DIPLOMARBEIT

A Superscalar 16 Bit Microcontroller for Real-Time Applications

ausgeführt am Institut für

Technische Informatik, Embedded Computing Systems Group
Technische Universität Wien

unter der Anleitung von
a.o.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
und
Univ.Ass. Dipl.-Ing. Martin Delvai

durch

Gottfried Fuchs
Canavesegasse 14
1230 Wien

Wien, 9. Dezember 2003 _____

Für meine Eltern

Danksagung

Für die fachliche und persönliche Unterstützung möchte ich folgenden Personen danken, die zum Gelingen dieser Diplomarbeit beigetragen haben: Martin Delvai, Thomas Handl, Wolfgang Huber, Peter Tummeltshammer und Angela Schörgendorfer

Besonderer Dank gebührt meinen Eltern und Großeltern, die mir dieses Studium durch ihre finanzielle, vor allem aber durch ihre seelische Unterstützung ermöglicht haben.

Kurzfassung

Im Zuge dieser Diplomarbeit wurde LANCE, ein superskalarer 16 Bit Mikrocontroller für Echtzeitanwendungen, entwickelt. LANCE ist Teil eines Baukastensystems für Mikrocontroller und basiert auf dem SPEAR Prozessor (Scalable Processor for Embedded Applications in Real-Time Environments), welcher am *Institut für Technische Informatik - Embedded Computing Systems Group* an der *Technischen Universität Wien* entwickelt wurde. Das Baukastensystem besteht aus mehreren Prozessorkernen, an die über eine generische Schnittstelle eine Reihe von Extension Modulen angeschlossen werden kann. Die Extension Module werden verwendet um den Prozessor an anwendungsspezifische Aufgaben anzupassen. Extension Module, welche für einen bestimmten Prozessorkern entwickelt wurden, können aufgrund der standardisierten Schnittstelle ohne jegliche Änderung von allen anderen Prozessoren verwendet werden. Die grundsätzliche Idee hinter dem LANCE-Entwurf war, einen Prozessorkern mit deutlich höherer Leistung als SPEAR zu entwickeln, ohne dabei die Code-Kompatibilität zu verlieren. Des Weiteren muss das Zeitverhalten von LANCE (wie auch von SPEAR) im Detail vorhersagbar sein, damit der Entwurf von Echtzeit-Systemen erleichtert wird. Um die zuvor genannten Anforderungen zu erfüllen, musste beträchtlicher Aufwand investiert werden um den Befehlsspeicher und das Register File dem Superskalar-Entwurf anzupassen. Aufgrund der Tatsache, dass LANCE zwei Befehle parallel ausführt, hat sich die Anzahl der Speicherzugriffe im Vergleich zu SPEAR verdoppelt. Die Notwendigkeit für massives Daten-forwarding zur Aufrechterhaltung der Code-Kompatibilität zu SPEAR, wie auch die erwähnte Verdopplung der Speicherzugriffe pro Taktzyklus, waren die größten Herausforderungen während der Entwicklung von LANCE. Ein weiteres Problem entstand aus der bereits definierten Extension Module Schnittstelle, welche nur einen Modulzugriff pro Taktzyklus zulässt. Die oben genannten Probleme konnten durch sorgfältigte Abstimmung der parallelen Pipelines gelöst werden, sodass schließlich ein funktionsfähiger Prototyp vorliegt.

Abstract

In the course of this diploma thesis LANCE, a superscalar 16 bit microcontroller for real-time applications, has been developed. The LANCE design is part of a modular construction system for real-time applications and based on the SPEAR processor (Scalable Processor for Embedded Applications in Real-Time Environments), which has been developed at the *Institute for Computer Engineering - Embedded Computing Systems Group* at the *Vienna University of Technology*. The modular construction system consists of several processor cores, a set of different so-called extension modules and a generic interface between these two types of components. The extension modules are used to adapt the processor core to different requirements imposed by a specific application. An extension module developed for one processor core can be used without any modification in all the others due to the standardized interface.

The basic idea behind the LANCE design was to design a processor core with significantly higher processing power than SPEAR without losing code compatibility. Furthermore, LANCE has to be temporally predictable like SPEAR to offer enhanced support of embedded real-time system design. To satisfy the previously mentioned requirements, considerable effort had to be invested to fit the instruction memory and register file to the superscalar design approach. Due to the fact that two instructions are executed in parallel, the memory access rate has doubled compared to SPEAR. The need for massive data forwarding to achieve code compatibility and the increased memory accesses per clock cycle were topics of great concern during the design of LANCE. Moreover, the extension module access implicated further problems due to the already defined module interface which only supports one access per clock cycle. The above introduced problems have been resolved by carefully tuning the parallel pipelines, which finally lead to a fully operative prototype.

Contents

List of Figures

List of Tables

1 Introduction

Embedded systems are used in a wide range of devices in everyday life such as refrigerators, microwave ovens, TVs, VCRs, DVD-players, printers, cameras, automotive control equipment (e.g. climate control, ABS, engine control, etc.), aircrafts and many more. There are many definitions of embedded systems, but all of them can be combined into a single concept: hardware and software, which are expected to function without human intervention and together form a component of some larger system [11].

1.1 Motivation

Why develop another 16-bit microcontroller? The ongoing miniaturization progress of electronic and electro-mechanic components will provide even more possible application areas for embedded computer systems in the future. An example application which is becoming more and more important these days are embedded networks. Usually embedded networks comprise a lot of different sensors to receive information from the environment, a processing unit to treat this information and a certain number of actuators to interact with the environment. To simplify the system and to reduce costs, sensors and actuators are not connected directly to the processing unit, but networked by a so-called field-bus [14]. This implies that each sensor/actuator has to be equipped with a network interface that implements the communication protocol. Such an interface comprises a microcontroller and a communication unit. An additional benefit of this local intelligence is that the microcontroller can also be used to preprocess the data in order to reduce the traffic on the field-bus. Furthermore, failure detection mechanisms and recovery strategies can be implemented in each network node to improve safety and reliability of the entire network. Due to the fact that such networks contain various nodes with different requirements in terms of processing speed, memory size and interfaces, different standard microcon-

trollers have to be used within the same network. This implies that the communication protocol, which is the same for all nodes [18], has to be implemented and tested for each microcontroller separately. This makes the development process costly and error-prone. Therefore it would be desirable to use the same microcontroller for all network nodes including the processing unit, but this would result in a waste of resources (silicon area, power consumption, etc.). Hence an ideal development system for embedded networks should provide a microcontroller that features modularity and is scalable in terms of computational power and functionality. Another problem arises if an embedded system has to provide real-time functionality. In real-time computer systems correctness of the system behavior depends not only on the logical results of the computations, but also on the instant at which these results are produced [24], therefore response and delay time of the embedded system have to reside in guaranteed boundaries. Common off-the-shelf processors, which are optimized for average performance, make it difficult to calculate exact worst-case behavior of executed program code. Worst-case assumptions (only cache misses, hazards, mispredicted branches, etc.) made for such systems lead to unrealistic bad performance.

Based on the requirements mentioned above, a modular construction system for real-time applications [8][9] (shown in Figure 1) has been developed at our institute. All components of the modular construction system have been implemented as VHDL designs [3] and tested on an APEX-FPGA development board [1][13]. Scalability of the microcontroller with respect to computational power is achieved by providing different processor core implementations of the same instruction set. Extension modules can be used to customize the various processor cores to application-specific requirements (extension modules and their interface are shown in section 1.3). A central role in this concept plays the standardized interface between processor core and extension modules, which ensures that all extension modules can

be attached to any of the available cores. Three different code-compatible real-time processors are provided, while the design and implementation of the LANCE processor is the objective of this diploma thesis.

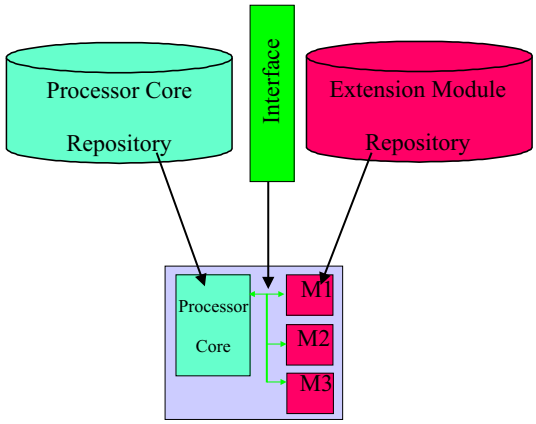

Figure 1: Modular Construction System

The LANCE design is intended to provide the modular construction system with more computational power than the two already existing processor cores NEEDLE and SPEAR are able to offer (NEEDLE and SPEAR are described in detail in section 1.2). The basic idea behind the LANCE design is to create a processor with twice as much processing power as SPEAR, without losing code compatibility and real-time capability. Performance gain shall not only be achieved through higher clock rates, but also by extending the SPEAR processor to a superscalar design. Independent from clock rate, the superscalar approach will lead to a major improvement of the instruction execution rate. The effects of both, increased clock rate and superscalar design, should provide LANCE with significantly higher overall performance than the SPEAR processor core.

1.2 Processor Cores

The two already existing, fully code-compatible 16-bit real-time processor cores SPEAR and NEEDLE will be presented in detail in this section.

1.2.1 SPEAR

SPEAR stands for "Scalable Processor for Embedded Applications in Real-time environments"[6][7]. The SPEAR design has been developed to provide moderate computational power and represents a RISC architecture which executes instructions through a three-stage-deep pipeline. The instruction set comprises 80 instructions which are described in detail in section 3.1 and appendix A. The main components of the processor are:

- Extended Register File
- Exception Vector Table
- Data Memory
- Instruction Cache
- Extension Modules
- ALU

Instruction and data memory are both 4 kB in size, but it is possible to add up to 128 kB of external instruction memory and 127 kB of additional data memory. The uppermost 1 kB of the data memory is reserved for memory mapping of the extension modules. As a result of the memory mapping, no dedicated instructions for extension module access are needed (common load/store instructions are used) which satisfies the RISC [28] philosophy of our approached design. The register file holds 32 registers which are split into 26 general purpose and 6 special function registers, three of which are used to construct stacks efficiently (see frame pointer operations in section 3.2.2) and

the remaining three are used to save the return address in case of an interrupt or subroutine call. SPEAR supports 32 exceptions, 16 of which are hardware exceptions (=interrupts) and 16 can be activated by software (=trap). The entries of the exception vector table hold the corresponding jump addresses to the interrupt/exception service routines for each interrupt or exception. The SPEAR ALU performs all provided arithmetic and logical functions, but is also responsible for offset calculation on jumps. Furthermore, the ALU is used to pass through data out of the exception vector table or register file. Figure 2 shows a block diagram of the SPEAR processor.

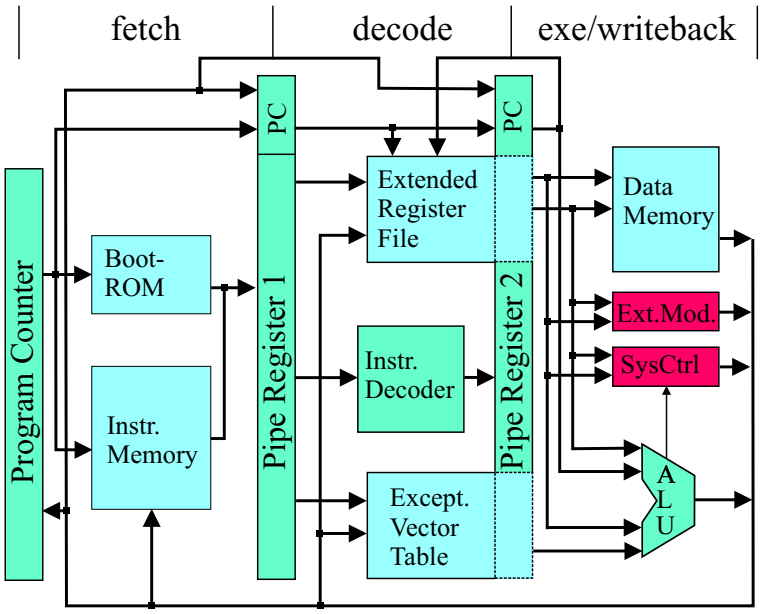

Figure 2: SPEAR Architecture

The SPEAR Pipeline shown in Figure 3 is structured into an: instruction fetch (FE), instruction decode (DE) and combined execute/writeback (EX/WB) stage. Inside the fetch cycle, instruction memory is accessed and one instruction opcode is passed to the decode stage. During the decode cycle the control signals for the memories and the ALU are generated, fur-

thermore the instruction's operands are retrieved from the register file. The execute/writeback stage performs the instruction's intended operation and writes the resulting value to the appropriate memory location. If an extension module access (EXT) happens, it is also executed during the EX/WB cycle.

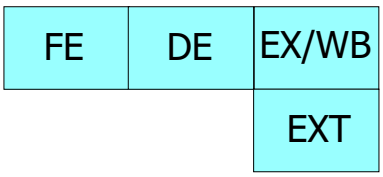

Figure 3: SPEAR Pipeline Architecture

1.2.2 NEEDLE

The NEEDLE processor core [12] is fully code-compatible to SPEAR and is intended to pay attention to a compact design, therefore it is primarily optimized to reduce silicon area. NEEDLE uses the same extension module interface as SPEAR does, and an identical Processor Control Unit (which is presented in chapter 1.3). NEEDLE has no pipeline and requires several clock cycles per instruction. To minimize chip area, instruction- and data memory, the register file and the exception vector table have been mapped into the same physical memory. This way only one address- and data bus is necessary to access all memory elements which leads to reduced silicon area. The physical memory block is 4 kB in size, of which 2 kB are used as instruction memory and 1.92 kB as data memory. The remainder is allocated to the register file and the exception vector table. Additionally the processor core comprises only an ALU, a multiplexor, three registers (program counter, instruction register, accumulator register) and a sequencer. The block diagram of NEEDLE is shown in Figure 4.
The instruction register (IR) is needed to store the currently processed instruction. The accumulator register (ACCU) is used to store one operand

for the ALU and the program counter (PC) contains the address of the next instruction. The most complex part of NEEDLE is the sequencer which has been implemented as eight-stage state machine. The sequencer's task is to distinguish between the 80 different instructions and generate the appropriate control signals for the ALU and the memory. Instructions are executed within three phases, the fetch, the decode and the execute phase. As the write-back to the memory runs parallel to the fetch phase, some of the instructions only need two instead of three clock cycles to be processed. The NEEDLE processor supports 32 exceptions as well, 16 of which are hardware exceptions (= interrupts) and 16 can be activated by software (= traps).

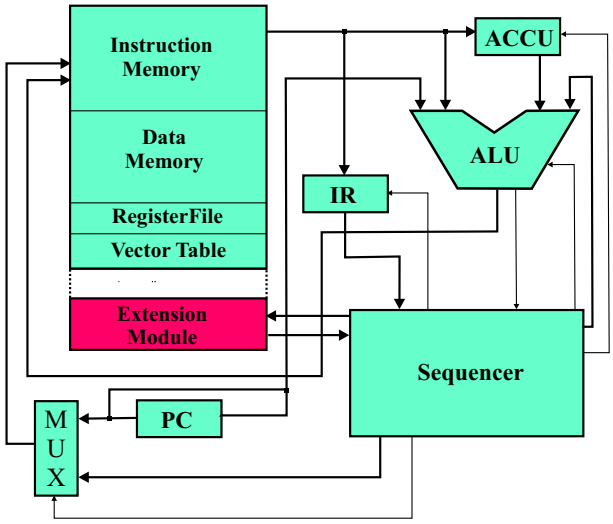

Figure 4: NEEDLE Architecture

1.3 Extension Modules

As mentioned in chapter 1.1, extension modules are used to fit the processor for different applications. For reason of simplicity and lucidity, the integration of and the access to extension modules should work the same way for all of them. Thus a generic interface for all extension modules has been defined [16]. All extension modules are mapped to a unique location at the upper most region of the data memory. The modules are accessed via eight registers using simple load and store instructions, since from the processors point of view the extension modules are simply memory locations. A block diagram of the generic extension module interface is shown in Figure 5. The first two registers are the module's *status* and *config* registers, of which the status register is **read only**. The lower 8 bit of these registers are defined within the interface specification, hence they are identical for all extension modules, whereas the upper 8 bit are module specific. The *status* register tells the processor the current state of the extension module. Among other things it shows if an interrupt has been activated, an error has occurred, or the extension module is still busy. The *config* register is used to specify parameters for the module's operations. Next to a softreset bit, which is used to deactivate the extension module, an interrupt acknowledge bit exists to reset the interrupt status. The remaining six registers Data 0 - Data 5 are allocated for module specific issues.

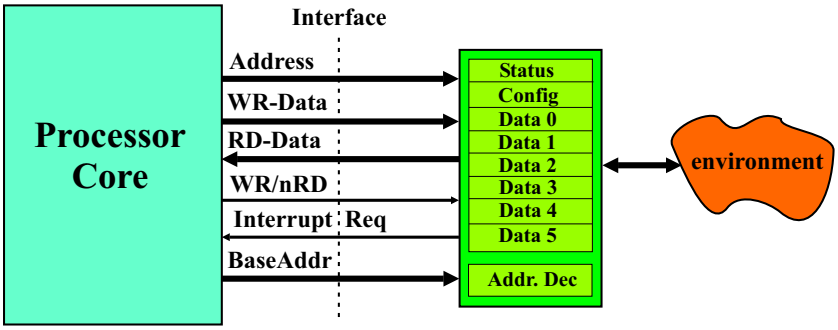

Figure 5: Generic Extension Module Interface

Dependening on their functionality there are three different types of extension modules:

- *Processor Control Modules*: These modules have direct influence on the processor core's behavior. An example for such a module is the "protection control module", which provides the processor core with the possibility to restrict memory access to specific locations. This is necessary if the processor runs an operating system.
- *Function Modules* are used to provide the processor core with hardware implementations of different functions like integer multiplication, floating point operations or even complex algorithms.
- *I/O Modules*: Since the processor core has no direct interface to the environment, all interaction is aided by I/O extension modules. Examples for I/O extension modules are: RS-232, USB, 7-segment display, VGA, PS/2, etc.

1.3.1 Processor Control Unit

The *processor control unit* plays a major role among the extension modules, because it contains the processor status register as well as the interrupt handler. As the status register and the interrupt handler are vital components of the processor core, the *processor control unit* can be accounted as proper component of the processor. The status register and the interrupt handler have been moved to an extension module due to compatibility issues. This way all processor cores use the same *processor control unit* and therefore it is guaranteed that the status register and the interrupt mechanism act exactly the same way in all processor cores. Since the processor would not work without the *processor control unit*, it is always mapped to the uppermost address of the data memory. As depicted in Figure 6, the *processor control unit* is subdivided into four parts:

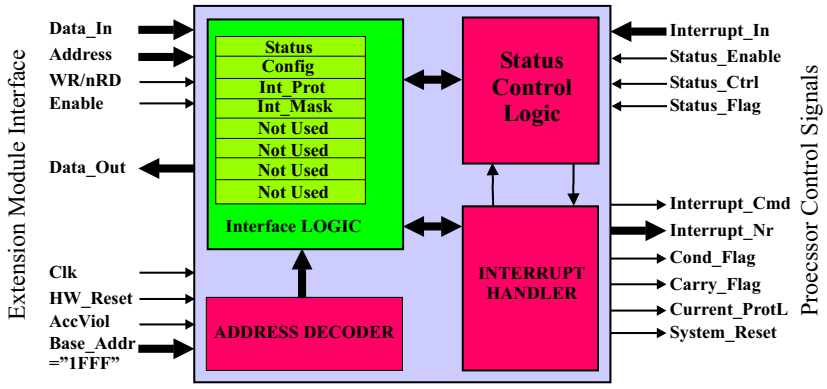

Figure 6: Processor Control Extension Module

- Generic Interface: The registers of the interface hold the status and config register as well as the interrupt protocol and interrupt mask register.
- Status Control Logic: The status control logic provides the processor with correct status values and is responsible for updating and saving the status register.
- Address Decoder: Each extension module is mapped to a unique memory location within the data memory. The address decoder recognizes if the modules is accessed and activates it.
- Interrupt Handler: The interrupt handler is responsible for synchronizing and logging incoming interrupts and provides the processor with the correct interrupt number.

Figure 7 depicts the register interface of the *processor control module*, the status register and essential parts of the config register will be illustrated below. The first two registers of the generic interface are the status and the config register (as described in the previous section). The module specific part of the status register holds the processor status register, and can only be

read by the processor. Besides common status flags (zero-, carry-, overflow- and negative flag) the processor status register holds, additional flags:

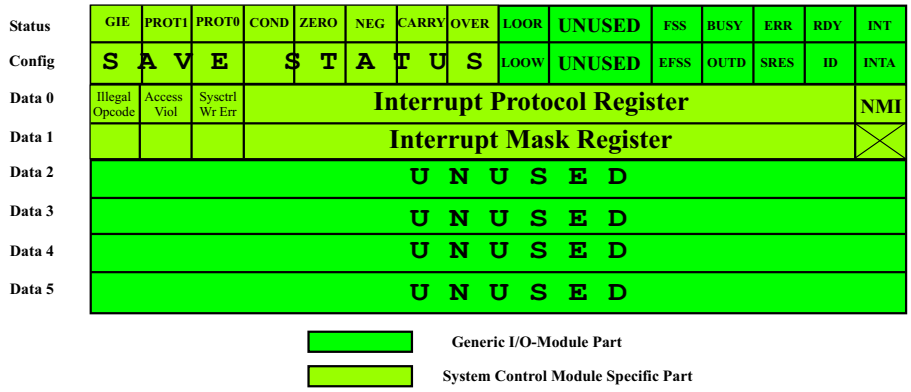

Figure 7: Register Interface of the Processor Control Extension Module

- **GIE**: If the global interrupt enable (GIE) is set (='1') all interrupts are accepted by the processor control unit; if GIE is deactivated the interrupts are only logged but not executed (with the exception of the non maskable interrupt NMI which is always executed and mapped to bit 0).

- **PROT1/PROT0**: Protection bits 1 / 0 show the protection level of the current processor task. "11"for zero protection, "10"for low protection, "01"for high protection and "00"for the supervisor mode (will be ignored if the protection control module has not been activated).

- **COND**: The condition flag is used to determine if conditional instructions will be executed or not. The cond flag is only set or deleted by compare or bit-test instructions.

- **ZERO**: The zero flag indicates that the result of the last ALU operation was zero.

- **NEG**: The negative flag is set if the result of the ALU operation is negative.
- **CARRY**: The carry bit is used for arithmetic instructions.
- **OVER**: The overflow bit indicates an overflow of a calculation.

The module specific part of the config register is used to save the processor status if an interrupt/exception occurs.
The interrupt protocol register is responsible for the logging of all incoming interrupts. All interrupts that are masked within the interrupt mask register (bit set at the corresponding position) will be logged, but the interrupt service routine will not be executed.

1.4 Program Download

To provide the user of the microprocessor with a versatile programming interface between PC and microprocessor the download hardware has been split into two parts, the communications unit (COM-Module) and the programmer module (PROG-Module) which includes a *Boot-ROM* for startup. This approach has the advantage that it is quite simple to change the download medium. Only the communications module and the corresponding initialization program code (situated inside the programmer's *Boot-ROM*) have to be replaced. The communications unit transports the downloaded program code into the processor, but the actual programming into the instruction memory is done by the programmer module. Figure 8 depicts dataflow during the programming procedure.

The programmer module's *Boot-ROM* is used to set up the communication module and initiates the data transfer from the communication unit to the programmer and the instruction memory. After the download is finished the programmer module switches from the *Boot-ROM* to the instruction memory and the processor starts to execute instructions out of the instruction memory.

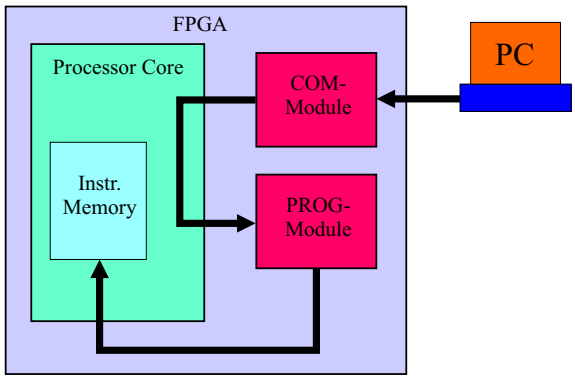

Figure 8: Program Download

1.5 Chapter Organization and Overview

Chapter two gives an overview on superscalar designs considering three different past and one state of the art processor designs.

In chapter three a detailed specification of the microcontroller is given, furthermore the instruction set is described in detail. This chapter ends with a survey of code restrictions that may arise due to the superscalar approach.

Chapter four treats the design of LANCE and gives deep insights into the design decisions on each component of the microprocessor.

Chapter five focuses on the hardware design flow and the development environment.

Chapter six presents the test environment and the results, and gives a comparison of NEEDLE, SPEAR and LANCE.

Chapter seven covers the conclusion of this diploma thesis.

In Chapter eight an outlook on topics for further research is given.

2 Superscalar Designs - State of the Art

In this chapter different superscalar designs are introduced to give an overview of how superscalar processors can be designed. The term "superscalar" is used for architectures which are able to execute two or more instructions in parallel. The Intel Pentium, Motorola 68060 and PowerPC 601 were chosen due to the fact that all three designs are the first superscalar approach in their product families and therefore less complex than currently available processors. Furthermore, a short overview on the Intel Pentium 4 architecture will be given to show one state of the art superscalar design.

All the presented superscalar designs are commercial off-the-shelf processors for use in workstations and personal computers. They are all optimized for best performance using standard application software. This optimization concept led to several architectural constructs like caches, dynamic branch prediction or out-of-order execution which improve average performance, but make it very difficult (in some cases impossible) to calculate exact WCET (**W**orst **C**ase **E**xecution **T**ime) boundaries for executed program code [38].

2.1 Intel Pentium

The first superscalar member of the Intel processor family for Personal Computers is the Intel Pentium® [19][20]. The processor core consists of two parallel pipelines, the u-pipe and the v-pipe, as shown in Figure 9. The u-pipe handles any kind of instruction, whereas the v-pipe only supports simple integer and simple floating-point instructions. Each pipeline is able to execute frequently used instructions in a single clock cycle. Together these two pipelines can issue two integer instructions or one complex floating-point instruction in one clock cycle. The pipes are organized as five-stage-deep pipelines. The pipeline stages are as follows:

- PF Prefetch

- D1 Instruction Decode
- D2 Address Generate
- EX Execute - ALU and Cache Access
- WB Writeback

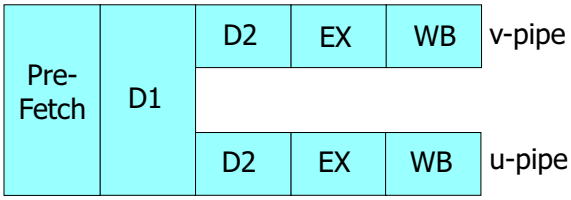

Figure 9: Intel Pentium Pipeline Architecture

The first pipeline stage is the Prefetch (PF) stage in which instructions are prefetched from the on-chip instruction cache. Because the processor has separate caches for data and instructions (Harvard-Architecture), prefetches do not conflict with data references. In the PF stage two independent pairs of prefetch buffers work in conjunction with the branch target buffer (BTB). One of these two buffers prefetches instructions sequentially while the other retrieves instructions according to the BTB predictions.
The pipeline stage after the PF stage is Decode1 (D1), in which two parallel decoders issue the next two sequential instructions. The decoders also determine whether the instructions can be paired (executed in parallel) or not. When instructions are paired, the instruction issued to the v-pipe is always the next sequential instruction after the one issued to the u-pipe. Instructions with data dependencies or complex instructions (which require both pipelines) are not allowed to be paired. Jumps can only be paired if they are issued to the v-pipe. More details on instruction pairing is provided in [20]. The D1 stage is followed by the Decode2 (D2) in which addresses of memory resident operands are calculated.

In the Pentium processor the Execute (EX) stage of the pipeline is used for both ALU operations and data cache access. In EX, all u-pipe and v-pipe instructions are also verified for correct branch prediction. If the BTB has mispredicted a branch, both pipelines have to be flushed and a new instruction stream will be fetched.
The final pipeline stage is Writeback (WB), in which the instructions are enabled to modify the processor state and complete execution.

During their progression through the pipeline, instructions may be stalled due to certain conditions. The u-pipe and the v-pipe instructions enter and leave D1 and D2 at the same time. If there is a stall in one pipeline, the instruction in the other pipeline is also stalled at the same pipeline stage. No instructions are allowed to enter EX stage of either pipeline until the instructions in both pipelines have advanced to WB. This concept of instruction issuing is know as *in-order execution*.

2.2 Motorola 68060

The Motorola 68060 [5][26][4] is the forth generation microprocessor of the M68000 Family and it is code compatible with previous family members. The 68060 employs a deep pipeline, dual-issue superscalar execution, a branch cache, a floating point unit and 8 kB each of on-chip instruction and data cache.
As shown in Figure 10, the 68060 consists of two distinctive parts: a four-stage Instruction Fetch Pipeline (IFP) for prefetching instructions and loading them into the FIFO instruction buffer, and dual four-stage operand execution pipelines (OEPs) which perform the actual instruction execution.
The four stages of the IFP are:

- IAG Instruction Address Generation
- IC Instruction Fetch Cycle
- IED Instruction Early Decode
- IB Instruction Buffer

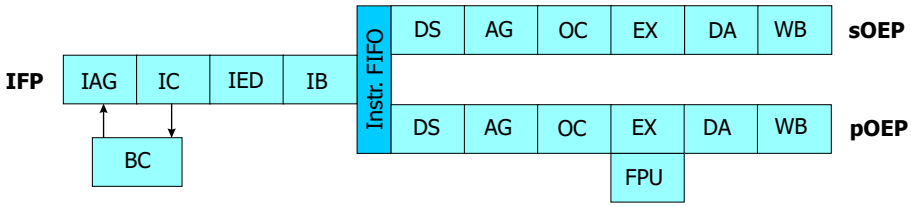

Figure 10: Motorola 68060 Pipeline Architecture

The first pipeline stage in the IFP is the Instruction Address Generation (IAG) which calculates the next prefetch address. The Branch Cache (BC), which improves prefetch efficiency by detecting changes in the sequential flow of the fetch stream based on past execution history, is also accessed in this pipeline stage.

After the IAG sends the correct address to the instruction cache, the Instruction Fetch Cycle (IC) stage performs the cache lookup and fetches the bit pattern of the next instruction.
The Instruction Early Decode (IED) stage implements a lookup table function to provide the OEPs with decode information concerning instruction resource requirements along with controlling information for the superscalar dispatch algorithm. The IED stage also converts the variable length instructions with multiple formats into a fixed-length extended operation word. At the end of the IFP the prefetched and converted instruction along with the extended operation word are issued to the Instruction Buffer (IB).
In the IB stage the instructions are read from the FIFO and loaded into the dual OEPs.
The Operation Execution Pipelines, known as the primary OEP (pOEP) and the secondary OEP (sOEP), are partitioned into a four-stage-deep pipeline. The pOEP supports all instructions, whereas the sOEP only executes a subset of the instruction set. The four OEP stages are:

- DS Decode and Select
- AG operand Address Generation
- OC Operand Cycle
- EX Execute cycle

For instructions writing data to memory, there are two additional pipeline stages:

- DA Data Available
- ST Store

The Decode and Select (DS) stage determines the next state for the entire operand pipeline and also selects the components required for operand address calculation. If multiple instructions can issue into the AG stage, the

first and the second instruction move into the respective AG stages. If only a single instruction can issue (because of architectural restrictions), the first instruction issues into pOEP and the second and third instruction will be used for pairing in the next cycle, which leads to a sliding two-instruction-window to examine possible pairs of instructions.
In the operand Address Generation (AG) stage each pipeline calculates the effective address for instructions requiring data from memory.
The Operand Cycle fetches register and memory operands.
Finally the Execute (EX) stage performs the desired instruction executions in the integer and floating point units including condition flag updating. The 68060 performs condition code checking inside the EX stage, which leads to a penalty of seven clock cycles if a branch has been mispredicted. To minimize this effect the microprocessor employs a sophisticated branch cache.
The two additional stages Data Available (DA) and Store (ST) are needed to complete operand store operations.

The Motorola 68060 issues all instructions according the to *in-order execution* concept. One special feature, implemented for solving data hazards, is register renaming. This renaming optimization, performed in the DS stage, substitutes internal pipeline register contents for general register contents. For more details, see [4][37].

2.3 PowerPC 601

The PowerPC 601 [35][25] combines a RISC architecture with a superscalar machine organization. The 601 contains a 32kB cache (combined instruction- and data cache) and is capable of dispatching, executing and completing up to three instructions per clock cycle. As shown in the block diagram in Figure 11, the 601 consists of three different execution pipelines, which are:

- BU Branch Unit
- FXU Fixed Point Unit
- FPU Floating Point Unit

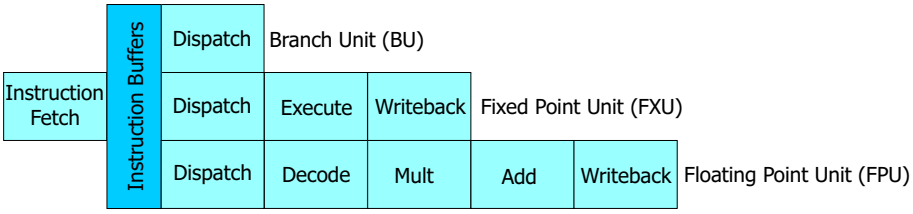

Figure 11: PowerPC 601 Pipeline Architecture

The Branch Unit (BU) uses a static branch prediction algorithm to predict the direction of unresolved branches and executes them. This prediction algorithm simply predicts a branch as *taken* if the displacement of the target address is negative, and as *not taken* if it is positive.
The Fixed Point Unit (FXU) serves as master pipeline and is used for all integer ALU operations. It also handles all processor Load and Store instructions (including the Floating Point Loads and Stores).
The Floating Point Unit (FPU) pipeline contains six stages and is able to execute all single precision instructions fully pipelined, only double precision multiplication and division are *double pumped* through the Mult and Add stages.
The different pipeline stages will now be described in more detail.

Inside the Fetch pipeline stage the unified cache is accessed, with instructions having lower priority than data. To provide the pipelines with a continuous stream of instructions, up to eight instructions are fetched into the instruction buffers in a single cycle, even though the absolute maximum processing rate is three instructions per clock cycle.
Instructions are dispatched within the Dispatch stage to the FXU and FPU. The Branch Unit (BU) also decodes, predicts and executes branches inside the Dispatch stage.
The Decode stage is responsible for decoding instructions, furthermore all source registers are read. Instructions going to the FXU are dispatched and decoded in this pipeline stage.
All fixed-point operations are executed inside the Execute stage, moreover in case of a load/store instruction the address processing and cache lookup take place.
The Mult pipe stage is responsible for floating point multiplications and hands its results over to the Add stage where the final floating point result is calculated.
The Writeback pipeline stage performs the register update, for the FPU pipeline it also takes care of result rounding and normalization.
Instructions may be dispatched out of order which makes it possible to get instructions out of the instruction buffers even while one pipeline is blocked. To keep track of the program order, a unique tagging and counting mechanism has been implemented. Although *out-of-order dispatch* is more complex to implement, it also allows to minimize potential dispatch stalls. Furthermore, the PowerPC 601 provides register renaming [37] (only FXU) and full data forwarding between pipeline stages to resolve as many data hazards as possible. If there are still any hazards left, they are automatically interlocked by hardware (stall cycles are inserted).

2.4 Intel Pentium 4

Intel's current flagship for desktop computers is the Intel Pentium 4 Processor. The Pentium 4 introduces a few new terms like the *NetBurst Micro-Architecture*, *Rapid Execution Engine*, *Hyper-Pipelined Technology* and *Hyper-Threading Technology*[15][23].
The *Hyper-Pipelined Technology* refers to the 20-stage pipeline of the NetBurst micro-architecture (shown in Figure 12). This pipeline is twice as long as its predecessor on the P6, and therefore provides Intel with the possibility of reaching much higher clock rates (if less work has to be done in each clock cycle, the cycle time can be shortened).

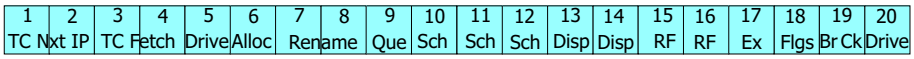

Figure 12: Pentium 4 Pipeline Architecture

The *NetBurst* instruction execution is broken down into three main parts: an *in-order* issue front end, an *out-of-order* superscalar execution core and an *in-order* retirement. The front end feeds a continuous stream of up to three micro-operations to the execution engine. The decode unit can decode one IA-32 [22] instruction per clock cycle and stores the resulting μops to the Execution Trace Cache. The trace cache can hold up to 12k μops, which for most of the time preserves the processor from the need of instruction decoding. The execution engine can have up to 126 instructions "in-flight"at once, 48 of which may be loads and 24 stores. The Pentium 4 core can dispatch up to 6 μops per cycle and uses register renaming to resolve as many data hazards as possible. Retirement in NetBurst reorders the instructions executed in an out-of-order manner to ensure that the system state is left as the programmer intended.
Rapid Execution Engine refers to the integer execution units of the Pentium 4. These low latency integer ALUs are able to perform fully dependent integer ALU operations at twice the main clock rate. Two instructions are

issued by calculating one during the first half of the clock cycle and the second during the second half.
Hyper-Threading Technology (HT) [21] stands for the ability of a processor to execute more than one thread at a time and to appear to the Operating System as two logical processors. HT has been developed to increase usual processor utilization (about 35%). By sending two threads into the processor at the same time, execution units that would otherwise have been idle can be used by the second thread, which can increase processor utilization by up to 50%. The trace cache is shared between the two threads, furthermore the trace cache and the retirement logic alternate between the threads and therefore ensure that both logical processors can progress on program execution.

In the case of a mispredicted branch the 20-stage-deep pipeline leads to average costs of 20 cycles, which makes branch prediction the single most important unit inside the Pentium 4 architecture (and therefore more advanced than in any other intel processor).
The NetBurst processor architecture was designed to provide a platform which desktop processors for the next few years will be based on. Intel claims that this 20-stage pipeline will allow them to reach clock frequencies of 10 GHz in the future without changing the micro-architecture.

3 Specification

This chapter gives a detailed specification of all requirements and constraints regarding the design of LANCE. The basic idea behind LANCE is to design a processor with twice as much processing power as SPEAR. A major requirement was to maintain code compatibility to SPEAR and NEEDLE as far as possible. This ensures that program code developed for one processor can be used without any modification on another one, which supports rapid prototyping concepts. From there, it is also necessary to implement exactly the same extension module interface.
Top priority issue during the design of LANCE has been to maintain real-time capability, therefore following requirements have to be met:

- predictable program execution time at an accuracy of one clock cycle to support WCET analysis
- deterministic interrupt response time
- support of priority levels to ensure immediate response on operating system tasks

The performance gain which is intended to be achieved should result out of two approaches, in the first place by increasing the clock rate due to architectural optimizations. The second approach will extend the SPEAR processor to a dual-issue superscalar design. The effects of both increased clock rate and superscalar design should provide the LANCE processor with significantly higher overall performance than SPEAR. Due to the fact that the generic extension module interface only supports one extension module access each clock cycle, special architectural mechanisms have to be integrated to ensure fully predictable timing behavior (extension module access integration is shown in detail in section 4.4.7).

3.1 Instruction Set

Following a RISC architecture concept, the instruction set of NEEDLE, SPEAR and LANCE comprises 80 different one-word instructions. The instructions can be separated into five distinctive groups:

- two-register-instructions
- one-register-instructions
- instructions using one constant
- register-constant-instructions
- instructions without arguments

two-register-instructions: these instructions have two register addresses as arguments and can be further subdivided into:

- load/store instructions where the first register is the operation's destination/source register and the second register holds the memory address that has to be accessed, for example *LDW r1, r2* : mem(r2)− >r1
- arithmetic operations like *ADD r1, r2; OR r1, r2; AND r1, r2* where the first and second register hold instruction source values, furthermore the first register serves as destination register.

one-register-instructions: only operate on the contents of one register. The result of the operation can either be written back to the same register e.g. *NOT r1; SL r1; RR r1; SRA r1;* or may be used as branch target e.g. *JMP r1; JSRY r1.*

instructions using one constant:

- *JMPI offset;*(jump immediate) performs a near jump to the instruction address calculated from the program counter and the given offset (PC + offset). The offset is a 10-bit signed integer and therefore provides a jump range of -512 and +511.

- *TRAP n4;* activates one of 16 possible software interrupts (trap) and saves the return address to register 31. *TRAP 5* for example leads to a jump to the address stored on position 5 inside the exception vector table (further details on interrupts, traps and the exception vector table are provided in section 3.2.5).

register-constant-instructions: have one register and one constant as parameters. These instructions can be subdivided into several instruction subclasses:

- *LDL r1, n8*; (load low) is used to initialize the lower 8 bit of register r1 with the binary representation of the constant n8. (the instruction performs sign-extension of two's complement numbers, this means that the constants MSB is copied to bit 9 to 15 of r1)

- *LDH r1, n8;* (load high) In contrast to LDL this instruction stores the 8-bit constant n8 to bit 9 to 15 of register r1 without affecting the lower 8 bit.

- *CMPI_EQ r1, n5* compares the register contents of r1 with the signed integer constant n5, if r1 and n5 are equal the condition flag is set.

- *ADDI r1, n5* adds register r1 and the signed integer constant n5 and writes the result back to register r1.

- *LDOFX, LDOFY, LDOFZ, STOFX, STOFY, STOFZ r1, n5;* load-with-offset and store-with-offset instructions are dedicated framepointer operations. The registers r26 (framepointer X) to r28 (framepointer Z) are coupled with the framepointer load and store instructions. For example, *STOFY r11, 7* stores the content of r11 at the memory location which is given by the sum of r27 and the signed integer value 7. (more details on framepointer operations are provided in section 3.2.2)

- *STVEC r1, n5; LDVEC r1, n5;* are required to store and read back the exception vector table entries. Register r1 holds the jump address for the exception/interrupt service routine and the 4-bit signed integer constant specifies the position load/store position at the vector table

- *BTEST r1, n4; BSET r1, n4; BCLR r1, n4;* the 4-bit unsigned integer constant specifies the bit position of register r1 on which a bit set, clear or test is performed. E.g. *BTEST r12, 7;* tests bit number 7 of register 12, if it is set to '1' the condition flag will be set.

instructions without argument: *NOP* (no operation), *ILLOP* (illegal opcode), *RTSX*, *RTSY* (return from subroutine) the return address is stored in register r29 respectively register r30 and *RETI* (return from interrupt) the return address is stored in register r31.

3.2 Instruction Set Features

Knowing the worst-case execution time (WCET) of programs is crucial for real-time systems. Safe WCET bounds for all time-critical tasks of a real-time system have to be established to verify the correct timing behavior of the whole real-time computer system. Analyzing execution paths as performed by static WCET analysis is supported by the instruction set due to constant instruction execution time of any single instruction. Furthermore, the instruction set supports the ONE-PATH programming paradigm [30] by providing constant-time conditional instructions like *MOV_CF* (move if condition is false). Using those conditional instructions there is the possibility to implement logical branches as strictly sequential program code.

3.2.1 Conditional Instructions

As mentioned above, constant-time conditional instructions are needed to support real-time system design, especially the ONE-PATH programming paradigm (32 out of 80 instructions are conditional). Conditional instructions like *MOV_CT r1, r2* evaluate the processors condition flag (set by a previous compare or bit-test instruction) to determine if the operation has to be executed or if a *NOP* is going to be executed. Inserting a NOP if the condition is false is necessary to provide data-independent constant execution time. The condition flag, which is located inside the processor status register, remains in its current state until the next compare or bit-test operation alters it. This approach simplifies execution of program blocks like:

```
if condition then
    expression 1;
    expression 2;
else
    expression 3;
end if;
```

```
evaluate condition
    expression1_CT;
    expression2_CT;
    expression3_CF;
end;
```

The implementation on the left side executes either *expression 1* and *expression 2* or *expression 3*, whereas the implementation on the right side which uses conditional instructions in any case performs all three expressions. Executing all instructions of the *if-then-else* block provides constant execution time regardless of the condition and does not necessarily lead to loss of performance. If there is a small amount of instructions within the *if-then-else* block, the implementation with conditional instructions will lead to equal or even better performance than the common implementation (left side). The common *if-then-else* block implementation typically consists of at least two jump instructions. One jump may take place after the condition has been evaluated and leads to *expression 3* (*else*-part), the other jump is performed after *expression 2* is executed and leads to the end of the *if-then-else* block. Each jump instruction implicates significant performance loss within pipelined processors, since the content of both pipelines has to be flushed (if the processor implements jump-prediction the amount of pipeline flushes can be minimized at the cost of more complex WCET analysis).

3.2.2 Framepointer Operations

Framepointer operations are used for efficiently building stacks inside the data memory (shown in Figure 13). In contrast to common stacks, data can be written to or read from any memory location within the frame (there is no need for emptying the whole stack to get data that has been pushed at the beginning). Framepointer operations are coupled with the registers r26, r27 and r28 (framepointer register X, Y and Z), hence three independent stacks can be built. To use framepointer operations, first of all a framepointer (e.g. framepointer X which corresponds to register r26) has to be set to the desired memory location to provide a base address for the stack. After that, a simple summation of the framepointer and the 5-bit offset performed by *LDOFX, LDOFY, LDOFZ* ("load with offset") or *STOFX, STOFY, STOFZ* ("store with offset") instructions lead to the intended stack operation.

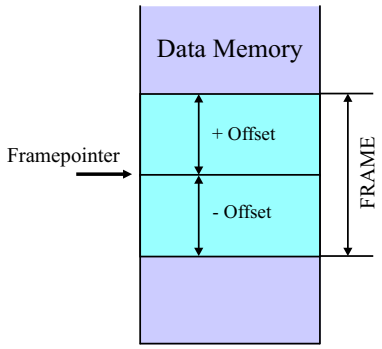

Figure 13: Framepointer Stack

3.2.3 Subroutine Calls

Two independent subroutine calls are supported by the instruction set. In case of a subroutine call the return address (address of the succeeding instruction of the subroutine call) is automatically stored to register r29 (subroutine X) or register r30 (subroutine Y) respectively, hence a subroutine nesting depth of two levels is provided. If further subroutine nesting is required, the programmer is responsible for saving and restoring the correct return address.

3.2.4 Immediate Instructions

There are three different types of immediate instructions to aid compact program code design:

- *CMPI_EQ r1, n5;* (compare immediate equal) The contents of register r1 and the sign extended 5-bit constant n5 are examined. If they are equal, the condition flag (inside the processor status register) is set.
- *ADDI r1, n5;* (add immediate) adds up the contents of register r1 and the signed extended 5-bit constant n5 (the carry flag will be ignored by this instruction).

- *JMPI offset;* (jump immediate offset) performs a near jump to the target address given by the signed 10-bit offset value. The signed 10-bit offset results in a jump range of -512 and +511 instructions.

3.2.5 Exceptions

Exceptions can be subdivided into two groups:

1. interrupts, which are generated by hardware
2. traps, which are activated by software via the *TRAP* instruction.

Apart from the way they are activated, interrupts and traps show the same behavior: the program status register is saved and the program counter is copied to register r31 to ensure that the actual processor state can be recovered after the service routine is finished.
Typically the exception vector table (shown in Figure 14) is initialized with the service routine address for each exception during program start-up using the *STVEC* instruction. If an interrupt or trap occurs, a jump to the corresponding service routine takes place.

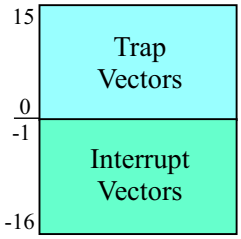

Figure 14: Exception Vector Table

3.3 Code Restrictions

The superscalar approach implicates various considerable problems regarding code compatibility due to possible data dependencies, for example:

```
instr.A: ADD r1, r2;
instr.B: ADD r4, r1;
```

If these two instructions are going to be executed in the same clock cycle, forwarding of register r1 (the result of instruction A) to instruction B has to take place inside the ALU. Of course not only the ALU, but also the data memory and other components of the processor core require data forwarding. Obviously a lot of forwarding paths between the two instruction execution pipelines will be necessary to resolve all code restrictions, which will lead to a notable increase of the processors gate count. The example given above also indicates that the need for forwarding will increase the latency of the pipeline stages (in the given example two sequential ALU operations have to be executed within one clock cycle). Another forwarding problem of great concern is given by the following example:

```
instr.A: CMP_EQ r1, r2;
instr.B: MOV_CT r4, r1;
```

During the decode pipeline stage SPEAR determines if a conditional instruction is going to be executed or not (the SPEAR architecture has been shown in section 1.2.1). In case of the superscalar approach the shown example needs special treatment to ensure that the condition flag is available in time to serve the conditional instruction as input.

How the code restrictions have been resolved and whether there are any left or not will be discussed in detail in chapter 4

4 Overall Design of the Microcontroller

In this chapter all design decisions for LANCE will be explained in detail. The LANCE processor is intended to extend the SPEAR design to a more powerful superscalar design. Therefore, an in-depth analysis of the SPEAR architecture will be presented and all design relevant issues of the superscalar approach will be discussed. Furthermore, important implementation details of the architectural components like instruction memory, register file, data memory, etc. are pointed out.

4.1 Increasing Speed

The first approach to improve the throughput of LANCE with respect to the SPEAR processor leads to detailed analysis of all data paths within the SPEAR architecture. As highlighted in Figure 15, the identified critical processing path comprises the *register file (pipe register 2)*, the *data multiplexors* at the ALU input ports, the *ALU* and the write-back bus to the *register file framepointer generator*.

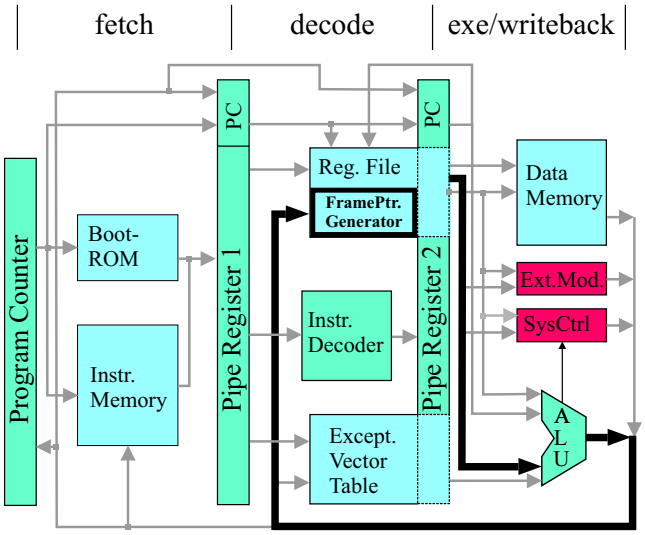

Figure 15: Critical Path

The following piece of code shows an operation along the critical path:

```
ADDI r26, 12
STOFX r2, 9
```

To provide the correct values for the initialization of register r26, which is coupled with the framepointer X operations, the input data for the *ADDI* instruction is multiplexed to the ALU input ports inside the execute/writeback pipeline stage. Concurrently (inside the decode pipeline stage) the succeeding STOFX framepointer operation has to sum up register r26 and the given offset to generate the appropriate memory location to store register r2. Therefore, the result of the ADDI instruction is forwarded out of the ALU to the *framepointer generator* where it is used for the address calculation. The ALU input multiplexor inside the execute/writeback, as well as the framepointer address generator inside the decode stage and the resulting unbalanced workload on these pipeline stages, limit the maximum clock rate of SPEAR. To resolve these limitations which result from the above identified bottlenecks, another pipeline stage (decode2) has been inserted between the SPEAR decode and execute/writeback stages, shown in the block diagram in Figure 16.

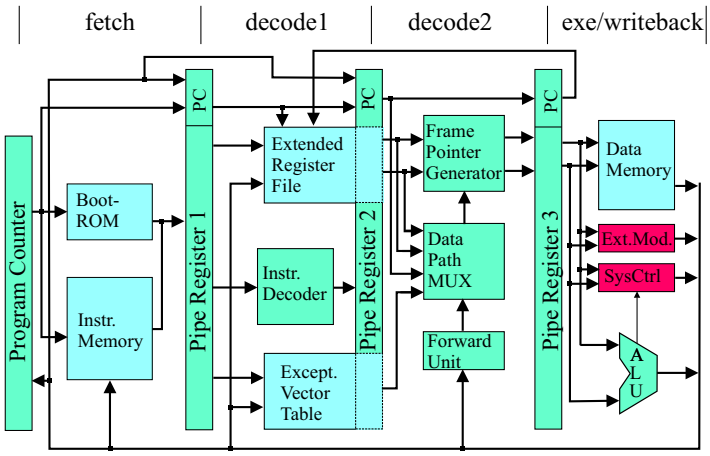

Figure 16: SPEAR Architecture Extended by an Additional Pipeline Stage

The additional pipeline stage allows to balance the workload of the pipeline stages which results in increase of clock rate and hence higher throughput. The address calculation for framepointer operations ($baseaddress + offset$) has been transferred to the decode2 pipeline stage. Moreover the multiplexor which provides the two ALU input ports with correct input data (out of the *program counter, the exception vector table* or the *register file*), also moves to the decode2 stage to speed up the execute/writeback cycle. For further increase of the ALU processing speed the ALU input data is preprocessed (details on preprocessing in section 4.4.3).

4.2 Superscalar Design Issues

One way to increased processing power while disregarding clock frequency arises from the superscalar design approach. The LANCE architecture consists of two operand execution pipelines called *Pipe A* and *Pipe B* (similar to the pentium design presented in section 2.1) and therefore represents a dual-issue superscalar architecture. The superscalar design of LANCE faces three main problems:

1. Data dependencies arise from the concurrent execution of two instructions per clock cycle
2. The number of possible *instruction memory, register file, exception vector table* and *data memory* accesses doubles in comparison to SPEAR.
3. Only one extension module access per clock cycle is supported by the standardized module interface.

To resolve as many code restrictions as possible both pipelines are able to execute any instruction of the instruction set. One for the programmer transparent restriction regarding extension module access exists within Pipe B (described in detail in section 4.4.7). As denoted in section 3.3, several data dependencies arise from the concurrent execution of two instructions. To ensure code compatibility to NEEDLE and SPEAR, considerable forwarding effort is necessary to resolve these data dependencies. It has been ensured that Pipe B always issues the succeeding instruction of Pipe A to limit the needed forwarding mechanisms to an absolute minimum (minimizing the number of forwardings also results in less silicon area). This constraint also assures that only forwarding from Pipe A to Pipe B is needed, but not vice versa. Section 4.4.1 gives detailed insights into the applied fetching mechanisms. Furthermore, section 4.5 points out implementation details on data forwarding.

As discussed in section 4.1, framepointer address generation has moved to the decode2 pipeline stage to achieve more balanced pipeline stages. This design decision implicates a code restriction illustrated by the following lines of code:

```
Pipe A: ADDI r26, 16;
Pipe B: LDOFX r22, -3;
```

The *ADDI* instruction alters the position of framepointer X inside the execute/writeback stage, but the *LDOFX* instruction has already calculated the memory address inside the decode2 pipeline stage. The resulting code restriction can not be resolved via simple data forwarding. Only inserting an additional framepointer address generator within the execute/writeback stage can solve this restriction. Since this extra hardware will extend the critical path, the code restriction is not resolved via hardware but has to be kept in mind during program code generation. Inserting a filler operation (an independent instruction or *NOP*) between the operation that sets the framepointer and the one that uses it is essential to achieve correct results. The drawback of this code restriction should be minor, because framepointers are only set infrequently and in many cases an independent filler operation can be found by code reordering mechanisms. Due to the fact that code compatibility to SPEAR and NEEDLE should be provided as far as possible, no further code restrictions were added to the LANCE design.
As LANCE determines whether a conditional instruction is going to be executed or not within the *pipeline register 3* (decode2 pipeline stage), the following code sequence introduced in section 3.3 demands special treatment:

```
Pipe A: CMP_EQ r1, r2;
Pipe B: MOV_CT r4, r1;
```

All compare and bit-test instructions issued to Pipe A perform their condition flag calculation inside the decode2 pipeline stage. Thus condition evaluation of conditional instructions which have been issued to Pipe B can take place

within *pipe register 3* as usual. Condition flag calculation of compare and bit-test instructions issued to Pipe B are performed within execute/writeback stage and the result is forwarded to *pipeline register 3* for further use by conditional instructions.

To ensure that each clock cycle two sequential instructions are read out of the *instruction memory*, the memory has been split into two blocks. One block holds all instructions with even addresses, the other block those with odd addresses. As mentioned above, Pipe A is loaded with the preceding instruction of that issued to Pipe B. A detailed description of the whole instruction fetch mechanism is given in section 4.4.1.

The *register file* has to be able to perform two independent write- and four read-accesses per clock cycle. As there are no memory modules within the APEX standard memory modules (an APEX-FPGA represents the desired target device) that meet those requirements, two different implementations have been verified. Both *register file* designs are presented in section 4.4.2.

Data Memory has to handle two read, two write or one read and one write action per clock cycle. To provide this functionality there have also been two approaches that will be shown in detail in section 4.4.6.

Finally the *exception vector table* also needed adapting to the superscalar design to provide both pipelines with the possibility to read and store the service routine jump addresses. Section 4.4.4 presents the implementation details.

The restriction that only one extension module access can be processed each clock cycle (due to the fact that the standardized interface does not support more) has been resolved by hardware. The extension module interface is only connected to Pipe A, therefore all extension module accesses occurring in Pipe B are handed over to Pipe A for further processing. Details on the whole hand-over mechanism are given in section 4.4.7.

4.3 Microcontroller Architecture

The LANCE processor aims to be code-compatible with and more powerful than SPEAR. All design issues discussed in section 4.1 and 4.2 led to a pipelined dual-issue superscalar design with a four-stage-deep pipeline, as shown in Figure 17. The presented LANCE architecture is able to execute any program code written for NEEDLE and SPEAR due to extensive data forwarding and special treatment of extension module accesses. However, the code restriction regarding framepointer operations, introduced in section 4.2, has to be considered.

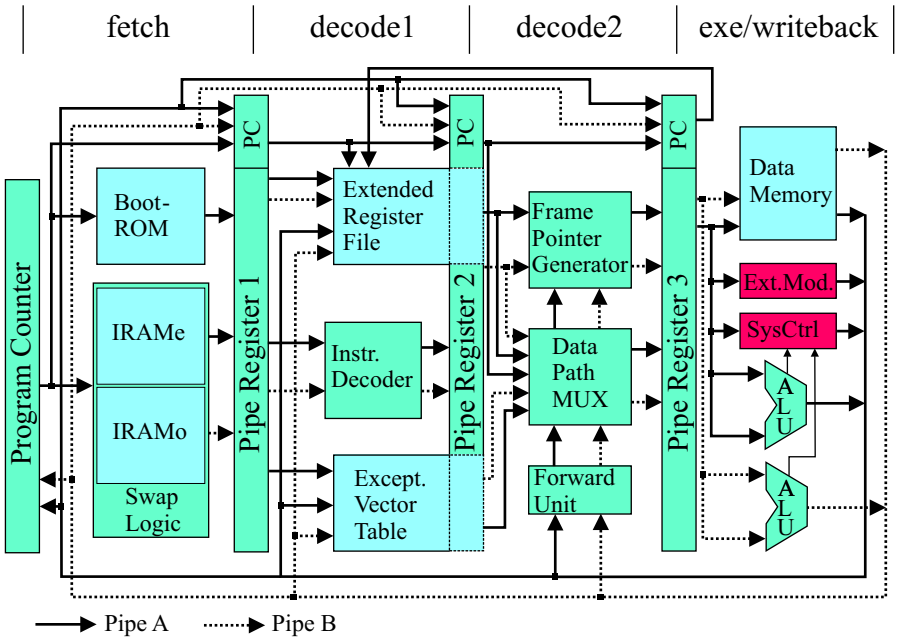

Figure 17: LANCE Architecture

4.4 Microcontroller Implementation

As mentioned earlier, LANCE comprises two instruction execution pipelines, Pipe A and Pipe B as shown in Figure 18. Both pipelines are able to handle any instruction within the instruction set of the processor. However, extension module accesses are processed, in a manner transparent for the programmer, via Pipe A only since the extension module interface does not support concurrent access to modules (section 4.4.7 provides detailed information on the implemented extension module access mechanism). Pipe A and Pipe B are organized as four-stage-deep scalar pipelines. The pipeline stages are as follows:

- Instruction Fetch (Fetch)
- Instruction Decode 1 (D1)
- Instruction Decode 2 (D2)
- Execute and Writeback (EX/WB)

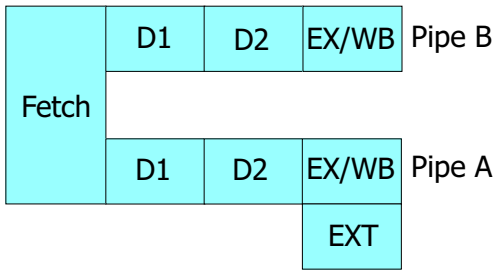

Figure 18: LANCE Pipeline Architecture

The first pipeline stage is the Fetch stage in which instructions are read from *Boot-ROM* or *instruction memory* and stored to the *pipe register 1*. On system startup instructions are always fetched from the *Boot-ROM* at a rate of one instruction per clock cycle, therefore only Pipe A has to execute instructions (Pipe B is filled with *NOPs*). Performing the boot sequence only

on one pipeline allows to use the same *Boot-ROM* as SPEAR and NEEDLE do. After the *instruction memory* has been filled, the processor switches over to the *instruction memory* and starts reading instructions at a rate of two instructions per clock cycle to utilize both pipelines.
The decode1 (D1) pipeline stage holds two parallel decoders, which generate the appropriate ALU and memory control signals corresponding to the processed instruction opcodes. The operands of both currently issued instructions are also read out of the *register file* and/or the *exception vector table* within the decode1 pipeline stage.
The decode1 stage is followed by decode2 (D2) where another control signal generation and preprocessing for the ALU takes place. Multiplexing of the correct input data to the ALU input ports is also done within the D2 cycle. *Compare* and *bit-test* instructions issued to Pipe A are evaluated within decode2 stage. This is necessary to provide conditional instructions processed in parallel inside Pipe B with the correct condition flag value. Furthermore, address calculation for framepointer operations is handled within the D2 pipeline stage.
In the LANCE design the execute/writeback (EX/WB) stage of the pipeline is responsible for *data memory* access, but also performs all arithmetic and logical operations within the ALU. Furthermore, results are passed through to the *register file* and the *exception vector table* as well as to the *program counter* and all *pipeline registers* along both write-back busses. If an extension module access (EXT) occurs it is also handled during the execute/writeback cycle.

Detailed information on design and implementation approaches of the processor core components is given in the subsequent paragraphs.

4.4.1 Instruction Memory and Boot-ROM

The *instruction memory* has to ensure that the processor operates at full capacity, therefore two valid, sequential instructions have to be read out of the *instruction memory* each clock cycle. In order to keep the instruction fetch mechanism simple, a single instruction memory, which contains two instructions per line, could be used. This concept, known as VLIW (Very Long Instruction Word) computer architecture [29], assures that two successive instructions, starting with an even memory address, are read each cycle. In the case of a jump to the second instruction of such an instruction-memory-line, the first instruction has to be discarded to guarantee correct program execution behavior. This would result not only in loss of performance, but also would make it impossible to perform exact WCET analysis since the program execution time would depend on the LSB of jump-target addresses which are known only at runtime (a jitter of one clock cycle on each branch instruction would be the consequence). The above mentioned problems regarding branch-instructions lead to the architecture depicted in Figure 19.

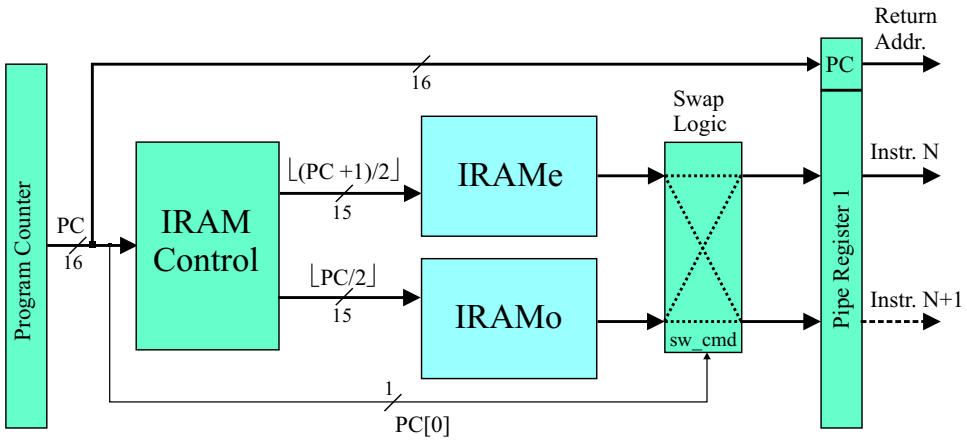

Figure 19: Instruction Fetch - Swap Mechanism

The *instruction memory* is 8 kB in size and has been split into two blocks of 4 kB each. One block holds all instructions with even instruction addresses (IRAMe), the other block the instructions with odd addresses (IRAMo). The *program counter*, without the least significant bit (LSB) $\lfloor PC/2 \rfloor$ - the LSB is implicitly resolved due to the fact that two separate instruction memories have been used - is directly mapped to the IRAMo address port. Since IRAMe's address port is fed by the *program counter* increased by one $\lfloor (PC+1)/2 \rfloor$, this mechanism guarantees that two successive instructions, starting with the one the *program counter* points to, are fetched each clock cycle. By default instructions out of IRAMe are mapped to Pipe A and the instructions out of IRAMo are issued to Pipe B. These instructions have to be swapped if the *program counter* holds an odd value, because it has to be ensured that Pipe B holds the successive instruction of the one issued to Pipe A. The implemented instruction fetch mechanism ensures that two valid, successive instructions are fetched each clock cycle and therefore supports accurate WCET analysis.

The *Boot-ROM* is used for initializing the microcontroller on system start-up. The Boot-ROM can hold only 128 instructions, thus it is used to set up the communication and programmer extension module to transfer the application code to the instruction memory. The *Boot-ROM* architecture has not changed in respect to NEEDLE and SPEAR to allow the programmer to use the same assembler [7] as well as the identical programmer extension module. Due to the fact that the Boot-ROM has just one output port it is only able to retrieve one instruction per clock cycle. All instructions are issued to Pipe A as long as they are fetched from the *Boot-ROM*, in the meantime Pipe B is filled with NOPs and therefore stays idle.

4.4.2 Register File

As denoted in section 4.2, the *register file* has to be able to perform six independent data accesses each clock cycle, two register write and four register read actions. As there are no memory modules within the APEX standard memory modules that meet those requirements (an APEX-FPGA [1] represents the desired target device), the following two different implementations have been realized.

1. discrete flip-flop register array
2. two mirrored dual-port memories clocked at twice the processor clock frequency

The first design approach for the register file, the *discrete flip-flop register array* allows as many read and write accesses as needed for the reason that it is built of common flip-flops. With a register file size of 32 registers of 16 bit each, the flip-flop implementation leads to 512 required flip-flops. The flip-flop count is quite high for an FPGA implementation due to the fact that APEX devices, which are presented in detail in section 5.1, need one logic element (LE) for each flip-flop. In Addition to the 512 flip-flops which implement the 32 registers another 1000 LEs are needed for the multiplexors which select the correct location for the 16 bit input and the two 16 bit outputs of the register file. The implementation which uses memory blocks needs far less chip resources than this one.
The second approach, the *mirrored dual-port memory*, is depicted in Figure 20. Dual-port RAM modules, as provided by the APEX-FPGA, permit memory access twice per clock cycle, once for reading and once for writing data. Duplicating the RAM block as well as the memory contents allows to perform one write access and two independent read actions each clock cycle. The mirroring process is quite simple, all data written to the RAM is written into *RAM block 1* and *RAM block 2* at the same time. For read access each RAM block is addressed separately, which leads to the two independently

read operations announced above. The SPEAR processor [7] uses exactly this design as register file implementation. The *mirrored dual-port RAM* has been adapted to the LANCE architecture by increasing the clock rate to twice the processor's clock frequency (increasing clock rate of certain components is not uncommon, as shown by the Pentium 4 design in section 2.4). Pipe A reads and writes its data in the first half of the processor's clock cycle and Pipe B does the same in the second half. In that way the *mirrored dual-port RAM* design supports two register writes (write back the resulting data of Pipe A and Pipe B) and four register file read actions (read two instruction parameters for each pipeline). Doubling the clock rate for the RAM blocks has been considered as possible solution due to the fact that a similar memory design has reached clock rates up to 160 MHz.

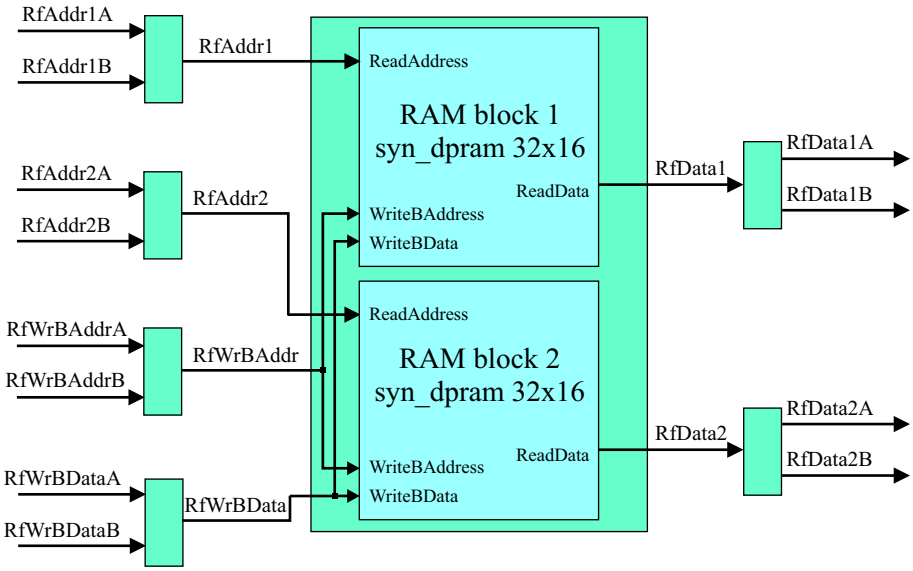

Figure 20: Register File Implemented as Memory

The *discrete flip-flop register array* was chosen as register file implementation for the LANCE design for two reasons. First of all it is more technology-independent and would also perform well within a ASIC (Application Specific Integrated Circuit) design or other FPGAs. The second reason is that the

mirrored dual-port memory does not perform well due to forwarding issues (the first half of the processor clock cycle has to last at least as long as the longest path in the exe/writeback stage to ensure correct data forwarding).

4.4.3 Instruction Decode

Instruction decoding has been split into two parts as depicted in Figure 17 in section 4.3:

- instruction decoder: generates ALU- and memory control signals (located inside decode1 pipeline stage).
- framepointer generator and data path multiplexor: performs framepointer address calculations and preprocesses data for the *ALU* (located inside decode2 pipeline stage).

Instruction decoder: The instruction decoder component is exactly the same as the one used in the SPEAR architecture. The only difference is that LANCE performs instruction decoding on two identical decoders in parallel to provide both pipelines with appropriate control signals.
Framepointer generator and data path multiplexor: The decode2 pipeline stage has been inserted to optimize the workload of each pipeline stage. Framepointer calculations from the register file and data-source multiplexing as well as data preprocessing from the *ALU* have moved to the decode2 stage.
As introduced in section 4.1, framepointer operations calculate the effective memory address by adding the offset (hardcoded into the framepointer instruction) and the appropriate framepointer, hence two *carry select adder* (CSA) have been implemented to provide this functionality.
The data path multiplexor provides the ALU input ports with correct input data out of the *program counter, exception vector table, register file* or the *writeback busses*. Furthermore, data preprocessing is done in terms of inverting operands for compare, invert or subtraction instructions to keep

the ALU delay as low as possible. As a result of data preprocessing, ALU operations have been merged. This leads to a smaller subset of ALU control signals which have been split into four distinctive groups to enhance the ALU performance.
Another important part of the decode2 pipeline stage is responsible for resolving compare and bit-test instructions issued to Pipe A to ensure correct behavior of conditional instructions processed in parallel. Therefore, another carry select adder is fed with the operands of pending compare instructions and the resulting condition flag is propagated to the *pipe register 3* to determine if the instruction inside Pipe B is going to be executed or not.

4.4.4 Exception Vector Table

In the LANCE architecture the *exception vector table* has to support two read, two write or one read and one write action per clock cycle. To provide this functionality, two different vector table implementations have been verified:

- discrete flip-flop array
- dual-port memory at double processor clock frequency

Similar to the *register file* implementations the exception vector table was built with flip-flops, which resolved the multiple access problem, but also led to drastic increase of silicon area (FPGA logic elements).
The second approach uses one dual-port memory clocked at twice the clock rate as the processor core (in contrast to the register file implementation no memory mirroring is needed, because only two, instead of four, read operations can occur concurrently). In the first half of the processor clock cycle the exception vector table is read and/or written by Pipe A, while Pipe B performs its operation on the vector table during the second half of the clock cycle.

As mentioned in section 4.4.2, the discrete flip-flop implementation provides a more technology-independent solution, whereas the dual-port memory approach leads to less logic elements on FPGA designs. Both implementations were tested, the resulting insights are presented in chapter 6.

4.4.5 ALU

The *ALU* implementation of LANCE adapted the SPEAR *ALU* in several different ways listed below.

- input multiplexor removed
- multiple small multiplexors for operation selection
- carry select adder with static carry-in
- data forwarding from Pipe A to Pipe B

First of all the multiplexor which provides the *ALU* with input data from the *register file, exception vector table* and *program counter* has been moved to the decode2 pipeline stage.
To keep the multiplexor, which selects the appropriate action for each ALU operation, as simple as possible, it has been replaced by a set of different small multiplexors. The control signals for these multiplexors are generated within the decode2 pipeline stage from the ALU control signals provided by the *instruction decoder*. Splitting the SPEAR ALU multiplexor to a set of smaller multiplexors led to a smaller amount of distinctive ALU operations and therefore aids speeding-up ALU operations.
Another important difference to the SPEAR ALU are the used carry select adders (CSA) which operate at hardcoded carry-in variables. The *ALU* comprises four distinctive CSAs, two of which are part of Pipe A and the others associated with Pipe B. The CSAs for each pipeline are distinguished by the carry-in value, one CSA has its carry-in port hardcoded to '0' the other one to '1'. The CSAs calculate their results as soon as input data changes. There is

no need to wait on any multiplexor switching to determine whether an ADD or a SUB instruction - with or without carry-in - has to be processed. The switching of the multiplexor to choose the correct result is done in parallel to the operations computation.
If no further code restrictions should be inserted, data forwarding from the *ALU* and/or the *data memory* associated with Pipe A to Pipe B's *ALU*, is necessary to ensure correct program behavior .
The concepts introduced above support a notable increase of ALU processing speed.

4.4.6 Data Memory

To provide the *data memory* with the possibility of two read, two write or one read and one write access per clock cycle, two designs similar to the exception vector table implementation (section 4.4.4) have been verified.
The discrete flip-flop array represents a solution for multiple memory access, which supports high clock rates but also implicates a large amount of required hardware resources. This concept does not perform well on FPGAs if large memories are necessary.
The dual-port memory clocked at double processor clock frequency leads to problems providing the address and/or data for the memory access at the right time if forwarding from Pipe A to Pipe B takes place. This approach needs far less hardware resources than the flip-flop approach if an FPGA represents the chosen target device.
Section 8.3 treats optimization issues in conjunction with FPGA devices as target technology, like the data memory access.

4.4.7 Extension Module Access

How extension module accesses are treated within the LANCE design has been introduced in section 4.2 and will be described in more detail in this section. Due to the fact that the standardized extension module interface does not support concurrent access to more than one extension module, a solution had to be found to eliminate this restriction within the LANCE design.

All extension modules are connected directly to Pipe A. If an extension module access takes place it can only be executed via Pipe A. If the instruction in Pipe B has to access an extension module, this will be identified within the execute/writeback pipeline stage. Once the extension module access has been recognized, the following four steps for instruction serialization, also depicted in Figure 21, are performed:

1. The instruction in Pipe A is executed as usual.
2. Pipe B's instruction is issued to Pipe A to be executed in the next cycle.
3. Pipe B's extension module access is replaced by a NOP.
4. All preceding pipeline stages are halted for one clock cycle, Pipe A performs the extension module access while Pipe B executes a NOP.

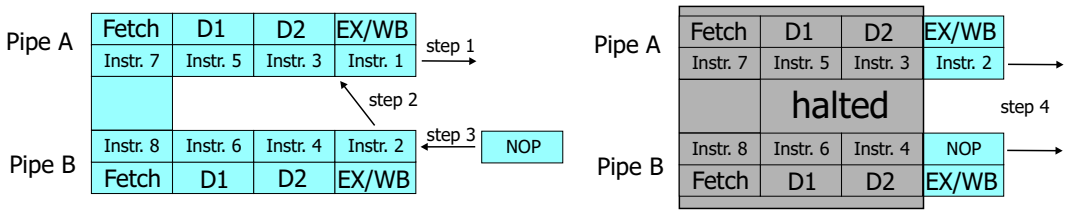

Figure 21: Extension Module Access

The decode2 pipeline stage determines if an extension module access is going to happen in Pipe A. To ensure that the LANCE architecture needs the same amount of time for extension module accesses no matter to which pipeline they are issued, Pipe A's extension module accesses are also halted for one clock cycle (one major requirement to maintain real-time capability of a processor is that all instructions **always** need a pre-determined time for execution). If both pipelines are going to access extension modules the introduced mechanism starts with processing the module access of Pipe A followed by the serialized module access of Pipe B. For each extension module access the pipeline stages are halted once which ensures constant execution time regardless of the amount and sequence of extension module accesses.

4.5 Data Forwarding

As denoted in section 3.3, a considerable amount of forwarding mechanisms had to be implemented to ensure code compatibility to NEEDLE and SPEAR. Due to the fact that the instruction fetch logic (presented in section 4.4.1) ensures that Pipe A always issues the preceding instruction of Pipe B, only forwarding from Pipe A to Pipe B is necessary.
The following data forwardings have been implemented:

forwardings within the writeback/execute pipeline stage:

- ALU Pipe A $->$ ALU Pipe B
- ALU Pipe A $->$ Data Memory Pipe B
- Data Memory Pipe A $->$ ALU Pipe B
- Data Memory Pipe A $->$ Data Memory Pipe B

forwardings between pipeline stages:
Data forwarding can take place from the write-back bus of either Pipe A or Pipe B (fed by the corresponding ALU or data memory) to Pipe A and Pipe B of the destination component. Inter-pipeline stage forwardings are needed for following paths:

- Writeback/Execute Pipe A $->$ Decode2 Pipe A
- Writeback/Execute Pipe A $->$ Decode2 Pipe B
- Writeback/Execute Pipe B $->$ Decode2 Pipe A
- Writeback/Execute Pipe B $->$ Decode2 Pipe B
- Writeback/Execute Pipe A $->$ Register File Pipe A

- Writeback/Execute Pipe A $->$ Register File Pipe B
- Writeback/Execute Pipe B $->$ Register File Pipe A
- Writeback/Execute Pipe B $->$ Register File Pipe B

Furthermore, the condition flag calculated inside the ALU of Pipe B is forwarded to *pipe register 3* to determine if succeeding conditional instructions will be executed or replaced by a NOP.

5 Development Environment

The LANCE processor core has been implemented in VHDL [3] like NEEDLE and SPEAR. The development environment used to design and download the LANCE architecture consists of Synopsys software for design verification and synthesis and the Quartus software for place-and-route and download into an FPGA [36][1]. A prototyping board from *El Camino* equipped with an APEX FPGA represents the target technology for the LANCE processor. The APEX FPGA and the prototyping board as well as the hardware design flow in relation to the above mentioned software are treated in this chapter.

5.1 APEX FPGA Family

An FPGA (Field Programmable Gate Array) is an integrated circuit which consists of an array, or a regular pattern, of logic cells. The logic cells can be configured to represent a limited set of functions. These individual cells are connected by a matrix of programmable switches. The developer's design is implemented by specifying the logic function for each cell and selectively closing the switches in the interconnect matrix. The array of logic cells and the interconnect matrix form a set of basic building blocks for logic circuits. These basic blocks are combined to achieve the intended behavior of more complex designs.
The logic cell architecture varies between different device families. In general, each logic cell combines a few binary inputs (typically between 3 and 10) to one or two outputs according to a boolean logic function specified in the programmed design. In most FPGA families, there exists the possibility of registering the combinatorial output of the cell, so that clocked logic (like counters or state-machines) can be implemented easily. The cell's combinatorial logic may be physically implemented as a small look-up table (LUT) or as a set of multiplexors and gates.

Field Programmable means that the FPGA's function is defined by the user's hardware configuration file (written in VHDL, VERILOG or System C) and not by the manufacturer of the FPGA chip. Typical integrated circuits perform a particular function defined at the time of manufacturing. Depending on the particular device, the program is either stored permanently or is loaded from an external memory each time the device is powered up. This kind of user programmability gives the user access to complex integrated circuit design without the high fabrication costs associated with ASICs (Application Specific Integrated Circuit).
The APEX family represents highly integrated FPGA devices which are manufactured in 0.22 μm to 0.15 μm processes. APEX devices are available in ranges from 30,000 to over 1.5 million gates. The features provided by APEX devices support high-performance system-on-a-programmable-chip (SOPC) solutions, allowing designers to integrate a system efficiently and use it in a broad range of applications. Figure 22 depicts the APEX architecture.

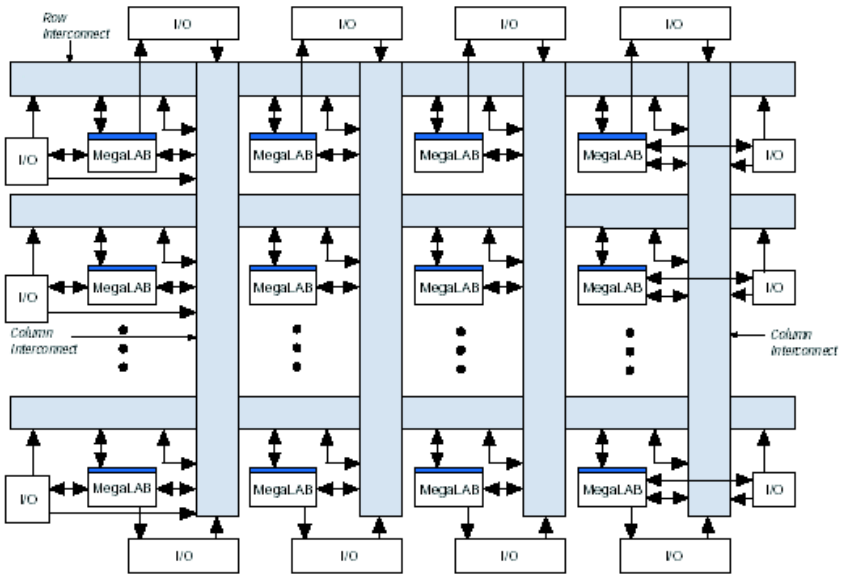

Figure 22: APEX Architecture

The APEX MultiCore architecture consists of so-called MegaLABs: These function blocks can be connected with each other as well as to I/O Pins. This MultiCore approach combines the strengths of LUT-based and product-term-based devices with an enhanced memory structure as shown in Figure 23. LUT-based logic provides optimized performance for data-path and register-intensive designs, whereas product-term-based logic is optimized for combinatorial paths, such as state machines. LUT- and product-term-based logic combined with memory functions provide a highly efficient approach, since applications usually require a combination of LUT-, product-term-, and memory-based components. Embedded system blocks (ESB) can implement a variety of memory functions, including first-in-first-out (FIFO) buffers, ROM or dual-port RAM functions. Embedding the memory directly into the FPGA improves performance in respect to external SRAMs. The ESBs support memory block sizes of 128x16, 256x8, 512x4, 1024x2 and 2048x1, but may be cascaded to implement larger sizes.

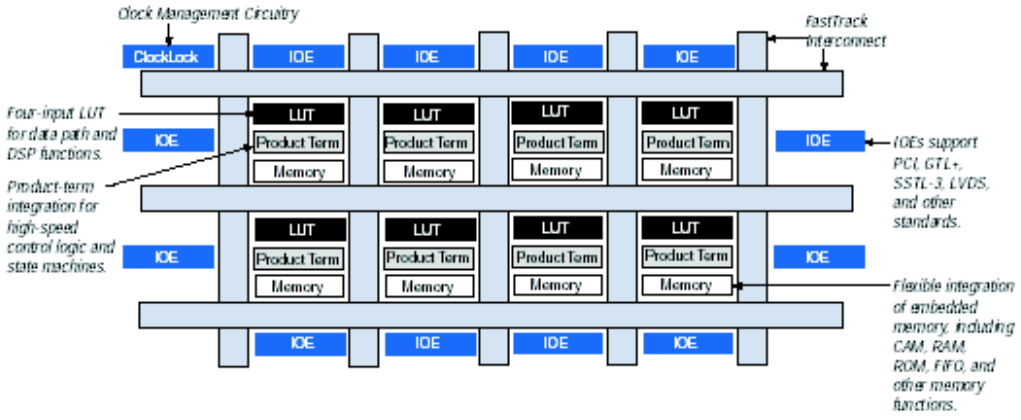

Figure 23: MultiCore and FastTrack-Interconnect Structures

Each I/O pin is fed by an I/O element (IOE) located at the end of each row and column of the FastTrack Interconnect. Two clocks are provided in APEX devices. These signals use dedicated routing channels to provide short delays and low clock skews. The clock pins can also feed ClockLock and ClockBoost

clock management circuits, which provide PLL (Phase-Locked Loop) functionality.
The MegaLAB Structure depicted in Figure 24, comprises a set of logic array blocks (LABs), one ESB, and a MegaLAB interconnect, which routes signals within the MegaLAB structure. The amount of LABs inside each MegaLAB depends on the specific APEX device, and can range from 10 to 24 LABs. Signal interconnections between MegaLABs and I/O pins are provided by the FastTrack Interconnect, a set of fast column and row channels (additionally LABs at the edge of MegaLABs can be driven by I/O pins via the local interconnect).

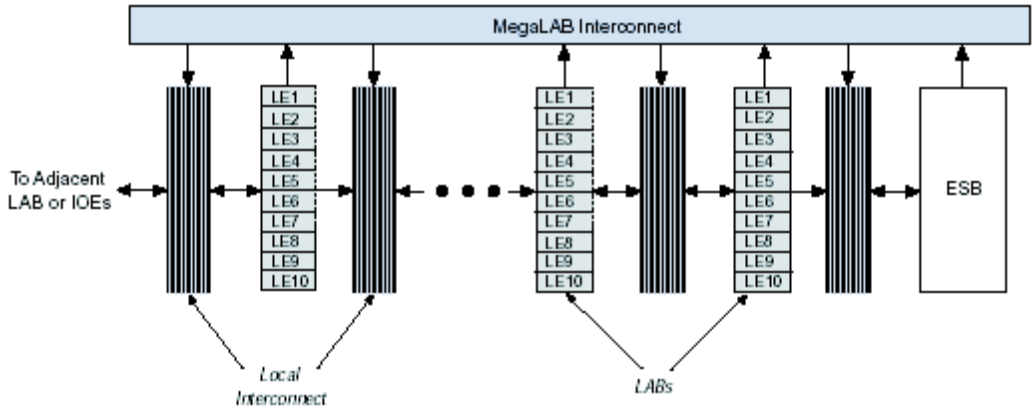

Figure 24: MegaLAB Structure

Each LAB consists of 10 logic elements (LE) and the associated local interconnect, as shown in Figure 24. Furthermore, the carry and cascade chains of each LE are also part of the LAB. Signals are transferred between LEs in the same or adjacent LABs, ESBs or IOEs via high-speed local interconnects. This feature minimizes the use of the MegaLAB and FastTrack interconnect and aids higher design performance and flexibility. The LAB-wide control signals can be generated from the LAB's local interconnect, global signals, or dedicated clock pins.

The logic element (LE), the smallest addressable logic unit in the APEX architecture, is very compact and provides efficient logic usage. Figure 25 shows a block diagram of an LE. Each logic element contains a four-input LUT, which is a function generator that is able to implement any function of four input variables. Furthermore, carry and cascade chains as well as a programmable register for D-, T-, JK-flip flop and shift register implementation are part of each LE. LEs can drive the local interconnect, MegaLAB interconnect, and the FastTrack interconnect structures.

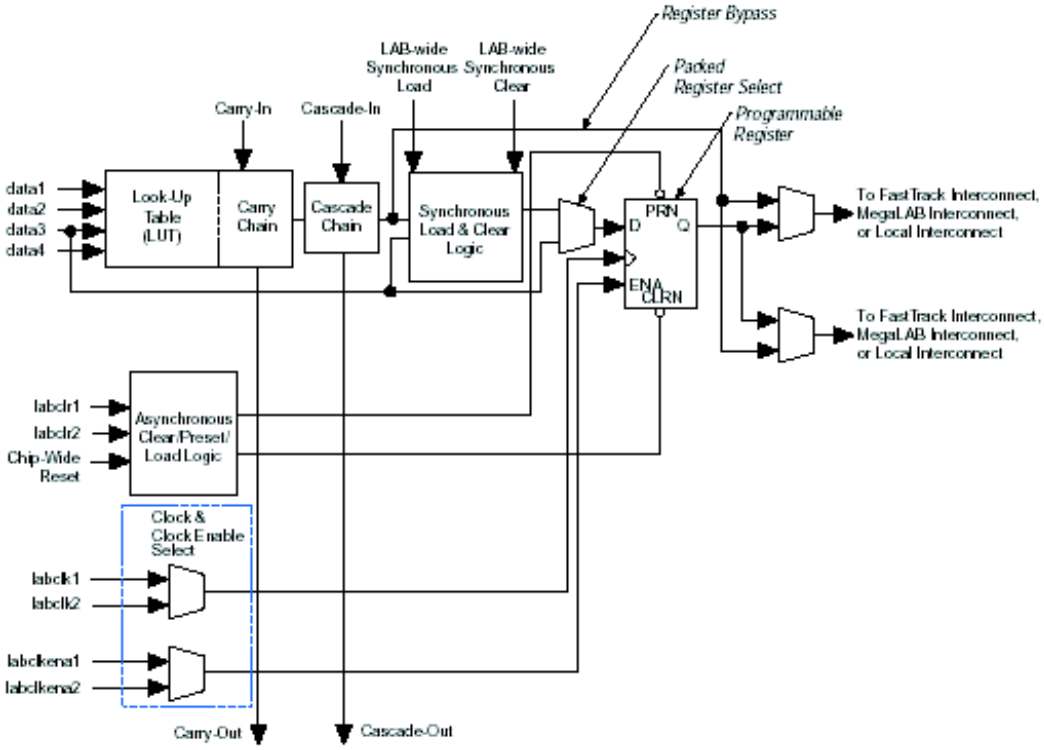

Figure 25: Logic Element Structure

5.2 DIGILAB 20Kx240 Prototyping Board

A DIGILAB 20Kx240 prototyping board was used for LANCE Processor tests [13]. Figure 26 depicts the configuration of the FPGA board.

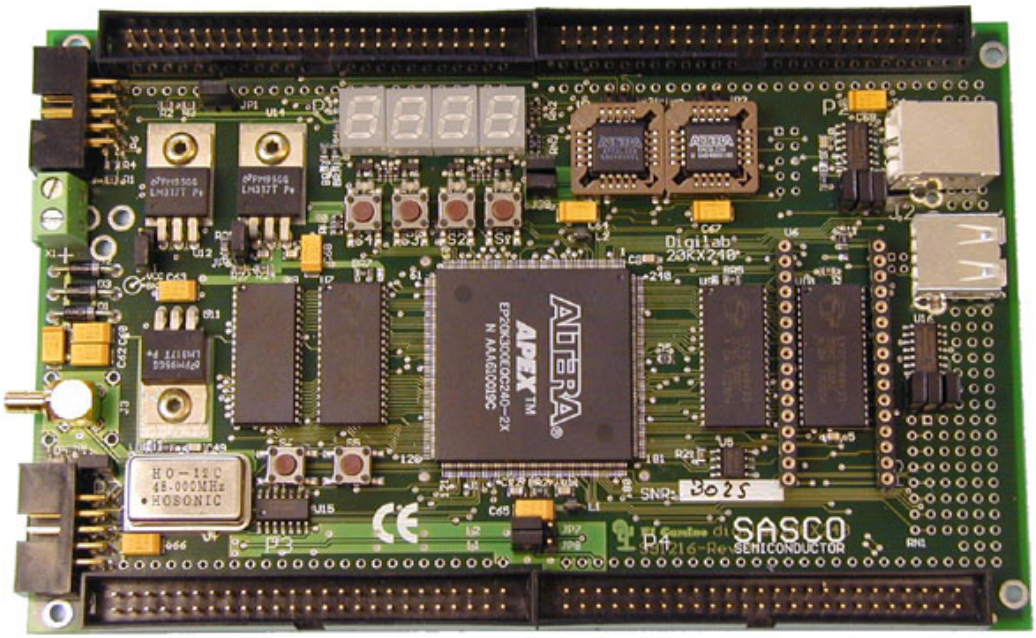

Figure 26: DIGILAB 20Kx240 Development Board

The chip in the middle of the board is the ALTERA APEX20K300EQ240-1X FPGA [1]. Two independent memory banks of the size of 512 x 16 Bit each are available in the two chips on the left and the right side of the APEX FPGA. The two EEPROMs above the APEX chip can be used for programming of the APEX FPGA.
There are two different ways of downloading the hardware configuration to the APEX device:

1. Direct programming to the FPGA: The APEX is directly programmed via the Quartus software.

2. Programming via EEPROMs: The hardware configuration is downloaded into the EEPROMs and transferred into the FPGA after a specific switch has been pressed.

A multiplexed four-digit 7-segment display, four two colored LEDs (red and green) and four push-buttons can be found next to the EEPROMs. In the upper left corner the programming interface to the PC can be found. At the lower left side of the board there are two additional switches, one of them is the global board reset, the other one triggers the hardware download from the EEPROMs to the APEX chip. At the upper left side two USB connectors can be found. The prototyping board supports three different I/O voltage levels 5V, 3.3V and 2.5V, which can be very useful when connecting external hardware via USB or the connectors at the upper and lower boundary of the board.

5.3 Design Flow

The hardware design flow used during the development of LANCE consists of several steps and is depicted in Figure 27. First of all circuits of the design are described in the hardware description language VHDL [17]. After this the correct behavior of the VHDL-code is verified via the behavioral simulation. The behavioral simulation is followed by the synthesis, where the VHDL-code is translated into logic cells. Afterwards the pre-layout simulation verifies the generated hardware, the timing behavior of each logic cell is included in this simulation. The succeeding place-and-route process maps the logic cells to the desired target device. After the place-and-route process, exact timing values for each logic cell as well as for the interconnects between the cells, are available. The obtained timing information is used within the post-layout simulation. The last step of the hardware design flow is the production of the chip, in the case of LANCE the download to the APEX FPGA. Each single step of the design flow is described in more detail in the subsequent sections.

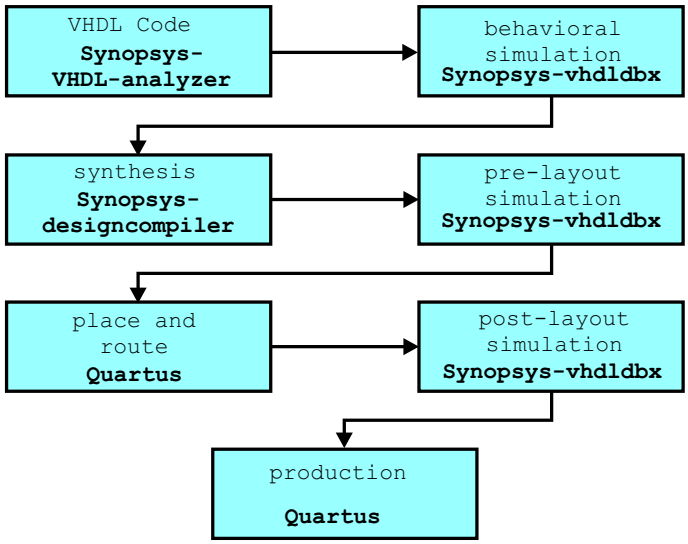

Figure 27: Hardware Design Flow

5.3.1 VHDL Coding

The LANCE processor as well as all extension modules have been designed in the hardware description language VHDL. VHDL stands for **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage and was initiated by the United States Department of Defense. In 1987 it was standardized within the IEEE standard 1076 [17], in 1993 the standard was revised (IEEE standard 1076-1993). VHDL is used for description of the behavior and structure of electronic hardware designs such as ASICs and FPGAs as well as conventional digital circuits. A detailed introduction to VHDL is given in [2][3].

5.3.2 Behavioral Simulation

For verification of correct programming syntax the Synopsys VHDL-analyzer is used [36]. After the VHDL-code has been analyzed, it is simulated with the VHDL Simulation System (VSS). This simulation includes no timing information for the created components. All timings in synchronous designs are based only on the clock signal. Figure 28 shows simulation results of the LANCE design.

5.3.3 Synthesis

For synthesis the Synopsys design-compiler, depicted in Figure 29, is used. The synthesis is responsible for transforming the behavioral model of the design (VHDL-code) into a structural representation, furthermore design optimizations are performed. The structural model represents a network of generic AND, OR and NOT cells. This network undergoes a further transformation into a target-technology-specific representation (FPGA cells to gates).

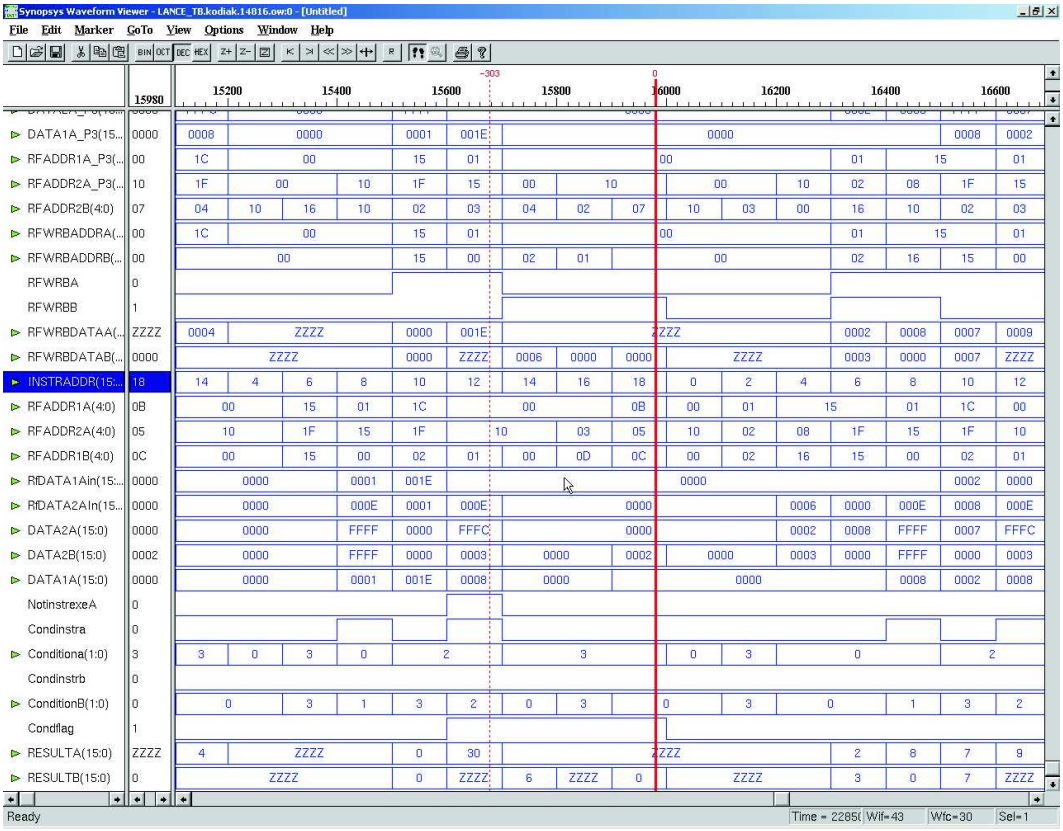

Figure 28: Synopsys Waveform Viewer

5.3.4 Pre-Layout Simulation

The pre-layout simulation, also performed via the Synopsys VHDL Simulation System, is necessary to verify the design after synthesis. The behavior of the hardware generated during synthesis is compared to the behavioral VHDL-model. Within this simulation the timing of all logic cells, but no delay caused by interconnects, is considered.

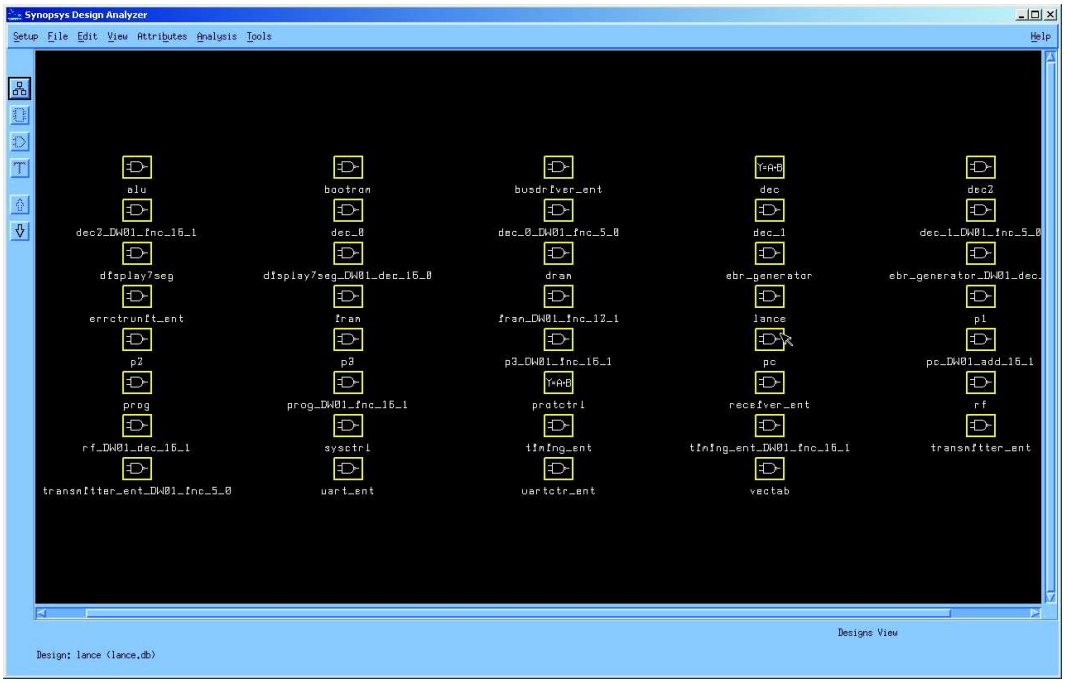

Figure 29: Synopsys Design Analyzer

5.3.5 Place and Route

During the place-and-route process the circuits generated during synthesis are attached to specific chip regions. Furthermore, the placed components are connected to each other. Usually the place and route follows a (user) defined optimization criterion (e.g. optimize to achieve a defined clock rate). Quartus represents the newest development environment for ALTERA programmable devices and is used for place and route as well as the download to the FPGA. The exchange format from the Synopsys design compiler to Quartus is the EDIF-netlist format. Figure 30 shows the results of a place and route of the LANCE processor.

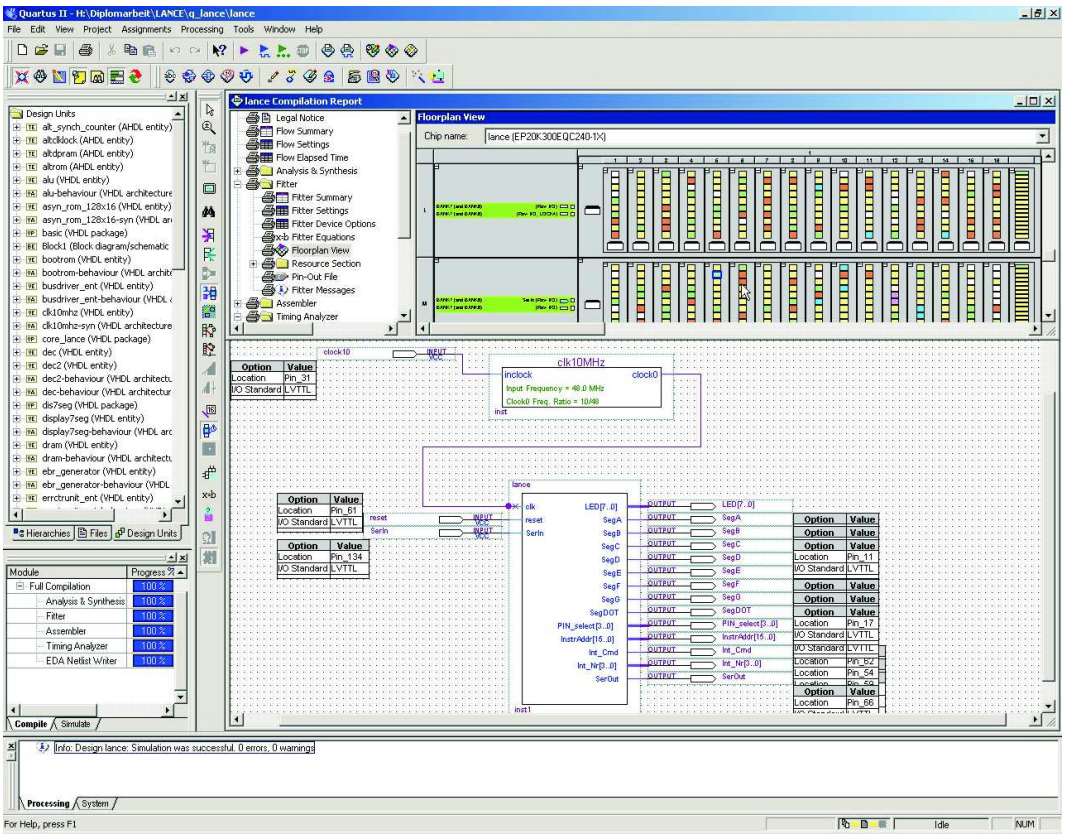

Figure 30: ALTERA Quartus

5.3.6 Post-Layout Simulation

During place and route a VHDL-file containing the exact timing behavior of the whole design can be generated. This VHDL-file is used to perform the post-layout simulation via Synopsys VSS to verify whether or not the intended design behavior is still provided.

6 Results

This chapter summarizes the results of the measurements and simulations of the developed superscalar microprocessor LANCE. The test environment built to compare the performance and overall behavior of NEEDLE, SPEAR and LANCE is also presented. To fit into the APEX20K300EQ240-1X FPGA provided by the DIGILAB 20Kx240 prototyping board a slightly simplified version of the LANCE processor was downloaded for testing. The behavior of LANCE was verified by using the 7-segment display and the LEDs of the prototyping board as well as a logic analyzer.

6.1 Test Environment

Code compatibility to NEEDLE and SPEAR had been one major requirement for LANCE design, therefore the test environment, designed for SPEAR can be used for testing LANCE. The test environment depicted in Figure 31, comprises the processor core, the processor control module (which is a vital part of the processor), a communications module (the UART extension module), the programmer module and the 7-segment display module. Additionally a boot-ROM initialization file and the download file of the test program are needed.

Before LANCE and its components have been downloaded to the prototyping board for testing, each single component and the whole processor have been simulated extensively. Since simulation only covers a short amount of time and a limited set of processor functions, the simulation can not proof correct behavior of LANCE.

The test on the prototyping board covers most of the processor's functions due to the usage of asynchronous interrupts, nested subroutine calls, conditional instructions, framepointer instructions, extension module access and many other operations.

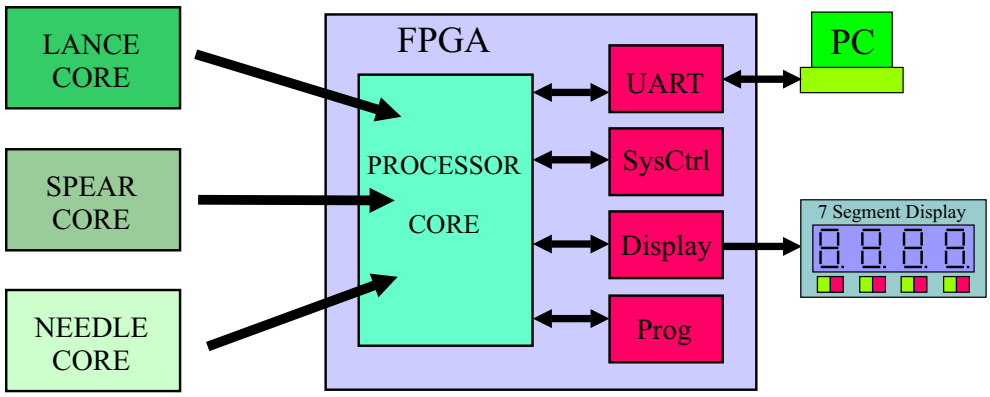

Figure 31: Test Environment

The test process can be structured into two main parts:

- Program download: LANCE operates on the boot-ROM contents
- Program execution: The downloaded program code is executed

At the beginning of the test the behavior of Pipe A as well as the UART and the programmer module is verified by downloading the actual test program from the Personal Computer (PC) to the instruction memory. First of all the processor waits for the synchronization pattern for the UART module to determine the transmission baud-rate. After synchronization the program code is transferred to the instruction memory. This boot sequence is provided by the boot-ROM program code and is processed via Pipe A only as described in section 4.4.1 (Appendix B shows the boot-ROM program code). The processor switches from boot-ROM to the instruction memory after the entire test program has been transmitted.
Both pipelines start fetching instructions as soon as the processor switches to the instruction memory. Whether the LANCE processor is running on the boot-ROM or the instruction memory can be determined easily by monitoring the program counter. The boot-ROM is active, if the program counter

is increased by 1 each clock cycle. However, if the program counter is increased by 2, instructions are fetched out of the instruction memory and both pipelines are utilized.
During the second part of the test the processor executes the previously downloaded program code. The test program which helps verify the correct processor behavior implements the calculation of the factorial of a number using recursive calls. At the begin of the test program the processor waits for the synchronization pattern for the UART module to determine the transmission speed. Using the UART module the hardcoded value for the factorial calculation can be replaced by a transmitted number. The C-program code corresponding to the factorial calculation is depicted below.

```
int factorial (int number)
{
    if (number == 0) return 1;
    return number * factorial(number - 1)
}
```

The *int factorial(number)* function calls itself with the argument decreased by one until $number = 0$. Afterwards it returns from the subroutine calls multiplying the result of the previous subroutine call with the current value of number which leads to the factorial of *number*.
After the synchronization of the UART module the factorial computation of the number takes place and the result is written to the 7-segment display extension module.
Due to the fact that the program download and the factorial calculation include complex program flow scenarios like multiple extension module access and nested subroutine calls in conjunction with interrupts (generated by entering new numbers for the calculation), a high test coverage has been achieved. The correct behavior of LANCE during the applied test scenarios showed that the LANCE design is fully operative.

6.2 Processor Characteristics

The LANCE processor consists of clock, reset and SerIn input-ports as well as the SerOut output-port. The SerIn and SerOut ports are needed to transfer the test program via UART module into the instruction memory. To simplify the testing of the processor the following ports, depicted in Figure 32, have also been added.

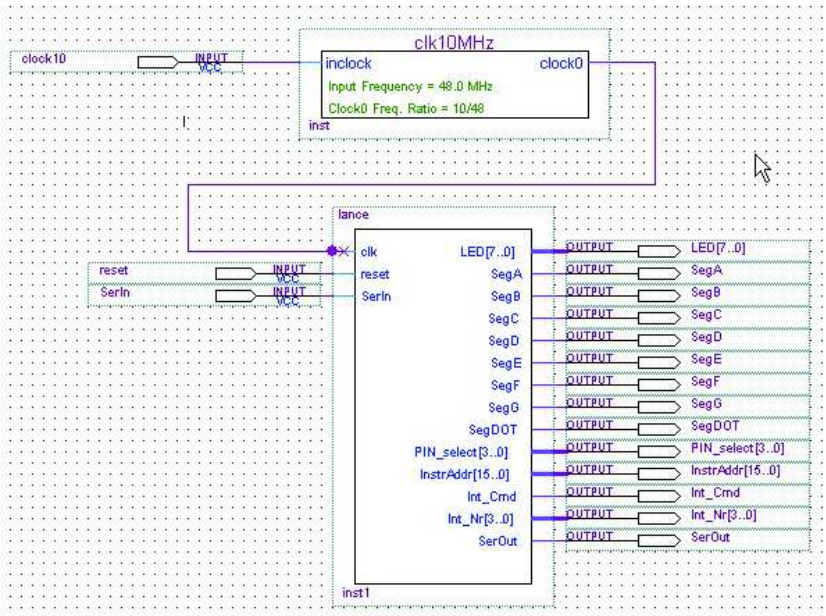

Figure 32: The LANCE Processor

- Instr_Addr: is directly connected to the program counter and is used for verifying correct behavior after program download (switch from boot-ROM to instruction memory), furthermore it is used to check the performance of jumps, interrupts and traps.
- Int_Cmd: shows if an interrupt has been identified by the processor core

- Int_Nr: holds the interrupt number of the interrupt detected by the processor control module
- LED: are used to access the LEDs on the prototyping board
- Pin_select and 7-segment signals: are connected to the 7-segment display on the test-board

The developed processor, which has been specified in chapter 4, did not achieve the expected performance gain due to the fact that the size of the design and the chosen implementation approaches do not scale well on FPGAs. To show the potential of the superscalar approach, several alternative concepts which needed fewer logic cells within an FPGA but have some design constraints released were tested also. Table 1 gives a comparison of NEEDLE, SPEAR and the version of LANCE which has been downloaded for testing, using an APEX20K300EQ240-1X as target device. The code restriction of the presented LANCE version concerns framepointer operations as introduced in section 4.2.

	NEEDLE	SPEAR	LANCE
Max Clock Rate	25 MHz	46 MHz	18 MHz
Silicon Area	0,33	1	3
MIPS	10	46	36
Instr-/Data Mem	2 kB/1.92 kB	4 kB/4 kB	8 kB/32
Register File	26 GPR+6 SPR	26 GPR+6 SPR	26 GPR+6 SPR
Interrupts/Traps	16/16	16/16	16/16
Instructions	80	80	80
Compatibility	full	full	1 code restriction

Table 1: Comparison of the NEEDLE, SPEAR and LANCE Processor Characteristics

The main cause for the unexpected low clock rates of the LANCE design is located in the high amount of used logic and the implicated enormous interconnect delays. As depicted in Figure 33, the interconnect delay along

the critical path of LANCE represents more than three-fourths of the processor's latency (Chapter 7 treats the interconnect delay problem in detail). The presented LANCE version provides the basis for superscalar microcontroller implementations which are optimized to specific requirements (e.g. target technology, code compatibility, gate count, etc.). A possible application field of the current available LANCE design is a single chip microcontroller for embedded real-time systems where an RC-oscillator is used instead of a quartz. At the low clock frequencies provided by RC-oscillators (below 10 MHz) LANCE provides almost twice the computational power of SPEAR.

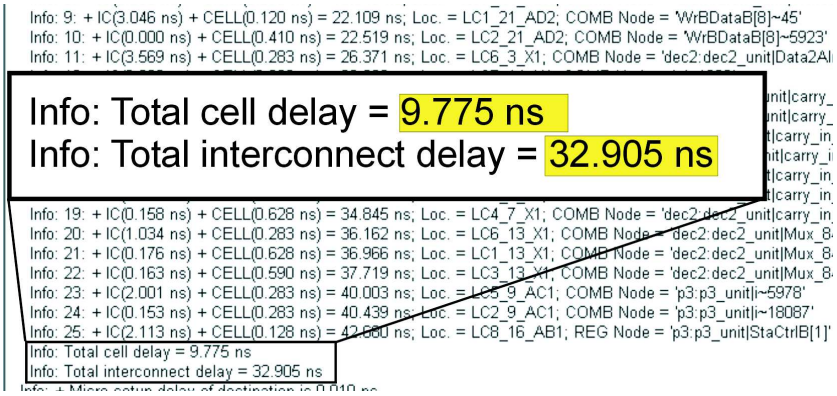

Figure 33: Quartus - Timing Report

6.3 Different Implementations

In order to verify the impact of various design approaches, different processor versions have been implemented. The parameters of the alternative LANCE designs as well as the resulting performance and restrictions are shown in Table 2.

	LANCE download	LANCE full	LANCE no forwarding	LANCE no flip-flops
DRAM	32 register	32 register	memory	memory
Register File	register	register	register	memory
exception-vector table	memory	register	memory	memory
vector table-Pipe A only	yes	no	yes	yes
DRAM Pipe A only	no	no	yes	yes
forward Pipe A $->$ Pipe B	yes	yes	no	no
forward EXE/WB $->$ decode2	yes	yes	no	no
Clock Rate	22 MHz	24 MHz	36 MHz	54 MHz
Si-Area	3	3.5	2	1.3

Table 2: Different LANCE Implementations

The LANCE processor version with the register file, exception vector table and data memory built of flip-flops ("LANCE full"column in Table 2), meets the functional requirements specified in chapter 3, but does not fit into the available APEX FPGA of the prototyping board. Therefore another version of LANCE ("LANCE download") had to be built to test the superscalar processor concept. The simplified LANCE processor implements the exception vector table as dual-port RAM which is connected only to Pipe A. This design fits into a APEX20K300EQ240-1X FPGA provided by the prototyping board. The restriction resulting from the simplified approach is that load and

store operations to the vector table have to be processed via Pipe A. This restriction has only a minor impact on program design, since operations on the vector table are only needed at processor startup for initializing of the exception service routine addresses.
The version of LANCE entitled with "LANCE no flip-flops"achieves high clock rates due to exploiting FPGA characteristic components like using ESB memory blocks instead of logic cells for data memory, register file and exception vector table. The problem regarding multiple memory accesses each clock cycle has already been described in section 4.2. The need for those multiple accesses was neglected by defining restrictions (data memory and exception vector table access only via Pipe A, ...). The implementation mentioned above therefore leads to a non-code-compatible processor version with a high clock rate.

6.4 Evaluation

A logic analyzer screenshot of the execution of the assembler code presented below is shown in Figure 34. LANCE executes instructions out of the instruction memory: the program counter is increased by two each cycle which indicates that two sequential instructions are issued in parallel.

```
006A:  LDL r20, -21
006B:  STW r2, r20;       Extension Module Access!
...      ...
006F:  CMP_EQ r21, r22
0070:  JMPI_CT finish;    jump_CT to label finish
0072:  LDL r19, 0
...
0098: finish: ...
```

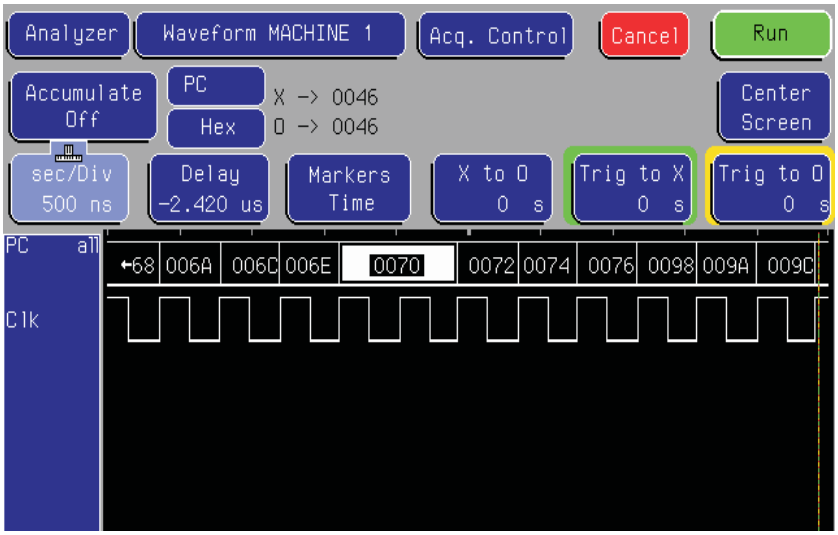

Figure 34: Logic Analyzer Screenshot - Instructions out of the IRAM

The *STW* instruction on address *006B* represents an extension module access. Since this instruction is issued to Pipe B, the extension module access

is recognized within the execute/writeback stage which corresponds to a *Program Counter (PC)* value of *0070*. As highlighted in Figure 34 the *PC* stays on a value of *0070* for two clock cycles which reflects the instruction serialization during extension module access (introduced in section 4.4.7). The execution of the *LDL* instruction on address *006A* takes place within the first clock cycle of the two cycle *PC = 0070* phase, whereas the the *STW* operation is processed during the second clock cycle.
The screenshot also shows that the jump of the *JMPI_CT* instruction which resides on address *0070* is executed because the PC value changes to *0098* after *JMPI_CT* passed through the execute/writeback pipeline stage.

6.5 Real-Time Capability

In real-time computer systems the correctness of computations depends not only on correct results in the value domain, but also on the physical instant at which these results are produced [24]. Therefore it is crucial that the response time to interrupts and the jitter of the response is known exactly (and minimal).

The LANCE design fulfils the requirement for minimum interrupt response time with minimum temporal jitter by aborting the current program flow and switching over to an ISR (interrupt service routine) as soon as an interrupt has been recognized. Figure 35 shows a logic analyzer screenshot of the interrupt response of LANCE.

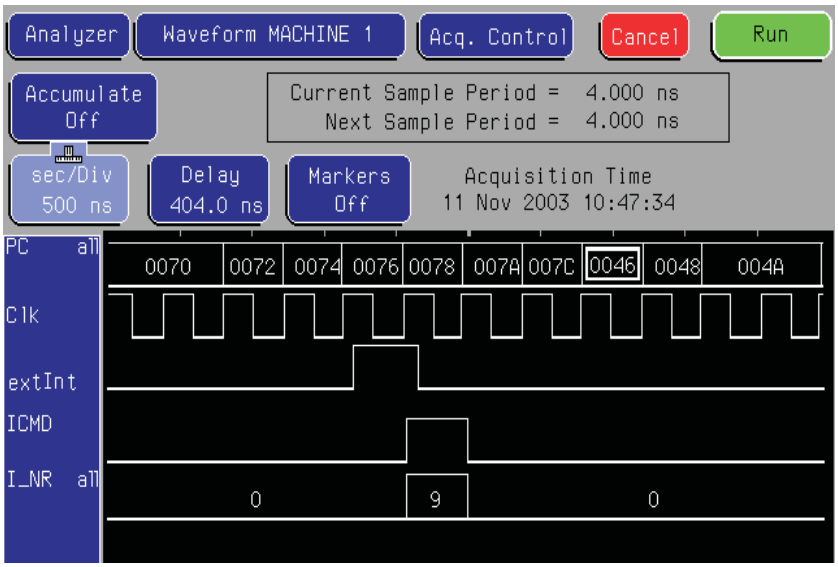

Figure 35: Logic Analyzer Screenshot - Interrupt Response

The interrupt response time can be divided into three parts: the recognition/synchronisation of the asynchronous interrupt, the context switch and the execution of the ISR. The interrupt is synchronized with the first rising edge upon its occurrence. One clock cycle after synchronization the processor

reads the ISR address from the *exception vector table* and passes it on to the decode2 pipeline stage. The return address as well as the processor status word are automatically saved after interrupt synchronization. With the next rising edge the address of the interrupt service routine is transmitted to the execute/writeback pipeline stage. Another clock cycle is required until the two first instructions of the service routine are fetched out of the *instruction memory*. Thus, for LANCE the interrupt response has a fixed delay of four clock cycles.
Due to the fact that the SPEAR pipeline only consists of three pipeline stages in contrast to the four-stage-deep pipeline of LANCE, the interrupt response time of SPEAR is only three clock cycles.

The exact prediction of task execution times on LANCE is supported due to extensive data forwarding and the LANCE processor's support of the SINGLE-PATH [30][9] programming model. The forwardings between stages of each pipeline as well as between both pipelines guarantee constant execution times of single instructions. Typical program code has a number of different program execution paths and the active execution path depends on current data values which implicate data dependent execution times. Program code which has been developed according to the SINGLE-PATH paradigm has only one execution path and therefore leads to completely predictable timing behavior.

7 Conclusion

The processor developed in the course of this diploma thesis represents a temporally predictable 16-bit superscalar processor. The design aim was to be code-compatible to NEEDLE and SPEAR as far as possible. The implementation fulfils all requirements posed by the specification except for the aimed performance within the used target device. The reasons for the unexpected low performance of LANCE reflect the current problems regarding chip design. The main problems concerning hardware design as well as approaches to solve these problems are presented in current literature and can be divided into the following topics:

- Gate delay vs. interconnect delay: the decreasing feature size in chip design shifts the focus from the decreasing gate delay to the increasing interconnect delay.
- Limits of FPGA: not all kinds of hardware design scale well on FPGAs due to technological constraints.
- Hardware/Software co-design: optimized performance can be achieved through balancing hardware and software effort.

The development of LANCE showed that the main challenge in current hardware designs does not lie in minimizing the gate delay, but in handling the interconnect delay of designed circuits. The increased importance of interconnect delay has its origin in the decreasing feature sizes of ASICs as well as FPGAs [32] [33]. Furthermore, rising gate counts of actual designs lead to additional wiring between components and therefore to increased interconnect delay. The used target device an APEX20K300EQ240-1X FPGA has been manufactured in a 0.18μm process. Figure 36 illustrates that the interconnect delay of a 0.18μm ASIC is about 2.5 times higher than the gate delay. The interconnect delay of the LANCE FPGA implementation is about 3.4 times higher than the gate delay. The notably higher interconnect delay

in respect to ASIC designs has its origin in the programmable interconnects which are used within FPGAs.

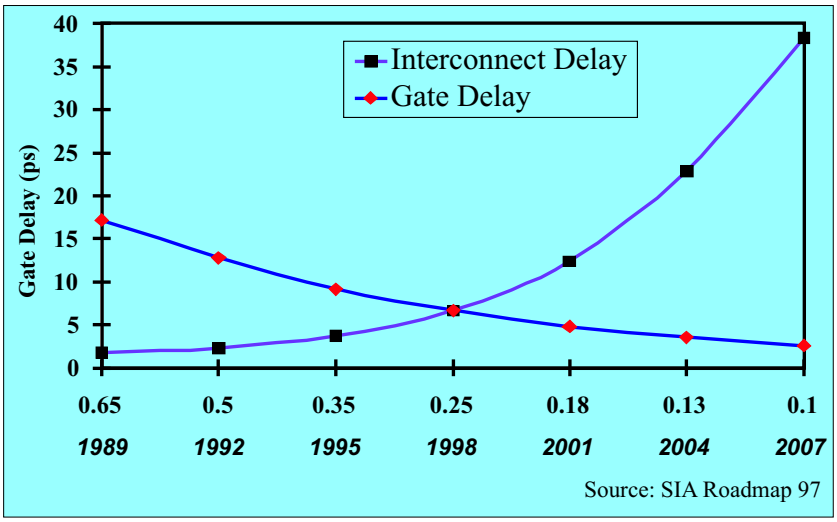

Figure 36: Interconnect Delay

The LANCE design has shown that FPGAs are not an ideal target technology for all kinds of hardware designs. Designs which lead to bad performance on FPGAs due to enormous wiring costs are:

- designs with a high degree of FPGA utilization where interaction between the majority of the implemented components is needed
- memory intensive designs where the memory is implemented in discrete logic

The LANCE design has shown that the target technology characteristics are a topic of great concern during the development process. The design performance mostly depends on whether target technology features have been considered during development or not (e.g. memory implementation: memory blocks vs. flip-flop array).
Another way to solve performance bottlenecks within hardware designs is

provided by hardware/software co-design. Hardware and software effort is balanced for optimized performance and flexibility of the whole developed system. There is no need to resolve all code restrictions within a processor (e.g. LANCE) via hardware, some restrictions may be handled by the compiler or have to be kept in mind by the programmer. This approach leads to less complex hardware and decreases gate count as well as interconnect delay and power consumption of the design. Balancing the hardware and software effort to attain optimal processor performance is a non-trivial problem. This optimization process can not provide a general optimum since requirements vary for each application.

For future hardware designs not only the target technology and the implicated features and constraints have to be considered during the development process, but also notable effort has to be put into hardware/software co-design. Only extending the perspectives and the field of activity of the hardware designer enables innovative hardware/software solutions.

8 Outlook

Three different topics for further research and design optimizations are presented in this chapter.

8.1 Task to Pipe

The Task-to-Pipe idea addresses current processor concepts like the Hyper-Threading technology of the Pentium 4 processor. Hyper-Threading Technology (HT) stands for the ability of the Pentium 4 processor to execute more than one thread at a time [23] [15], therefore the processor appears as two processors to the operating system.
In the case of the LANCE design it should be possible to adapt the processor that Pipe A and Pipe B operate on two independent tasks (threads). Only little adaption on the existing instruction fetch mechanism would be necessary due to the fact that the instruction memory already consists of two memory blocks. A second program counter as well as independent pipe registers would be necessary to support two autonomous instruction streams. One main objective to support two parallel tasks would be to develop a shared-memory concept for data exchange between the two pipelines. The register file may be duplicated, or split into two parts, to support both pipelines. A register locking mechanism is also a possible solution, but would make it more difficult to obtain deterministic timing behavior. The extension module interface as well as the system control module (especially the processor status register) would also need special treatment.

The Task-to-Pipe concept could be used to split operating system tasks and application tasks to distinctive pipelines. This should preserve the processor from the need of context switching if the operating system is permanently executed on Pipe A and the application threads are issued to Pipe B.

8.2 Fault Tolerant System

Fault tolerance is necessary in safety-critical real-time systems to ensure that no single failure can lead to catastrophic system failure [24].
A fault tolerant processor provides redundancy in terms of additional hardware. In case of LANCE Pipe A and Pipe B could be considered two separate pipelines both working on identical sets of input data. The results provided by Pipe A would be cross-checked with the results of Pipe B. A predefined test-sequence could be used to identify the erroneous pipeline after different pipeline results have been detected. After the erroneous pipeline has been identified it can be disabled, or the whole processor may switch over into a defined failure-mode. Furthermore, CRC-checks for all data buses could be implemented to increase the error detection rate.
Another possible approach would be to check not only the results of the whole pipeline, but also the outgoing signals of each component. This would make it possible to re-route instructions on redundant hardware to bypass erroneous components and therefore would lead to increased fault tolerance. If for example the decode stage within Pipe A and the execute/writeback stage of Pipe B were defective, the system could still provide correct results if the instructions were re-routed to proper working components of the other pipeline.
The third possible design could implement a third pipeline and a voting mechanism which compares the results of all pipelines and selects the result that has been computed by the majority.
The main problem concerning superscalar designs, the multiple memory access per clock cycle, is not present in the introduced fault tolerant designs, since only one instruction is issued per clock. The resulting processor should be able to operate at a peak performance below that provided by SPEAR, but therefore would implement a fault tolerant 16 bit processor. The silicon-area of a fault tolerant processor based on LANCE should be notably below that of LANCE due to the fact that far less data forwarding will be needed.

8.3 FPGA Optimizations

This section treats design optimizations for the existing LANCE designs to achieve higher performance with FPGAs as target technology.

The LANCE design is a fully operative superscalar processor, but the achieved performance within FPGA devices does not satisfy the expectations. The discrete flip-flop approach for the register file, exception vector table and data memory will perform well in ASICs, but for FPGAs as target technology another solution has to be found to achieve moderate performance.
The performance penalty of the LANCE design was caused mostly by the "multiple memory access problem" which led to increased hardware effort for the register file, exception vector table and data memory. Therefore it would be desirable to remove the flip-flop based memory implementations. One possible solution would be to migrate to ALTERA mercury FPGAs which support quad-port RAM modules [1]. Quad port RAMs would lead to simple memory interfaces as implemented within SPEAR. Since this solution is extremely hardware dependent, another design approach which would lead to reduced hardware effort and especially decreased interconnect delay would be preferable.
A revised version of the "mirrored dual-port memory at doubled clock rate" approach (discussed in section 4.4.2), will eventually lead to higher processor performance within FPGAs.

A Appendix - The Instruction Set

Complete instruction set.

	instruction	explanation
1.	LDL r1,n8	8 bit const n8 → r1[7..0]; r1[15..8] = n8[7]
2.	LDH r1,n8	8 bit const n8 → r1[15..8]; r1[7..0] = r1_old[7..0]
3.	ADD r1,r2	r1 = r1 + r2 (without carry_in)
4.	ADDC r1,r2	r1 = r1 + r2 + carry_in
5.	SUB r1,r2	r1 = r1 - r2 (without carry_in)
6.	SUBC r1,r2	r1 = r1 - r2 - carry_in
7.	MOV r1,r2	r2 → r1
8.	MOV_CT r1,r2	r2 → r1 if condition_flag = true
9.	MOV_CF r1,r2	r2 → r1 if condition_flag = false
10.	CMPI_EQ r1,n5	Compare Equal, r1 and signed 5 bit constant n5
11.	CMP_EQ r1,r2	Compare Signed Equal
12.	CMP_LT r1,r2	Compare Signed Less Than
13.	CMP_GT r1,r2	Compare Signed Greater Than
14.	CMPUN_LT r1,r2	Compare Unsigned Less Than
15.	CMPUN_GT r1,r2	Compare Unsigned Greater Than
16.	ADDI r1,n5	Add immediate r1 and signed 5 bit constant n5
17.	ADDI_CT r1,n5	Add immediate r1 and signed 5 bit constant if condition_flag = true
18.	ADDI_CF r1,n5	Add immediate r1 and signed 5 bit constant if condition_flag = false
19.	AND r1,r2	r1 = r1 AND r2
20.	OR r1,r2	r1 = r1 OR r2
21.	EOR r1,r2	r1 = r1 XOR r2
22.	LDW r1,r2	Mem(r2) → r1
23.	STW r1,r2	r1 → Mem(r2)
24.	LDOFX r1,n5	Mem(StptrX+n5) → r1
25.	LDOFY r1,n5	Mem(StptrY+n5) → r1
26.	LDOFZ r1,n5	Mem(StptrZ+n5) → r1
27.	STOFX r1,n5	r1 → Mem(StptrX+n5)
28.	STOFY r1,n5	r1 → Mem(StptrY+n5)
29.	STOFZ r1,n5	r1 → Mem(StptrZ+n5)

	instruction	explanation
30.	LDVEC r1,n5	LoaD Exception VECtor n5 → r1
31.	STVEC r1,n5	r1 → STore Exception VECtor n5
32.	TRAP r1,n4	activates jump to address stored on position n5 of the Exception Vector Table
33.	BTEST r1,n4	if Bit n4 is set in r1 → condition_flag = true
34.	BSET r1,n4	Bit n4 will be set in r1
35.	BSET_CT r1,n4	Bit n4 will be set in r1 if condition_flag = true
36.	BSET_CF r1,n4	Bit n4 will be set in r1 if condition_flag = false
37.	BCLR r1,n4	Bit n4 will be deleted in r1
38.	BCLR_CT r1,n4	Bit n4 will be deleted in r1 if condition_flag = true
39.	BCLR_CF r1,n4	Bit n4 will be deleted in r1 if condition_flag = false
40.	NOT r1	r1 is negated
41.	NOT_CT r1	r1 is negated if condition_flag = true
42.	NOT_CF r1	r1 is negated if condition_flag = false
43.	SL r1	r1 is shifted left
44.	SL_CT r1	r1 is shifted left if condition_flag = true
45.	SL_CF r1	r1 is shifted left if condition_flag = false
46.	RL r1	r1 is rotated left (MSB becomes LSB)
47.	RL_CT r1	r1 is rotated left if condition_flag = true
48.	RL_CF r1	r1 is rotated left if condition_flag =true
49.	SR r1	r1 is shifted right
50.	SR_CT r1	r1 is shifted right if condition_flag = true
51.	SR_CF r1	r1 is shifted right if condition_flag = false
52.	RR r1	r1 is rotated right (LSB becomes MSB)
53.	RR_CT r1	r1 is rotated right if condition_flag = true
54.	RR_CF r1	r1 is rotated right if condition_flag = false
55.	SRA r1	r1 is shifted arithmetic right
56.	SRA_CT r1	r1 is shifted arithmetic right if condition_flag = true
57.	SRA_CF r1	r1 is shifted arithmetic right if condition_flag = false
58.	ROLC r1	r1 is rotated left with carry (carry becomes LSB)
59.	ROLC_CT r1	r1 is rotated left with carry if condition_flag = true
60.	ROLC_CF r1	r1 is rotated left with carry if condition_flag = false

	instruction	explanation
61.	RORC r1	r1 is rotated right with carry (carry becomes MSB)
62.	RORC_CT r1	r1 is rotated right with carry if condition_flag = true
63.	RORC_CF r1	r1 is rotated right with carry if condition_flag = false
64.	JSRX r1	JSR, the return address is stored in register X
65.	JSRX_CT r1	JSR, if condition_flag = true, the return address is stored in register X
66.	JSRX_CF r1	JSR, if condition_flag = false, the return address is stored in register X
67.	JSRY r1	JSR, the return address is stored in register Y
68.	JSRY_CT r1	JSR, if condition_flag = true, the return address is stored in register Y
69.	JSRY_CF r1	JSR, if condition_flag = false, the return address is stored in register Y
70.	JMP r1	Jump, r1 contains address
71.	JMP_CT r1	Jump, if condition_flag = true
72.	JMP_CF r1	Jump, if condition_flag = true
73.	JMP offset	Jump (immediate) offset contains address
74.	JMP_CT offset	Jump, if condition_flag = true
75.	JMP_CF offset	Jump, if condition_flag = false
76.	RTSX	RTS, return address in register X
77.	RTSY	RTS, return address in register Y
78.	RTE	Return from Exception
79.	NOP	No OPeration
80.	IllOp	Illegal Opcode

B Appendix - Assembler Code

```
begin:      ; set UART interrupt vector to 9th bit
            LDL r0,lo(rec_int);
            LDH r0,hi(rec_int);
            STVEC r0,-7;
            ;set framepointer
            LDL r26,-32;
            LDL r27,-8;
            ; UART config: no parity bit,
             ;one stop bit, no transmition control
            ; msglength= 8 bit
            LDL r19, 0;
            LDH r19, 0b01000111;
            STOFX r19,-15;
            ; set UART to sync mode
            LDL r19, 0b01000001;
            STOFX r19,-14;

            ; initialize constants
            LDL r10,0;
            LDL r0,0;
            LDL r19,0b00000000;   set GIE
            LDH r19,0b10000000;    set GIE
            STOFY r19,1;          in SST memory
            LDL r31, lo(wait)  ;set return address
            LDH r31, hi(wait)
            RETI;

            ; wait until UART is synchronized
wait:       LDOFX r11,-16;
            BTEST r11,8
            JMPI_CF wait;
            ; sync finished
            LDOFX r17,-9;
            STOFX r17,11;    display-register
            STOFX r17,-12;    transmit-register
            LDL r19, 0b10011000;  transmit BR
            STOFX r19,-14;
            ; set UART to receveive mode
            LDL r19, 0b10100000;
            activate receive mode
            STOFX r19,-14;

wait1:      ; wait for CR
            LDOFX r11,-16;
            BTEST r11,10;
            JMPI_CF wait1;
            ; set event flag
            LDL r19, 0b11100100; receive-completion
            STOFX r19,-14;

wait2:      JMPI wait2;

rec_int:    LDOFX r12,-16;   receive interrupt
            BTEST r12, 2;
            JMPI_CT rec_error;

            LDOFX r12,-15;
            BSET r12, 0;
            STOFX r12,-15;
            ADDI r10,1;  increment byte counter
            STOFY r0,2;     ; clear interrupt

byte1:      CMPI_EQ r10,1;  address high-byte
            LDL r21,0xfe;
            STOFX r21,12;
            JMPI_CT ld_high;
```

```
byte2:      CMPI_EQ r10,2;
            low-byte der address
            LDL r21,0xfc;
            STOFX r21,12
            JMPI_CT ld_low;

byte3:      CMPI_EQ r10,3;
            type-byte,01 -> end of download
            LDL r21,0x0c;
            STOFX r21,12
            JMPI_CF byte4;
            LDOFX r9,-12;
            CMPI_EQ r9,1;
            JMPI_CT end_prog;
            RETI;

byte4:      CMPI_EQ r10,4;  instr. high-byte
            LDL r21,0xf8;
            STOFX r21,12
            JMPI_CT ld_high;

byte5:      CMPI_EQ r10,5;   instr. low-byte;
            LDL r21,0xf0;
            STOFX r21,12
            JMPI_CT ld_low;

byte6:      LDL r19, 0b00000000;
            LDH r19, 0b10000000;
            LDL r21,0x00;
            STOFX r21,12
            STOFX r19, -7;
            LDL r10,0;
            RETI;

ld_low:     LDOFX r8,-12;
            OR r8,r9;
            STOFX r8,11
            CMPI_EQ r10,5;
            JMPI_CT data
            STOFX r8,-6;
            RETI;

data:       STOFX r8,-5;
            RETI;

ld_high:    LDOFX r9,-12;
            LDL r15,8;


shift_l:    CMPI_EQ r15,1;
            SL r9;
            ADDI r15,-1;
            JMPI_CF shift_l;
            STOFX r9,11
            RETI;

end_prog:   LDL r19, 0;
            LDH r19, 0b00000011;
            ;switch to instruction memory
            STOFX r19, -7;

rec_error:  LDOFX r12,-16;
            STOFX r12,11;
            JMPI rec_error;
```

Figure 37: Boot-ROM Assembler Code

References

[1] Altera Corporation. *Apex 20K Programmable Logic Device Family, http://www.altera.com*, 2003.

[2] P. J. Ashenden. *The VHDL Cookbook.* Dept. Computer Science University of Adelaide South Australia, 1990.

[3] P. J. Ashenden. *The Designer's Guide to VHDL.* Morgan Kaufmann Publishers, 2nd edition, 2001.

[4] J. Circello, G. Edgington, D. McCarthy, J. Gay, D. Schimke, S. Sullivan, R. Duerden, C. Hinds, D. Marquette, L. Sood, A. Crouch, and D. Chow. *The Superscalar Architecture of the MC68060.* Motorola Inc., Microprocessor and Memory Technology Group, 1995.

[5] J. Circello and F. Goodrich. *The Motorola 68060 Microprocessor.* Motorola Inc., Microprocessor and Memory Technology Group, 1993.

[6] M. Delvai. *Entwicklung eines Echtzeitkontrollers für das Echtzeitprotokoll TTP/A.* Master's thesis, TU Wien, Institut für Technische Informatik, 2000.

[7] M. Delvai. *Handbuch für SPEAR, Technical Report.* TU Wien, Institut für Technische Informatik, 2002.

[8] M. Delvai, U. Eisenmann, and W. Huber. Modular Construction System for Embedded Real-Time Applications. In *Proc. Austrochip 2002, Vienna, Austria*, 2002.

[9] M. Delvai, W. Huber, P. Puschner, and A. Steininger. Processor Support for Temporal Predictability - The SPEAR Design Example. In *Proc. 15th Euromicro International Conference on Real-Time Systems, Porto, Portugal*, 2003.

[10] M. Delvai, W. Huber, B. Rahbaran, and A. Steininger. *SPEAR - Design-Entscheidungen für den Scalable Processor for Embedded Applications in Real-Time Environments*, 2001.

[11] die.net. *Definition: embedded systems, http://dict.die.net/*, 2003.

[12] U. Eisenmann. *Design and implementation of a highly efficient communication node for real-time applications.* Master's thesis, TU Wien, Institut für Technische Informatik, 2002.

[13] El Camino GmbH. *Digilab 20Kx240 Manual*, 2003.

[14] W. Elmenreich and S. Pitzek. Smart transducer- principles, communications and configurations. In *Proc. 7th IEEE International Conference on Intelligent Engineering Systems (INES), Assuit, Egypt*, 2003.

[15] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. *The Microarchitecture of the Pentium 4 Processor*. Desktop Platforms Group, Intel Corp., 2001.

[16] W. Huber. Spezifikation der Schnittstelle zwischen Extension-Modulen und SPEAR. Technical report, Institute for Technical Computer Science, VLSI - Design, Vienna, 2001.

[17] Institute of Electrical and Electronics Engineers, Inc. *Standard 1076, IEEE Standard VHDL Language Reference Manual*, 1987.

[18] Institute of Electrical and Electronics Engineers, Inc. *Standard 1451.2-1997, IEEE Standard for a Smart Transducer Interface for Sensors and Actuators - Transducer to Microprocessor Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats*, 1997.

[19] Intel. *Embedded Pentium® Processor - Datasheet*, 1998.

[20] Intel. *Embedded Pentium® Processor Family - Developer's Manual*, 1998.

[21] Intel Corporation. *Hyper-Threading Technology , http://www.intel.com/technology/hyperthread/*, 2003.

[22] Intel Corporation. *IA-32 Intel® Architecture Optimization Reference Manual, http://www.intel.com/design/pentium4/manuals/245472.htm*, 2003.

[23] Intel Corporation. *Intel Pentium 4 processor website, http://www.intel.com/design/intarch/pentium4/pentium4.htm*, 2003.

[24] H. Kopetz. *Real-Time Systems Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, 1997.

[25] C. R. Moore. *The PowerPC 601 Microprocessor*. Advanced Workstations and Systems Division, International Business Machines Corporation, 1993.

[26] Motorola Inc. *MC68060 - Superscalar 32-Bit Microprocessors - Product Brief*, 1994.

[27] D. A. Patterson and J. L. Hennessy. *Computer Architecture a Quantitative Approach*. Morgan Kaufman Publishers, 2nd edition, 1996.

[28] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design*. Morgan Kaufman Publishers, 2nd edition, 1998.

[29] Philips Semiconductor. *An Introduction to Very Long Instruction Word (VLIW) computer architectures*, 1993.

[30] P. Puschner and A. Burns. Writing temporally predictable code. In *Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, 2002.

[31] C. E. Salloum. *Realisierung eines generischen Online Debuggers für Embedded Systems.* Master's thesis, TU Wien, Institut für Technische Informatik, 2003.

[32] M. Schutti. Deep Submicrion (DSM) Effects. In *Proc. Austrochip 2003, Linz, Austria*. DICE.

[33] Semiconductor Industry Association. *The National Technology Roadmap For Semiconductors*, 1997.

[34] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. In *Proceedings of the IEEE, VOL. 83, NO. 12*, 1995.

[35] J. E. Smith and S. Weiss. *PowerPC 601 and Alpha 21064: A Tale of Two RISCs.* Tel Aviv University, 1994.

[36] Synopsys Inc. *VHDL Simulation Reference Manual*, 2000.

[37] R. Tomasulo. *An efficient algorithm for exploiting multiple arithmetic units. IBM Journal of Research and Development, 11(1):25-33*, 1967.

[38] I. Wenzel. *Principles of Timing Anomalies in Superscalar Processors.* Diplomarbeit, TU Wien, Institut für Technische Informatik, 2003.