. For each operator that is to be tested, the correct result is calculated by the C program, and placed in a register ($12). For this calculation, the source code of the simulator comes in handy, because it already contains a C code implementation of every ECO operator.

Overview of the used registers:

$8	faults counter
$9	a (first operand)
$10	b (second operand)
$11	calculated result
$12	correct result

A little obstacle lies in the fact that the C function rand() does not necessarily return 32 random bits, on my computer, only the lowest 31 bits were used ($RAND\_MAX == (2^{31}) - 1$). On other computers, RAND_MAX can be even lower. I used a simple trick to obtain 32 random bits: I simply used rand() twice for every new a and b, once for the upper half, once for the lower half of the register. This was a good idea, especially because divisions of operands with the highest bit set did prove erroneous at first.

Now code is emitted that calculates the result on the simulated (or real) processor, and places the result in yet another register ($11) - there are enough of them present, so they can be used lavishly. The next instruction compares the latter two registers, and goes on to the next test if they agree. If they don't, the error counter ($8) is incremented, then the next test starts. If $8 is still 0 at the end of the test series, the operators work flawlessly, at least with the values used for the test series.

To be able to show the registers with GTKWave, I used a little trick, which was necessary because GTKWave is able to display arbitrary vectors of wires or registers, but not arrays of vectors (like the main register file is one). So I created a few wires for the registers that I wanted to watch, continuously assigning the values from the registers. That way, one can watch the values, but because the wires are not connected to anything, the Xilinx-Tools sort them out, not creating any structure on the FPGA for them, so no resources are wasted.

To also tests the operators with two registers and an immediate, the autotest program also emits code with 'b' specified as an immediate value, but there are a few points to keep in mind:

- The immediate field is only 16 bits long, so only the lower half of b can be specified.
- The assembler may substitute an instruction with an immediate value with several instructions that first fill a register with the desired value, then use that register instead of the immediate field. To prevent this (we want to test the immediate variant after all), the assembler directive ".nosyn" is written to the first line of the generated assembler file.

- Because of the limited size of the immediate, the expected result has to be calculated with this shorter value, zero-extended or sign-extended, depending on the type of the operand being tested. So it is first limited to 16 bits, written to the register, and then extended again (zero- or sign-extended depending on the instruction) to calculate the correct result.

The combination of known values (that can be chosen to test edge cases that are difficult to implement or likely to fail) and random values (with all bits being completely random), one can be quite confident that errors are found quickly. And because GTKWave can be used as usual, the cause of a fault found can be analyzed easily.

6.2 Branches

The described mechanism for automatic testing can be expanded to branches as well. They don't write a value to any register, but they are taken (or not taken) depending on the values a and b, so the idea is to:

- decide whether the particular branch should be taken or not (depending on the current a and b).
- emit ECO assembler that is different when the branch has to be taken vs. when is mustn't be taken.
- when the branch has to be taken, it branches to the instruction after the next one, the next one being an increment of the error counter. The generated code would look like:

```
; the branch has to be taken
   bleu $9, $10, JMP_00000019
   add $8, $8, 1   ; increment error counter
JMP_00000019:
```

- when the branch mustn't be taken, the code is a little more complex, and looks like this:

```
; the branch mustn't be taken
   bltu $9, $10, JMP_INC_00000020
   j JMP_NOINC_00000020
```

```
JMP_INC_00000020:
   add $8, $8, 1   ; increment error counter
JMP_NOINC_00000020:
```

When the branch is taken, it branches to the `add` instruction, otherwise (when everything is correct) the `add` is skipped by the use of the `jump` instr. (that has to be working already[1]).

6.3 `mvts`/`mvfs` instructions

Testing the `mvts`/`mvfs` instructions was quite simple, in part, because the instructions don't have to calculate any new values, in part because the special registers 1-4 don't really have a special function yet (that will come when the TLB is implemented). I contrived the following assembler test program:

- First, copy the contents from 5 normal registers to 5 other normal registers ("shadow registers") for later comparison
- Move the contents to the 5 special registers
- Clear the original 5 normal registers
- Move the values from the special registers to the normal registers
- Compare with the contents of the shadow registers (there should be no difference)

Of course, the value going to s.r. 0 should not cause a switch to user mode, otherwise the `mvfs` will fail.

6.4 Interrupts/exceptions

Testing the interrupts and exceptions was a bit of a "time killer", because I had no inspiration how to test them automatically, so I did it mostly manually. There are different details that have to be checked for correctness, namely:

[1] If no `jump` instr. is ready yet, there's another possibility: The jump can be replaced by `add` with immediate "-1", so if the branch is not taken (which is correct), the counter is first decremented, then incremented. This could also be a bit faster than the jump, at least on a pipelined implementation.

- $30 should always get the correct value (the value of the PC), not any higher or lower value
- The testbed should not output any errors (actually it shouldn't do any output)
- When leaving the IRQ handling state (after which the first instruction of the IRQ handler code will be executed), the PSW has to change to the correct value (can relatively easily be checked with the 'decrypt' software).
- highest_lvl should always contain the correct value (ID of the highest active IRQ (that the handler has to process))
- No register (ordinary or special register) may contain bits that are undefined ('x') or high-z ('z') at any time, especially not the PSW (otherwise chaos is near because the behaviour of the machine is not defined any more)

I enabled the timer by writing a value of 120 (the precise number does not matter) to the relevant location. That way, timer interrupts were generated periodically. To raise exceptions, I crafted instruction that violate a condition, so that the hardware had to raise the corresponding interrupt.

I padded the exception-causing instructions with simple `add` instructions, with linear increasing immediate values. That way, I was able to see if the `rfx` instruction worked as designed. The handler was quite simple, increasing the value of $30 by 4 (in case of an exception) to skip the offending instruction (to see if this is cleanly possible). Interrupts did not get the register $29 modified, because no instruction must be skipped after leaving the handler.

The `add` instructions (with the increasing immediate values) all wrote to $8, so I could see if any number were skipped or if the sequence was complete. Because many of the crafted-to-raise-an-exception instrs. also tried to write to register $8, I could see if none of them actually changed the value of the target register.

The `trap` instruction simply was one of the these instructions, and the `rfx` instruction was tested as well, because it was part of the simple handler code.

6.5 `jalr` instruction

I also had no flashes of genius on how to test the `jalr` instruction completely automatic, so I took the same approach as with the exceptions/interrupts, crafting some

simple assembler code that features all jump instructions, and is likely to fail if any of them is not working. It simply skips over some `add` instructions that would set register $8 to 0x55 (arbitrary value), so if any one of the jumps would not work, the value 0x55 could be found in reg. $8. Also, because the program loops endlessly, any instruction that is not part of the loop would signal an error.

Chapter 7

Raising and handling interrupts and exceptions

7.1 Overview

The biggest part of my work was the implementation of interrupts and exceptions, comprised of the following components:

- special registers (with s.r. 0 being the most important, because it contains the processor status word)
- instructions for moving values to and from the special registers (`mvfs`, `mvts`)
- an instruction that deliberately causes an exception (`trap`)
- an instruction to continue "normal" operation (`rfx`) ("normal" as seen from the software perspective)
- additions to the overall implementation to trigger interrupts/exceptions whenever necessary

To make things work, I've made several extensions to the architecture, consisting of

- new states (for the `rfx`-instruction, for the IRQ handling)
- new state transitions (to/from the IRQ handling state)
- new control signals (assigned by the control depending on the current state)

- new case distinctions for some control signals (they depend not only on the state any more)

Each component will be described in detail below.

7.2 Definition of 'interrupt' vs. 'exception'

Interrupts and exceptions (I use the umbrella term "IRQ" when I mean both without differentiation) form a vital part of any processor architecture, because they form the basis for TLB handling, secure multitasking (where no user mode task can hog up the whole CPU time), error handling, etc.

Please note: On some platforms, the terms 'interrupt', 'exception' and especially 'IRQ' are used in a different manner, but there seems to be a bit of chaos on the desired semantics. So I choose to use them with the semantics stated here.

All IRQs cause the CPU to execute a special piece of code, called the "IRQ handler", "ISR" (for "Interrupt Service Routine"), or just "handler" that is (normally) a part of the operating system. Each IRQ is specified by it's number that gets written to the PSW right before the handler is executed, so the IRQ can be processed accordingly.

The handler can be located at one of two possible locations, ROM_BASE+4 or RAM_BASE+4. Which one is used is determined by the value of the V-Bit (Vector-Bit) located at bit 27 of the PSW. EXC_TLB_MISS uses different handler addresses, explained below.

Whenever the handler is called by the hardware, $30 is overwritten with the PC (the address of the instruction currently executing, or getting fetched). This is important in two situations, first, to make it possible to continue with program execution (if desired), and second, when using the `trap` instr. In the section about the `trap` instruction, the details will be presented.

The main difference of interrupts and exceptions lies in the fact that exceptions are being caused by the currently running instruction (or a nonsuccessful instruction fetch), whereas interrupts have nothing to do with the running code, and are instead caused by events external to the CPU. The other noteworthy difference lies in the fact that it's possible to mask the interrupts, but not the exceptions (because they are important and need immediate action).

If an instruction has multiple faults at one (for example, EXC_PRV_ADDRESS and EXC_BUS_ADDRESS), only the exception with the highest ID is raised and pro-

cessed. Only if the handler code causes the CPU to reexecute the instruction, another exception could get raised (or the same one again if nothing has changed).

7.3 Exceptions defined so far

16 exceptions can be supported by the current architecture, but only 10 are defined so far, leaving room for future expansions.

7.3.1 0x19 - EXC_WRT_PROTECT

(Write protection exception - Status: todo when TLB is present)

This exception will be raised when a store instruction can't be executed, because the processor is in user mode and the memory page is write protected. The fine details still have to be defined.

Other protections (read, execute) would also be thinkable, but it remains to be explored if they would really be useful. If they are ever implemented, they certainly will get their own exceptions.

7.3.2 0x18 - EXC_PRV_ADDRESS

(Privileged address exception - Status: fully implemented)

There are two scenarios for this exception, either the PC points to a privileged address while user mode is active, or a load or store instruction tries to access a privileged address (also only in user mode). The "secret" behind privileged addresses is that they are reserved for kernel usage, and are completely inaccessible while in user mode. The privileged addresses have the highest bit set, while the non-privileged don't. The addresses with the two highest bits set are directly mapped to hardware addresses, without translation via the TLB (the two highest bits are simply masked away). This is very useful, for example to access I/O (that is memory mapped) directly, but clearly no user mode program may be able to use these addresses, circumventing the TLB.

7.3.3 0x17 - EXC_TLB_DBLHIT

(TLB double hit exception - Status: todo when TLB is present)

Will be raised when the TLB finds more than one entry (physical address) for a given logical address, which is obviously an error, because any address (logical or physical) must be unambiguous. Normally, this exception does never occur, if it does, the OS has made a serious error. Typically, the whole computer will be halted. There even existed a version of MIPS that could suffer hardware damage when a TLB doublehit occured.

7.3.4 0x16 - EXC_TLB_MISS

(TLB miss exception - Status: todo when TLB is present)

When the TLB does not contain a mapping for a memory location, the OS must come to the rescue, providing the TLB with the needed information. The code that does this should be as fast as possible, otherwise the whole computer would be slowed down quite a bit. To save clock cycles, this exception is the only one that gets a special treatment, namely a different handler address (ROM_BASE+8 or RAM_-BASE+8). The handler code at this location can begin it's work without checking the exception type, because it should only get called on a TLB miss.

7.3.5 0x15 - EXC_DIVIDE

(Divide instruction exception - Status: fully implemented)

If the divisor is zero, the mathematical result is undefined, and this exception is raised. The target register does not get overwritten, but most of the time, it won't matter, because the program will get terminated by the OS anyway.

The exception is raised one cycle later than most other exceptions, because the 2nd operand has to be fetched from the register file first. If the 2nd operand is provided by the immediate field, it would be possible to raise the exception one clock cycle earlier, but I didn't integrate this small optimization, because it is simply not very important if a faulting program crashes some nanoseconds earlier. Exceptions should be exceptional after all, not too common, so they don't have to be as fast a humanly possible.

7.3.6 0x14 - EXC_TRAP

(Trap instruction exception - Status: fully implemented)

The only exception that does not signify an error, instead it is normally raised when the user mode program wants to give control to the OS, most of the time transporting some information in registers specified by the OS designers. From the hardware perspective, this is the most simple exception. The lowest 26 of the instruction have to be ignored by the hardware (per definition), because that way it's possible to transport some information to the trap handler code (type of trap or anything else). The assembler accepts an optional constant up to 26 bits for the trap instr.

7.3.7 0x13 - EXC_PRV_INSTRCT

(Privileged instruction exception - Status: fully implemented)

As explained, some instructions are privileged to code running in kernel mode to guarantee the OS's control over all user mode processes and the system as as whole.

7.3.8 0x12 - EXC_ILL_INSTRCT

(Illegal instruction exception - Status: fully implemented)

'Illegal' simply states that the instruction word has been read but is invalid, either because the opcode is unused, or because the instruction is `mvfs` or `mvts` and the special register is not in the valid range (0..4 inclusive).

7.3.9 0x11 - EXC_BUS_TIMEOUT

(Bus timeout exception - Status: fully implemented)

When there's no device to communicate with on the given address, the EXC_-BUS_TIMEOUT is raised. In the current implementation, the bus controller knows which address ranges are being used, so the exception gets raised immediately (contrary to the name).

7.3.10 0x10 - EXC_BUS_ADDRESS

(Bus address exception - Status: fully implemented)

The hardware enforces a word alignment for word reads/writes, likewise halfwords have to be halfword-aligned. If the address does not meet these requirements, EXC_BUS_ADDRESS is raised, and the read/write is not carried out.

Some instructions can occur in two different stages of execution, for example, a EXC_BUS_TIMEOUT could occur during instruction fetch, or during the execution of a load/store instruction. I can be difficult (sometimes even impossible) for the handler code to differentiate between the two, but so far no situation is known where it would be relevant for the handler to have this information. In case it becomes significant in the future, an additional field could be added to the PSW that specifies whether the exception was caused by a instruction fetch, or later (during instr. execution).

When adding exception support to an implementation, one has to make sure that instructions that cause an exception should not otherwise change the state of the machine. If, for example, the `mvts` instruction fails because the processor is currently in user mode, clearly the special register mustn't change. This is easy to archive most of the time, but it's still important to test the implementation before relying on it's correctness.

7.4 Interrupts

From the 16 interrupts that can be present, only one is actually defined and implemented so far. Other interrupts that probably will get added in the future include a disk interrupt and interrupts for terminals (two per terminal – receive and transmit).

The CPU contains a 16 bit input from the bus, one connection for each interrupt. The input has to be high for only one clock cycle, because the CPU contains an irqPnd (pending IRQs) register that saves the request, so it can be processed as soon as there are no more higher prioritized IRQs. If the interrupt could become irrelevant in the meantime, the CPU would have to request somehow if it is still relevant, but so far no interrupts are known that would become irrelevant without having been processed.

Much attention to details has to be applied by the author(s) of the interrupt handler code, because there's one very important, yet not apparent catch: The code that was running when the interrupt got active is to be resumed later,[1] so the handler has to save any register it intends to use, using `stw`.

Many assemblers reserve a register for internal usage, to give the illusion of word-sized immediate values (that the hardware does not provide) or for convenient assembler instructions that the hardware also does not provide. When using the existing assembler, register $1 is used for this purpose. But if a register is reserved for the

[1] This also holds true for (at least) one exception, EXC_TLB_MISS

assembler, that register has to be saved, too, because the handler, like any code, can contain instructions that use the “assembler only” register. The best approach is to save the “assembler only” register first, using code that is enclosed by

```
.nosyn   ; don’t automatically use "assembler only" register
...      ; save "assembler only" register
.syn     ; proceed normally
```

The .nosyn directive prevents the assembler from generating immediate statements that would normally be used to construct the specified immediate values if necessary, but that would be fatal as long as the “assembler only” register is not saved. After being saved, the register can be used as usual, hence the “.syn”. Likewise, before returning, the “assembler only” register should be restored last, also preceded by the .nosyn directive.

7.4.1 0x0E - IRQ_TIMER

(Status: fully implemented)

The only interrupt source so far is a timer, which is fairly simple, yet important for multitasking. It generates an interrupt after n clock ticks, and is configured by the OS, by writing the desired value for n into memory mapped location 0x30000000. When this location is overwritten, the timer is restarted, so the next interrupt will occur n clock ticks later.

When running a multitasked OS, the timer guarantees that the OS will regain control periodically, which can be used to do context switches whenever they are due.

The I/O-mapped memory location for the timer will be write protected while in user mode, because of being important for multitasking. This is possible as soon as the TLB is finished (that will contain a write protected flag for every page). Accountable for protecting the timer clearly is the OS, the hardware just has to provide the means to protect sensitive memory areas from unwanted access.

7.5 Special registers

The special registers reside inside a register file very similar to the regular registers, but with one difference: The PSW (s.r. 0) is constantly needed by the control unit, so

the second data output port of this register file constantly outputs the contents of s.r. 0. If the current design is a wise decision or difficult to maintain when later adding the TLB remains to be seen, but currently there are no known problems with it.

Probably because of the number of special registers (not a power of two), the Xilinx-tools do not infer a BlockRAM, generating 160 flipflops instead, but this could be circumvented -if needed- by manually instantiating a BlockRAM with a sufficient size.

7.6 Move to/from special registers

Due to the way the special registers are implemented, moving values between the regular and special registers is relatively simple, but the instructions have to act according to these rules:

- $0 does never change it's value, `mvfs` has to obey to this rule, too. This is done by OR-ing the bits of the target register number, and AND-ing with the write enable signal from the control. Because this is built into the register file, no special precautions were needed for the `mvts`/`mvfs` instrs.
- `mvts` is a privileged instruction. The target register mustn't change if/when the instruction is not allowed.
- if the number of the special register is not valid, the instruction has to considered illegal. The number of special registers present is fixed, so this can be checked directly by the control, without help from the special registers fie.

7.7 `trap` instruction

The `trap` instruction is very simple, one only has to keep in mind that the 26 lowest bits of the instruction word can contain any constant that the hardware has to ignore. That way, the handler can load the instruction word and extract the constant by simply executing

```
ldw $8, $30, 0
lhdi $9, 0xB8000000
and $8, $8, $9
```

Now the constant is found in $8 ($8 and $9 are arbitrary, $30 contains the interrupt return address which is in this case the address the `trap` instruction had).

7.8 Returning from exceptions

When executing the `rfx` (return from exception) instruction, the processor has first to check the user mode flag, and raise the exception EXC_PRIV_INSTR if it is set. If all is well, the current user mode and interrupt enable flags have to be restored, along with the previous ones. The PC is overwritten with the contents of register $30.

7.9 Additions to the overall implementation

Generally, it is beneficial for performance to raise exceptions as soon as possible, instead of wasting time executing instructions partially, which can be avoided most of the time.

The first step of each cycle of our multicycled CPU is the instruction fetch, which is also the first action that can raise an exception, because the instruction word has to be read from memory.

If the fetch was successful, the instruction is decoded and throughly examined. In this stage, many different exceptions can get triggered, preventing the instruction from being executed altogether.

In case the instruction was found to be faultless, the execution starts. Only some of the instructions can cause an exception once the execution has started, examples being the division/remainder instructions and the load/store instructions. Others, such as the other arithmetic/logic instructions, never fail (as long as the hardware works correct).

Register $0 can not be overwritten, but instructions writing to it are executed exactly like any other, to ensure that every necessary exception does get triggered as usual. It is extremely unlikely that this leads to any performance issues, because the software normally shouldn't contain instructions that have no effect, apart from the "nop".

Chapter 8

Notes on the Translation Lookaside Buffer

The TLB may not yet be present, but the specification is more-or-less complete, so I will describe it shortly. But be warned that it's not unlikely that details will change (for example, if cases are found that are not specified yet, or that turn out as being suboptimal).

Every of the 32 entries of the TLB maps (translates) the highest 20 bits of a virtual address to the corresponding physical address, the lowest 12 bits are simply taken across unchanged. Every entry can be valid or invalid, so this information has to be preserved and evaluated by the hardware. One idea is simply to use the fact that virtual addresses with the highest two bits set are directly mapped to hardware addresses, not needing the TLB translation. Therefore, the TLB should never contain entries where the highest two bits of the virtual address are set, making entries with this characteristic **invalid**. That way, no additional "valid"-flag has to be stored. And because addresses with the highest two bits set will never get translated by the TLB (they are directly mapped without using the TLB), no special provisions will have to be present.

So, an additional valid-flag is not necessary, but nevertheless, it can be beneficial, so probably, it will get integrated. The idea behind this is that the handler that gets active on a TLB miss has to be as fast as humanly possible, to quote "See MIPS Run":

> Valid Bit (V): If this is 0, the entry is unusable. This seems pretty pointless: Why have a record loaded into the TLB if you don't want the translation to work? It's because the software routine that refills the TLB is optimized for speed and doesn't want to check for special cases. When some further

processing is needed before a program can use a page referred to by the memory-held table, the memory-held entry can be left marked invalid. After TLB refill, this will cause a different kind of trap, invoking special processing without having to put a test in every software refill event.
Dominic Sweetman: See MIPS Run [SMR]

If a context switch occurs, all entries have to be invalidated at once ("flush"). It would be possible to provide a mechanism to do this in hardware, but probably, not much would be gained, because a context switch is slow anyhow, with the flush **not** being the most time consuming part, so this will probably be omitted.

The TLB will be used on every bus access, so, if possible, the TLB should be fully associative, in other words, every entry should be searched in parallel (this can't be simulated in software, it can only be done when designing hardware). Searching in parallel is one thing, but when the right entry is found, it has to find it's way to the memory address register as quickly as possible, but only if exactly **one** entry is found, otherwise the correct exception has to be raised.

On some versions of the MIPS hardware, a doublehit (two entries are found for one request) can be fatal, it can actually damage the hardware! Clearly this has to be avoided in our design. What makes matters worse is that it is not yet completely clear (to us), **how** a hardware damage is possible at all, one explanation could be: The register containing the found entry could drive the output lines of the unit, the others could be set to high-z ('z'). This is perfectly fine if exactly one entry is found, and it doesn't create problems when a TLB_MISS occurs (because the output is not driven, but also not used), but if two registers try to drive an output line simultaneously, a short-circuit current could flow, damaging the hardware. When using the Xilinx-Tools, they should be able to detect this, preventing the design from being synthesized at all, but I wouldn't guarantee that.

The entries are not completely equal, some of them (4 in our current design) are "static", which simply means that they don't ever get changed by a write random (TBWR), only when writing a specified index. They can be used by the OS in any way it sees fit.

There's an instruction to replace a random entry (TBWR), but the "random" index doesn't have to be truly random[1], actually any entry can be chosen. One simple implementation could simply increase a counter on every clock cycle, wrapping around after reaching 31. But one thing should be kept im mind: When doing many random

[1]That would make a hardware random number generator necessary, bloating the design

writes, every non-static entry should be replaced eventually. If -for example- only the entries with an even index get replaced, the "randomness" has to be improved somehow, ideally reaching equal probability for every index.

When an index is specified by the software, of course the specified index could be out of range. It would be possible to raise an exception in this case, but it would be easier (and probably sufficient) to just use the lowest 5 bits[2] of the provided index (making the resulting index valid in any case).

[2]Or, generally speaking, $log_2(n)$ bits, n being the number of entries of the TLB (which should be a power of two).

Chapter 9

Future possibilities

The current version is near to completion, only the TLB is missing, along with the four instructions dealing with it (`tbs`, `tbwr`, `tbri`, `tbwi`), but still a plethora of possible future experiments and improvements remains. For example, the following could be interesting, some of them additions to the architecture, some of them alternative ways of implementing it.

But of course, one shouldn't underestimate the amount of time that can be required to take even a well defined idea to practice (working code, correct formal hardware definition, etc.). Specifying a new entity, and getting all the details right is also quite a bit of work at times.

9.1 Embedded systems

The full ECO32 can, like the lightweight ECO32e, be used as the CPU core of an embedded system. The bigger ECO32 uses a bit more space inside an FPGA, but depending on the rest of the system, this may not be a problem at all.

9.2 Atomic swap instruction

An atomic swap instruction, that simply swaps two registers, could be useful. The Xilinx tools synthesize a BlockRAM for the regular registers, and this BlockRAM is capable of reading or writing two registers simultaneously, making a swap instruction really fast.

On the other hand, it's questionable whether much could be gained from such an instruction, because right now, a swap can be done in three instructions (without using a third register), so the speed gain shouldn't be overestimated.

9.3 Caches

The current version lacks caches (aside from the already planned TLB), so a future project could be the research/design/implementation of caches, for example an instruction cache. The RAM and FlashROM on the prototyping board are fast (relative to the FPGA), but accessing an arbitrary location still takes some wait cycles. There's BlockRAM as well as distributed RAM present inside the FPGA, enough for a small cache (or several small caches), and both is fast enough to be accessed in one clock cycle.

9.4 Floating point arithmetics

Floating point handling is not yet designed, but there are two separate concepts that one could give a try, either a software implementation or a hardware implementation. The software for the former could probably be taken from an open source project, and being adjusted for the peculiarities of the ECO ISA. A hardware implementation could be quite a bit of work, but of course it would be much faster. The space on the FPGA should allow a full-fledged hardware solution without problems.

9.5 Pipelined implementation

The current implementation is multi-cycled, with the already stated benefits, but a pipelined implementation would both be faster and closer to commercially available processors, making it an attractive future undertaking.

9.6 Specialized instructions

The currently used FPGA has a large number of slices to offer, with the current implementation using up only a certain amount of them, so there most probably would be

enough room for specialized instructions that provide time-consuming calculations in hardware, possibly on several data words in parallel.

Of course research would have to be done first, to find out which instructions would benefit from a direct hardware-implementation most, to not clutter up the hardware with superfluous structure. The ECO is a RISC processor after all, not a CISC processor.

9.7 Multiple cores

If the FPGA is large enough, one could try to design a multicore architecture, featuring several ECO32 cores on one FPGA, researching the opportunities and obstacles that arise. The semantics of interrupts and exceptions would have to be remodeled, but probably most of the design wouldn't have to be changed much.

Chapter 10

Glossary

Caution: Sometimes the same terms are used with differing semantic when comparing literature, so here's the semantic I had in mind.

ANSI / ANSI C: "ANSI" stands for "American National Standards Institute", a standards body that has released the official documents defining the syntax and semantics of the C programming language. C was designed by Brian W. Kernighan and Dennis M. Ritchie, that also published "The C Programming Language", describing every detail of the (then new) language, but later, subsequent versions of the language standard were developed by the ANSI. ANSI C has several advantages in comparison to the older K&R C, which can be found in detail in [NU04].

ansify, to: To 'ansify' means to translate C source code written in ancient K&R C (Kernighan & Ritchie C) to new, standard compliant ANSI C. The resulting source code than can be compiled by modern, ANSI-compliant compilers like gcc and lcc. The process can't always be automated completely, but software tools can help quite much.

Bitstream: In this document, the term "bitstream" always refers to the binary configuration data that is loaded into an FPGA, letting it perform the desired functions. A bitstream can be generated automatically from HDL source files by software tools.

BlockRAM: In addition to the "normal" slices, many FPGAs contain static RAM cells, called "BlockRAM", inside their fabric. BlockRAM has the advantage of being much faster than external RAM modules, not only because it is based on

static RAM (in contrast to external **dynamic** RAM modules), but also because it is distributed inside the FPGA, so pathways are extremely short (and thus, fast).

Exception: Internal cause for a switch to kernel mode and a jump to the IRQ handler. The IRQ handler has to decide whether the currently running instruction (if any) should be reexecuted, skipped, or the running process has to be terminated.

gcc / g++: "gcc" stands for "GNU C compiler", "g++" for "GNU C++ compiler", used to compile ANSI C code (respectively C++ code) to machine code for many different architectures and operating systems. Two of the most indispensable pieces of software on Linux, because they are used for the kernel and much of the available software. Please note: The abbreviation "GCC" (in uppercase) can also stand for "GNU compiler collection", consisting of compilers for C, C++, Java, Fortran, etc.

HDL: Hardware description language. Examples are **Verilog** and **VHDL**.

Interrupt: External cause for a switch to kernel mode and a jump to the IRQ handler. An interrupt is not triggered by the currently running instruction, so the instruction should always be reexecuted when the IRQ handler is finished.

IRQ: interrupt or exception. The term "IRQ" is used when no verbal differentiation between interrupts and exceptions is desired.

IRQ handler: A piece of machine code that is provided by the operating system. It gets executed whenever a IRQ is processed by the machine. The OS has full control over the actions taken to remedy the issue. Of course the IRQ handler should be very efficient as it will be run quite often. Most of the time, it will be programmed directly in assembler code, for greater control over all details. It is very OS dependant, also very CPU dependant, and thus not very portable.

ISA: Here, "ISA" stands for "Instruction Set Architecture", not to be confused with the term "Industry Standard Architecture"[1]. The Instruction Set Architecture describes the details of the processor that are important for assembler language programming, for example the registers, native data types, memory architecture, etc.

[1] A bus architecture on ancient PC mainboards.

The ISA can be seen as a definition for an interface between software and hardware, how it is implemented by the hardware is hidden from the software, and how it is used by the software is insignificant for the hardware.

Binaries for a specific ISA should be able to run on any hardware implementing exactly this ISA.

lcc: ("Local C Compiler" or "Little C Compiler") - A retargetable C compiler. This compiler is optimized to be easily adaptable to many different architectures. An ECO32 specific backend was created by Mr. Geisse, making it possible to compile ANSI C source code for our processor.

Priority: The IRQs are sorted by priority, IRQs with a higher number take precedence over ones with a lower number. Possible priorities range from 0..31, with the exceptions taking the numbers 16..31, so they are always higher prioritized than the interrupts (that range from 0..15).

Signed / unsigned: The regular registers take 32 bit quantities without differentiating between the bits, so it is up to the programmers to choose the right instructions (signed vs. unsigned) to operate on the values. When using the signed instructions, negative numbers are stored in twos-complement (as most hardware does).

Sign extended / zero extended: When a value that is n bits wide has to be represented in a register that is $n + k$ bits wide ($n, k > 0$), the value has to be sign extended if it is a signed number (that can contain negative values that are stored two's complemented), and zero extended if it is unsigned (can only be ≥ 0). Sign extending can be done by simply copying the sign bit (highest bit of the original value) to all additional bits, zero extending by filling all additional bits with '0's. For example, 0xFFFF would yield 0xFFFFFFFF when sign extended, 0x0000FFFF when zero extended (from 16 to 32 bits each).

Slice: An FPGA is composed of many (up to many thousands) of identical slices (also called "Logic Cells"), that each contains one ore more CLBs (the FPGA we used for our project contains 17280 slices with two CLBs per slice). Each CLB (Configurable Logic Block) can be programmed to perform a simple logic operation like `and`, `or`, `not`, `xor`, etc. The more complex a design, the more slices are used.

V-Bit: The V-Bit (for "Vector-Bit") selects the address for the IRQ handler. When the bit is cleared, ROM_BASE+4 is used, when set, the address RAM_BASE+4 is used instead.

Word/Halfword/Byte: A word is a 32 bit value, a halfword contains 16 bits, a byte is made of 8 bits. On other architectures, the widths could vary, but when referring to the ECO, they are defined to this values.

Chapter 11

Literature

COD2e: David A. Patterson and John L. Hennessy: **Computer organization & design: the hardware/software interface** (2nd edition), Morgan Kaufmann 1998
ISBN: 1-55860-491-X
Homepage: http://www.mkp.com/cod2e.htm

SMR: Dominic Sweetman: **See MIPS Run**, Morgan Kaufmann 1999
ISBN: 1-55860-410-3

KRC: Brian W. Kernighan and Dennis M. Ritchie: **The C Programming Language** (2nd edition), Prentice Hall, 1988
ISBN: 0131103628

ECO32 project: http://homepages.fh-giessen.de/ hg53/eco32/

ECO32e project: http://homepages.fh-giessen.de/ hg53/eco32e/

HES06 course: http://homepages.fh-giessen.de/ hg53/hes-ss06/index.html

XSA-3S1000 prototyping board: http://www.xess.com/prod035.php3

JW LaTeX: Jon Warbrick: **Essential LaTeX ++**, 1994
Download: http://noodle.med.yale.edu/latex/essential.pdf

WP ISA: http://en.wikipedia.org/wiki/Instruction_set

WP RC: http://en.wikipedia.org/wiki/Reconfigurable_computing

WP FPGA: http://en.wikipedia.org/wiki/FPGA

WP CPLD: http://en.wikipedia.org/wiki/CPLD

The following three documents are previous ECO32-related diploma theses that can be obtained from Mr. Geisse. They will also be included into future versions of the ECO32 package, to be obtained from the project homepage.

DK03: Dennis Kuhn: **Porting the Unix v7 kernel to the RISC Processor Eco32**, 2003

FG03: Felix Grützmacher: **Design and Implementation of a C Standard Library as Foundation for Porting System Software**, 2003

NU04: Norman Ulbrich: **Portierung von Unix Version 7-Kommandos auf den RISC-Prozessor ECO32**, 2004

List of Figures

Chapter 12

Declaration of academic honesty

12.1 German original

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. [1]

12.2 English translation

Hereby I declare that

- I have written this diploma thesis by myself.
- I marked the adoption of quotes out of literature as well as the use of thoughts from other authors in the corresponding paragraphs of the work.
- I did not present my diploma thesis for any other examination.

I am aware of the fact that a wrong declaration will have legal consequences. [2]

12.3 Signature

Hüttenberg, July 2006, Rolf H. Viehmann

[1] Source of the declaration: http://www.fh-giessen.de/fachbereich/mni/eidesstatt.shtml
[2] Source of the declaration: http://dict.leo.org/archiv.ende/2005_03/11/20050311104319e_en.html