Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

DIPLOMARBEIT

SPEAR2 - An Improved Version of SPEAR

ausgeführt am Institut für

Technische Informatik, Embedded Computing Systems Group
Technische Universität Wien

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

von

Martin Fletzer
Kreuzgassee 6A
2130 Mistelbach

Mistelbach, 8. Februar 2008 _____

Für meine Eltern

Danksagung

Bedanken möchte ich mich besonders bei Martin Delvai für die Unterstützung bei der Erstellung und die umfassende und gute Betreuung bei dieser Arbeit. Ich möchte mich auch bei allen Professoren und Lehrbeauftragten, sowie allen sonstigen Mitarbeitern der TU Wien für die gute Ausbildung bedanken.

Der größte Dank gilt meinen Eltern, die mir diese Ausbildung ermöglicht haben. Auch bei meiner Freundin möchte ich mich für die vielen mit Geduld ertragenen Entbehrungen während meines Studiums herzlich bedanken.

Kurzfassung

Ein Soft-core Prozessor ist ein konfigurierbarer Mikrocontroller der mit einer Hardwarebeschreibungssprache definiert wurde. Solche Prozessoren können für einfach Systeme angepasst werden, deren Aufgabe es ist I/O-Schnittstellen zu steuern. Sie können aber auch für komplexe Systeme geeignet sein, die ein Betriebssystem und Schnittstellen wie Ethernet oder DDR-SDRAM benötigen. Im Rahmen dieser Diplomarbeit wurde der Soft-core Prozessor SPEAR2 entwickelt. Die SPEAR2 Architektur ist ein 16/32 Bit Prozessor und basiert auf SPEAR (Scalable Processor for Embedded Applications in Real-time Environments). Der Vorgnger wurde am *Institut für Technische Informatik - Embedded Computing Systems Group* and der *Technischen Universität Wien* entwickelt.
Es gab mehrere Gründe eine verbesserte Version zu entwickeln: Den Code für neue Zieltechnologien anpassen, einige Nachteile von SPEAR entfernen, Konfigurierbarkeit unterstützen oder um nützlich Funktion zu ergänzen, wie zum Beispiel byteweise adressierter Speicher.

Um diese Ziele zu erreichen, wurde SPEAR2 von Grund auf neu geschrieben. Um leichtes konfigurieren von Speicher und Datenpfad zu ermöglichen, wurde ein eigene Art der Konfiguration definiert. Grundsätzlich handelt es sich bei SPEAR2 um eine 16 Bit Architektur. Der Datenpfad kann jedoch auf 32 Bit erweitert werden. Großer Aufwand wurde betrieben, damit die beiden unterschiedlich breiten Datenpfade korrekt mit den anderen Komponenten zusammenarbeiten. Die gröte Schwierigkeit war ein einheitlicher Speicherzugriff sowie eine einheitliche Schnittstelle zu den externen Modulen. Ein vorrangiges Ziel während der Entwicklung war die Verwendung der gleichen Befehle und Werkzeuge für beide Datenpfad Konfigurationen.
Obwohl beide Prozessoren beinahe die selben Befehle verwenden, verfügen beide über unterschiedliche Eigenschaften. Der erweiterte Datenpfad ermöglicht einen höheren Durchsatz und größeren Adressbereich, jedoch erhöht sich der Ressourcenbedarf um ca. 70 Prozent.

Abstract

A soft core processor is a configurable microcontroller defined in software. Such processors can be adapted to be appropriate for a simple system, where the only functionalities are the manipulation of general purpose I/O. Moreover, they may also fit a complex system, where an operating system and interfaces like Ethernet or DDR-SDRAM are required.
In the course of this master thesis, the soft core processor SPEAR2 was developed. The SPEAR2 architecture is a 16/32-bit processor and is based on SPEAR (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 motives for developing an improved version were versatile: fitting the code to new target technologies, eliminating some disadvantages of SPEAR, enabling configurability, or just adding useful features like byte addressed memory.

In order to satisfy these goals, SPEAR2 was written from scratch. A configuration framework was created to provide adjustable memory sizes and the option to change the width of the data path. Basically SPEAR2 is a 16-bit architecture, but the data path can be extended to 32 bit. Considerable effort had to be done to enable the correct interaction of two different data path widths with other components of the processor. The chief difficulty was attaining a uniform memory access as well as a uniform bus interface to extension modules for both configurations. Using the same instructions and the same toolchain for both configurations was a priority objective during development.
Although both processor cores have nearly the same ISA the resulting characteristics of the 16-bit and 32-bit version are quite different. The extended data path width enables higher performance and larger address space, but increases resource consumption by about 70 percent.

Contents

List of Figures

List of Tables

1 Introduction

In our daily life embedded systems play a more and more important role. A modern car for example contains between 50 and 100 embedded systems and the usage is rapidly growing. They are used in products even where we do not expect them. These products range from toys through home appliances to airplanes - from simple and uncritical to complex and highly critical applications.
The operational areas of embedded systems are very different. To provide the required flexibility, an embedded system comprises hardware (e.g. processing unit, sensors, communication interfaces, etc.) and software. The combination enables the possibility to customize embedded systems to meet different requirements.

1.1 Motivation

The requirements to embedded systems are very diverse. Thus there is a great diversity of available components. When starting a new embedded systems project an appropriate processor has to be chosen. For small applications a cheap 8-bit standard microcontroller can be sufficient. If much computational power is needed and no specific features are required (e.g. untypically high quantity of I/O pins or more UARTs than usual), then a dedicated processing unit (e.g. 32-bit embedded processors or digital signal processors (DSP)) may satisfy the needs. Between these two extremes various requirements specifications are possible. The drawback of such dedicated hard cores is their limited flexibility.
The opposite of a dedicated hard core is a soft core processor. A soft core processor is a microprocessor defined in software using a Hardware Description Language (HDL). The soft core processor can be synthesized and run in Field Programmable Gate Array (FPGA) or Application Specific Integrated Circuit (ASIC). Soft core processors are very flexible and can be configured with exactly what is needed - no more, no less. Thereby it is possible to tune the processor for less area or more performance. For example, the same processor can be used with or without caches and different number of pipeline

stages - flexibility, unreachable by a dedicated hard core processor.
In addition, the hardware used to implement the soft core processor can be used to implement any parts of the intended task for optimal design implementation. In general, implementing an algorithm only in software is a flexible solution and saves development time. On the other hand, implementing an algorithm in hardware can enable great performance improvements and at the same time requires less energy to fulfil the task. Some algorithms can be accelerated up to 100 times, if implemented in hardware. Together, implementing some parts of a problem in software and the other parts in hardware is a powerful solution and sometimes the only choice. Video compression in real time for example requires a lot of computational power. If low power consumption is required, the compression algorithm has to be implemented partially in hardware. Another big advantage of FPGAs and ASICs is their flexible interface. Nearly all known interfaces are realizable. Starting with a small UART through to PCI-Express interface, packages are available with up to several hundred pins and in a variety of sizes. The advantage of FPGAs over ASICs is their flexibility since an FPGA can be updated like software. On the other hand developing a design for ASICs takes more time and money, but ASICs provide much more performance than an FPGA.

The concept of using a soft core processor has several advantages. For example, only one chip is necessary and thereby the board layout can be simplified which results in a cheaper design. Soft core processors can be customized to provide the required performance without wasting resources. This can be achieved by extending the processor with special functions like a floating point unit or an encryption core. All needed interfaces can be provided by the FPGA/ASIC whereby the processor is very flexible.

The objective of this master thesis is the soft core processor SPEAR2. The processor bases on its predecessor SPEAR [3, 4, 5] and was newly-developed. There was some reasons why we developed a successor. The most critical factor was the use of asynchronous memory by SPEAR, because this type of memory is not supported by new FPGAs. But synchronous memory requires

an additional pipeline stage. We also tried to find a solution for the low I/O throughput capability of SPEAR, when accessing extension modules. Since elementary parts of the processor had to be changed, we decided to develop a new soft core processor. All the experience gathered with SPEAR helped us to develop an efficient and attractive soft core processor.

1.2 Outline

The next chapter gives an overview of state of the art soft core processors. For this purpose three different processor architectures are presented. The subsequent chapter gives detailed information about SPEAR, the predecessor of SPEAR2. Afterwards, the drawbacks of SPEAR are analysed and possible solutions are provided. The succeeding part of the master thesis is about SPEAR2. First the architecture and implementation of SPEAR2 are explained in detail. A separate chapter focuses on the specification of SPEAR2. The specification comprises the instruction set architecture and the hardware interface. Then the results of this master thesis are presented. Finally the conclusion is drawn.

2 State of the Art

In this chapter three different processor architectures are introduced to give an overview of available soft core processors. These soft core processors where chosen since they are widely used and comparable to SPEAR2 regarding resource usage and provided features. At the end of this chapter, the soft core processors are compared regarding some basic features.
Chosen soft core processors:

MicroBlaze: A soft core processor developed by Xilinx

Nios II: A soft core processor developed by Altera

LatticeMico32: A soft core processor developed by Lattice Semiconductor

The main business of the three companys is developing FPGAs. That's why every company optimized its soft core processor for its own FPGA platform. Since the source code for MicroBlaze and Nios II is not available, it is nearly impossible to use one of these soft core processors on different hardware. Only LatticeMico32 is available for free with an open IP core licensing, enabling easy adaption of the processor. All processors are 32-bit architectures and highly configurable. All three companies provide a toolchain for their processor.
There are several reasons why no 16-bit processor was chosen for introduction. First of all, 16-bit soft core processors are hardly used. Often they are not state of the art, or no sufficient toolchain is available. Nios for example - the predecessor of Nios II - represented a 16-bit processor but is not recommended for new projects.
8-bit processors are available, but not comparable to SPEAR2. Every company mentioned above has an 8-bit soft core processor in its product line.
Since SPEAR2 is positioned between 16-bit and 32-bit processors only 32-bit soft core processors were chosen.
The information about the three soft core processors rests basically on three documents. The MicroBlaze Processor Reference Guide [13] was used as main information source for MicroBlaze, the Nios II Processor Reference Handbook

[1] for information about Nios II, and the LatticeMico32 Processor Reference Manual [7] for the LatticeMico32 soft core processor.

2.1 MicroBlaze

Is an embedded processor soft core developed by Xilinx. It is a reduced instruction set computer (RISC) optimized for implementation in Xilinx FPGAs.

2.1.1 Overview

The MicroBlaze soft core processor is highly configurable, allowing to select a specific set of features required by design.
The processor's fixed features set includes:

- Thirty-two 32-bit general purpose registers
- 32-bit instruction word with three operands and two addressing modes
- 32-bit address bus
- Single issue pipeline

In addition to these fixed features, the MicroBlaze processor is parameterized to allow selective enabling of additional functionality. A subset of the optional features is listed below:

- The processor pipeline depth can be 3 or 5.
- Several buses are supported. Namely the On-chip Peripheral Bus (OPB) and the Local Memory Bus (LMB).
- To provide more performance, a hardware barrel shifter and/or divider can be enabled.
- Separate instruction and data caches are supported.
- If required, a floating point unit (FPU) can be used.

- Hardware debug logic can be added.

A complete list of available optional features is given in the MicroBlaze processor reference guide.

2.1.2 Instruction Set Architecture

All MicroBlaze instructions are 32 bits and are defined as either Type A or Type B. Type A instructions have up to two source register operands and one destination register operand. Type B instructions have one source register and a 16-bit immediate operand (which can be extended to 32 bits by preceding the Type B instruction with an IMM instruction). Type B instructions have a single destination register operand. Instructions are provided in the following functional categories: arithmetic, logical, branch, load/store and special.

2.1.3 Registers

MicroBlaze has an orthogonal instruction set architecture. It has thirty-two 32-bit general purpose registers and in addition up to eighteen 32-bit special purpose registers, depending on configuration options. Special purpose registers are used for handling exceptions or floating point unit.
Overview of general purpose registers:

- Register 0 is defined to always have the value of zero. Anything written to register 0 is discarded.
- Registers 1 through 13 are general purpose registers.
- Register 14 is used to store return addresses for interrupts.
- Register 15 is recommended for storing return addresses for user vectors.
- Register 16 is used to store return addresses for breaks.
- Register 17 through 31 are general purpose registers.

2.1.4 Pipeline Architecture

MicroBlaze instruction execution is pipelined. For most instructions, each stage takes one clock cycle to complete. Consequently, the number of clock cycles necessary for a specific instruction to complete is equal to the number of pipeline stages, and one instruction is completed on every cycle.
A few instructions require multiple clock cycles in the execute stage to complete. This is achieved by stalling the pipeline. Two different pipeline configurations are supported by MicroBlaze:

- Three Stage Pipeline - When area optimization is enabled, the pipeline is divided into three stages to minimize hardware cost: Fetch, Decode, and Execute.
- Five Stage Pipeline - When area optimization is disabled, the pipeline is divided into five stages to maximize performance: Fetch, Decode, Execute, Access Memory, and Writeback.

Branches

Normally the instructions in the fetch and decode stages are flushed when executing a taken branch. The fetch pipeline stage is then reloaded with a new instruction form calculated branch address. A taken branch in MicroBlaze takes three clock cycles to execute, two of which are required for refilling the pipeline. To reduce this latency overhead, MicroBlaze supports branches with delay slots.

Delay Slots

When executing a taken branch with delay slot, only the fetch pipeline stage in MicroBlaze is flushed. The instruction in the decode stage (branch delay slot) is allowed to complete. This technique effectively reduces the branch penalty from two clock cycles to one.

2.1.5 Memory Architecture

MicroBlaze is implemented with a Harvard memory architecture, i.e. instruction and data access are done in separate address spaces. Each address

space has a 32 bit range (i.e. handles up to 4 gigabytes of instruction and data memory respectively).

Memory Organization
Both instruction and data interfaces of MicroBlaze are 32 bit wide and use big endian, bit-reversed format.
Data access must be aligned (i.e. word access must be on word boundaries, halfword on halfword boundaries), unless the processor is configured to support unaligned exceptions. All instruction access must be word aligned.

Memory and Peripheral Access
MicroBlaze does not separate data access to I/O and memory (i.e. it uses memory mapped I/O). The processor has up to three interfaces for memory access: Local Memory Bus (LMB), On-Chip Peripheral Bus (OPB), and Xilinx CacheLink (XCL). MicroBlaze supports word, halfword, and byte access to data memory.
MicroBlaze has a single cycle latency for access to local memory (LMB) and for cache read hits, except with area optimization enabled when data side access and data cache read hits require two clock cycles. A data cache write normally has two cycles of latency.

2.1.6 Exceptions

MicroBlaze supports reset, interrupt, user exception, break, and hardware exceptions. Exception vectors are located at the first memory addresses.

Reset
When a reset occurs, MicroBlaze flushes the pipeline and starts fetching instructions from the reset vector.

Hardware Exceptions
MicroBlaze can be configured to trap the following internal error conditions: illegal instruction, instruction and data bus error, and unaligned access.

Interrupt
MicroBlaze supports one external interrupt source. On an interrupt, the instruction in the execution stage completes, while the instruction in the decode stage is replaced by a branch to the interrupt vector.

User Vector (Exception)
A user exception is caused by inserting a exception instruction in the software flow.

2.2 Nios II

Is a general-purpose RISC processor core, developed by Altera. The Nios II processor is a configurable soft-core processor, as opposed to a fixed, off-the-shelf microcontroller. In this context, "configurable" means that you can add or remove features on a system-by-system basis to meet performance or price goals.

2.2.1 Overview

A Nios II processor system is equivalent to a microcontroller or "computer on a chip" that includes a processor and a combination of peripherals and memory on a single chip. The term "Nios II processor system" refers to a Nios II processor core, a set of on-chip peripherals, on chip memory, and interfaces to off-chip memory, all implemented on a single Altera device. Like a microcontroller family, all Nios II processor systems use a consistent instruction set and programming model. The Nios II processor is optimized for implementation is Altera FPGAs, providing:

- Full 32-bit instruction set, data path, and address space
- 32 general-purpose registers
- 32 external interrupt sources
- Single-instruction $32 * 32$ multiply and divide
- Floating-point instructions for single-precision floating-point operations

- Single-instruction barrel shifter
- Hardware-assisted debug module enabling processor start, stop, step and trace

The Nios II architecture describes an instruction set, not a particular hardware implementation. A functional unit can be implemented in hardware, emulated in software, or omitted entirely. For example, in control applications that rarely perform complex arithmetic, the division instruction can be chosen to be emulated in software.

2.2.2 Instruction Set Architecture

There are three types of Nios II instruction word format: I-type, R-type, and J-type. The defining characteristic of the I-type instruction-word format is that it contains an immediate value embedded within the instruction word. I-type instructions words contain a 6-bit opcode field, two 5-bit register fields, and a 16 it immediate data field. The defining characteristic of the R-type instruction-word format is that all arguments and results are specified as registers. R-type instructions contain a 6-bit opcode field, three 5-bit register fields, and an 11-bit opcode-extension field. J-type instructions contain a 6-bit opcode field and a 26-bit immediate data field. J-type instructions, such as call and jmpi, transfer execution anywhere within a 256 MByte range.

2.2.3 Registers

The Nios II architecture supports a flat register file, consisting of thirty two 32-bit general-purpose registers, and up to thirty two 32-bit control registers. The control registers can be used for interrupt handling or hold an unique processor ID. The architecture supports supervisor and user modes that allow system code to protect the control registers from errant applications.
Overview of general purpose registers:

- Register 0 always returns the value zero, and writing has no effect.
- Register 29 through 31 are used to store return addresses.

2.2.4 Pipeline Architecture

Nios II instruction execution is pipelined. Some instructions require multiple clock cycles in the execute stage to complete. The consequent data conflict is solved by stalling the pipeline. Three different pipeline configurations are supported by Nios II.

One Stage Pipeline
When only one pipeline stage is used, a single instruction is dispatched at a time, and the processor waits for an instruction to complete before fetching and dispatching the next instruction. Because each instruction completes before the next instruction is dispatched, branch prediction is not necessary. This greatly simplifies the consideration of processor stalls. Maximum performance is one instruction per six clock cycles.

Five Stage Pipeline
A 5-stage pipeline provides a trade off between performance and hardware cost. Up to one instruction is dispatched and/or retired per cycle. Instructions are dispatched and retired in-order. Static branch prediction is implemented. The pipeline is divided into five stages: Fetch, Decode, Execute, Memory, and Writeback.

Six Stage Pipeline
A 6-stage pipeline should be used, if maximum performance is required. Up to one instruction is dispatched and/or retired per cycle. Instructions are dispatched and retired in-order. Dynamic branch prediction is implemented using a 2-bit branch history table. The pipeline is divided into three stages to maximize performance: Fetch, Decode, Execute, Memory, Align, and Writeback.

2.2.5 Memory Architecture

The flexible nature of the Nios II memory organization is the most notable difference between Nios II processor systems and traditional microcontrollers.

A Nios II core uses one or more of the following to provide memory access:

- Instruction master port
- Instruction cache
- Data master port
- Data cache
- Tightly-coupled instruction or data memory port - Interface to fast on-chip memory outside the Nios II core

The Nios II architecture supports separate instruction and data buses, classifying it as a Harvard architecture. The data master port connects to both memory and peripheral components, while the instruction master port connects only to memory components.

Instruction Master Port
The instruction master port performs a single function: it fetches instructions to be executed by the processor. The instruction master port is pipelined which minimize the impact of synchronous memory with pipeline latency and increases the overall maximum frequency of the system. The instruction master port can issue successive read requests before data has returned from prior requests. The Nios II processor can prefetch sequential instructions and perform branch prediction to keep the instruction pipe as active as possible.

Tightly-coupled Memory
Tightly-coupled memory provides guaranteed low-latency memory access for performance-critical applications. Compared to cache memory, tightly-coupled memory provides the following benefits:

- Performance similar to cache memory
- Software can guarantee that performance-critical code or data is located in tightly-coupled memory No real-time caching overhead, such as loading, invalidating, or flushing memory

Physically, a tightly-coupled memory port is a separate master port on the Nios II processor core, similar to the instruction or data master port. A Nios II core can have zero, one, or multiple tightly-coupled memories for both instruction and data access. Each tightly-coupled memory port connects directly to exactly one memory with guaranteed low, fixed latency.

Memory Organization

Nios II addresses are 32 bits, allowing access up to a 4 gigabyte address space. However, many Nios II core configurations restrict addresses to 31 bits or fewer. The Nios II architecture is little endian. Words and halfwords are stored in memory with the most significant bytes at higher addresses. Contents in memory are aligned as follows:

- A function must be aligned to a minimum of 32-bit boundary.
- The minimum alignment of a data element is its natural size, which can be 8, 16, or 32 bit. A data element larger than 32-bits need only be aligned to a 32-bit boundary.

Memory and Peripheral Access

The Nios II architecture provides memory-mapped I/O access. Both data memory and peripherals are mapped into the address space of the data master port. The processor's data bus is 32 bits wide. Instructions are available to read and write byte, half-word (16-bit), or word (32-bit) data.

2.2.6 Exceptions

The Nios II architecture provides a simple, non-vectored exception controller to handle all exception types. All exceptions, including hardware interrupts, cause the processor to transfer execution to a single exception address. The exception handler at this address determines the cause of the exception and dispatches an appropriate exception routine. The exception address is specified at system generation time.

Integral Interrupt Controller

The Nios II architecture supports 32 external hardware interrupts. The processor core has 32 level-sensitive interrupt request (IRQ) inputs, providing a unique input for each interrupt source. IRQ priority is determined by software. The architecture supports nested interrupts.

The software can enable and disable any interrupt source individually through the *ienable* control register, which contains an interrupt-enable bit for each of the IRQ inputs. Software can enable and disable interrupts globally using the PIE bit of the status control register. A hardware interrupt is generated if and only if all three of these conditions are true:

- The PIE bit of the status register is 1
- An interrupt-request input is asserted
- The corresponding bit of the *ienable* register is 1

Interrupt Vector Custom Module

The Nios II processor core offers an interrupt vector custom module which accelerates interrupts vector dispatch. Every interrupt source gets its own interrupt vector.

The interrupt vector custom module is based on a priority encoder with one input for each interrupt connected to the Nios II processor. The cost of the interrupt vector custom module depends on the number of interrupts connected to the Nios II processor.

If a large number of interrupts is used, adding the interrupt vector module might lower the maximum frequency.

2.3 LatticeMico32

Is a configurable 32-bit soft processor core for Lattice FPGA devices. The processor is available for free with an open IP core licensing, enabling easy adaption of the processor.

2.3.1 Overview

With separate instruction and data buses, this Harvard architecture processor allows for single-cycle instruction execution as the instruction and data memories can be accessed simultaneously. Additionally, the LatticeMico32 uses a RISC architecture, thereby providing a simpler instruction set and faster performance. As a result, the processor core consumes minimal device resources, while maintaining the performance required for a broad application set. Some of the key features of the 32-bit processor include:

- RISC architecture
- 32-bit data path
- 32-bit instructions
- 32 general-purpose registers
- Up to 32 external interrupts
- Optional instruction cache
- Optional data cache
- Dual WHISHBONE memory interfaces (instruction and data)

To accelerate the development of processor systems, several optional peripheral components are available with the LatticeMico32 processor. Specifically, these components are connected to the processor through a WISHBONE bus interface, a royalty-free, public-domain specification.

2.3.2 Instruction Set Architecture

All LatticeMico32 instructions are 32 bits wide. They are in four basic formats, as listed below:

- Register Immediate (RI) Format:
- Register Register (RR) Format

- Control Register (CR) Format
- Immediate (I) Format

LatticeMico32 supports a variety of instructions for arithmetic, logic, data comparison, data movement, and program control. Not all instructions are available in all configurations of the processor. Support for some types of instructions can be eliminated to reduce the amount of FPGA resources used.

2.3.3 Registers

The LatticeMico32 processor has thirty-two 32-bit general purpose registers and several control and status registers.
Overview of general purpose registers:

- Register 0 must always hold the value 0.
- Registers 1 through 28 are truly general purpose and can be used as the source or destination register for any instruction.
- Register 29 is used by the call instruction to save the return address but is otherwise general purpose.
- Register 30 is used to save the value of the program counter when an exception occurs.
- Register 31 saves the value of the program counter when a breakpoint or watchpoint exception occurs.

2.3.4 Pipeline Architecture

The LatticeMico32 processor uses a 6-stage pipeline. It is fully bypassed and interlocked. The bypass logic is responsible for forwarding results back through the pipeline, allowing most instructions to be effectively executed in a single cycle. The interlock is responsible for detecting read-after-write hazards and stalling the pipeline until the hazard has been resolved. This avoids

the need to insert nop directives between dependent instructions, keeping code size to a minimum, as well as simplifying assembler-level programming. The six pipeline stages are:

- Address - The address of the instruction to execute is calculated and sent to the instruction cache.
- Fetch - The instruction is read from memory.
- Decode - The instruction is decoded, and operands are either fetched from register file or bypassed form the pipeline.
- Execute - The operation specified by the instruction is performed. For simple instructions such as addition or a logical operation, execution finishes in this stage, and result is made available for bypassing.
- Memory - For more complicated instructions such as loads, stores, multiplies, or shifts, a second execution stage is required.
- Writeback - Results produced by the instructions are written back to the register file.

2.3.5 Memory Architecture

The LatticeMico32 processor has a flat 32-bit, byte-addressable address space. For LatticeMico32 processors witch caches, the portion of the address space that is cacheable can be configured separately for both the instruction and data cache. This allows for the size of the cache tag RAMs to be optimized to be as small as is required (the fewer the number of cacheable addresses, the smaller the tag RAMs will be).

Memory Organization

The LatticeMico32 processor is a big-endian, which means that multi-byte objects, such as half-word and words, are stored with the most significant byte at the lowest address.
All memory accesses must be aligned to the size of the access, as listed below:

- Byte Access: No address requirements.
- Half-word Access: Address must be half-word aligned (bit 0 must be 0).
- Word Access: Address must be word aligned (bits 1 and 0 must be 0)

No check is performed for unaligned access. All unaligned accesses result in undefined behavior.

2.3.6 Exceptions

The LatticeMico32 processor can raise eight types of exceptions, as listed below:

- Reset: Raised when the processor's reset pin is asserted.
- Breakpoint: Raised when either a break instruction is executed or when a hardware breakpoint is triggered.
- InstructionBusError: Raised when an instruction fetch fails, typically due to the requested address being invalid.
- Watchpoint: Raised when a data access fails, typically because either the requested address is invalid or the type of access is not allowed.
- DivideByZero: Raised when an attempt is made to divide by zero.
- Interrupt: Raised when one of the processor's interrupt pins is asserted, providing that the corresponding field in the interrupt mask is set and the global interrupt enable flag is set. The LAtticeMico32 processor supports up to 32 active-low, level-sensitive interrupts.
- SystemCall: Raised when an *scall* instruction is executed.

Exceptions occur in the execute pipeline stage. If there is an instruction in the memory pipeline stage, that instruction is first allowed to finish. All instructions from the execute stage back are then killed and do not cause any user-transparent state changes. For example, no flags are set.

Exception Handler

When an exception occurs, the CPU branches to an address that is an offset from a predefined value. The offset is calculated by multiplying the exception ID by 32. Since all LatticeMico32 instructions are four bytes long, this means each exception handler can be eight instructions long. If further instructions are required, the handler can call a subroutine.

2.4 Comparison

This section is used to give a short comparison of previously explained soft core processors. Table 1 gives an overview of supported features:

	MicroBlaze	Nios II	LatticeMico32
General Purpose Registers	32	32	32
Pipeline Stages	3/5 Stages	1/5/6 Stages	6 Stages
Instruction Cache	Optional	Optional	Optional
Data Cache	Optional	Optional	Optional
Floating Point Unit	Optional	Optional	No
Interrupts	1	32	0-32
HW-Debug Support	Optional	Optional	Optional
Hardware multiplier	Optional	Optional	Optional
Hardware divider	Optional	Optional	Optional
Hardware barrel shifter	Optional	Optional	Optional
Memory Management Unit	Optional	No	No

Table 1: Features of State of the Art Soft Core Processors

The reachable clock frequency and resource consumption have not been considered, since these attributes depend strongly on configuration and target technology.

3 SPEAR - Basis for a new Architecture

SPEAR stands for **S**calable **P**rocessor for **E**mbedded **A**pplications in **R**eal-time environments. A detailed description is given in [3, 5, 4]. SPEAR was designed for embedded systems with real-time requirements. Thus the two most important features for such a processor are adaptability and real-time capability.
To enable adaptability, SPEAR is able to use extension modules. The extension modules are mapped to uppermost data memory and can be accessed like normal memory with load/store operations.
The execution time of a given real-time task is influenced by many factors. SPEAR offers a better support for real-time execution by reducing the hardware-jitter to the granularity of one clock cycle. This is achieved by constant execution time of all instructions, deterministic behaviour on interrupt execution, and all data and control hazards of the pipeline are resolved in hardware.

3.1 Overview

SPEAR features 16-bit RISC architecture. That means that all buses and registers are 16 bit wide. The design executes instructions in a 3-stage pipeline. The instruction set comprises 80 instructions. The data memory and instruction memory are both 4 KB in size and are separated (Harvard architecture). The uppermost 1 KB of the data memory is reserved for memory mapping of the extension modules. The register file consists of 32 registers. 26 of them are general purpose registers and 6 registers are used for special functions. Figure 1 shows the structure of the processor.

3.1.1 Pipeline

The pipeline is built by the three stages. First comes the fetch stage which is responsible to read one instruction from the instruction memory. The next stage decodes this instruction and is hence called decode stage. The decoding comprises the generation of control signals for the ALU, generation

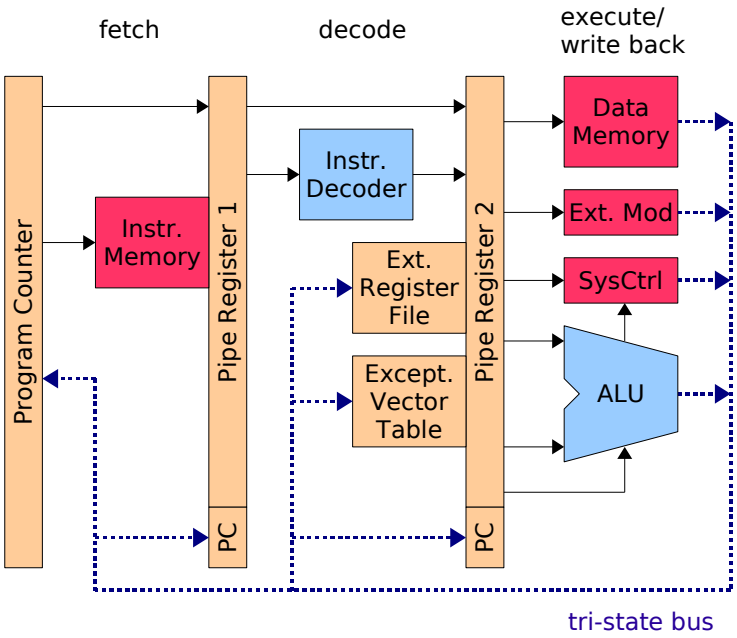


Figure 1: Block Diagram of SPEAR

of immediate values and read out of the operands from the register file. The execute/write-back stage performs the intended operation. This can be an arithmetic/logical ALU operation or an access to the memory respectively to an extension module. The result of a memory read out or ALU operation is afterwards written to the write back bus.
The pipeline supports full data forwarding between pipeline stages to prevent data hazards. In addition also control hazards are resolved by hardware and it is not necessary to insert pipeline stalls. This simplifies the calculation of worst case execution time.

3.1.2 Memory Architecture

The memory architecture was designed to be as simple as possible. In fact, data and instruction memory allow only 16-bit access. Beside the simplified hardware, this approach enables to address as twice as much memory compared to a byte-based memory.

3.2 Exceptions

SPEAR supports two types of exceptions:

1. Interrupts, which are triggered by hardware.
2. Traps, which are triggered by software.

For both types 16 different sources are possible. Therefore 32 exceptions are supported. The addresses of the service routines are stored in the exception vector table, a dedicated memory block located in the decode stage of the SPEAR processor core. Figure 2 depicts the exception vector table, it contains 32 entries, whereas the trap vectors are stored at positive and interrupt vectors are stored at negative positions. There are special instructions to build and manipulate the exception vector table.

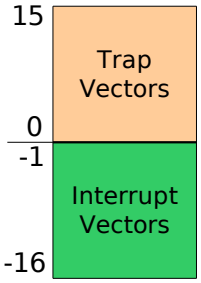


Figure 2: Exception Vector Table of SPEAR

3.3 Register File

The register file holds 32 registers and six of them are special function registers: FPTRX, FPTRY, FPTRZ, RTSX, RTSY, and RTE.

3.3.1 Frame Pointer Registers

These registers are used to build frames. Frames are similar to stacks with the difference that each data element inside the frame can be accessed without emptying the stack. The address of a data element is given by the content

of the frame pointer register plus an offset specified in the instruction. Thus for using frame pointers, first the base address of the frame has to be load to the corresponding register. After that, the frame can be used with the frame pointer instructions. These instructions can hold a 5 bit offset. Hence a frame comprises 32 words. Figure 3 shows a frame.

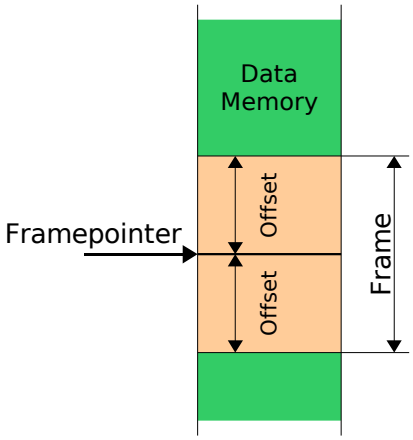


Figure 3: Organization of a Frame

Frames are useful for accessing extension modules where some adjacent addresses will be accessed several times. The registers r26 (FPTRX), r27 (FPTRY) and r28 (FPTRZ) are used for this purpose, allowing to manage three independent frames contemporaneously.

3.3.2 RTSX- and RTSY-Register

Depending on the subroutine call instruction, SPEAR saves the return address either in register r29 (RTSX) or in r30 (RTSY). Thus allows two subsequent subroutine calls without additional overhead. If more nested subroutine calls are required, the return address has to be saved and restored manually.

3.3.3 RTE-Register

This register is used to save the return address in case of an exception. If nested interrupts are used, the return address has to be saved and restored manually.

3.4 Instruction Set Architecture

The instruction set comprises 80 instructions whereof 32 instructions are implemented as conditional instructions. All instructions are 16 bit and have the same execution time.

3.4.1 Structure of Instructions

The structure of instructions can be described using instruction formats. Some 32 bit architectures like the MIPS architecture uses only two different instruction formats (R-format and I-format). Information about the MIPS architecture is given in [9, 10]. Using only few different instruction formats make the decoding of instructions easier. This was not possible for SPEAR because of the tight instruction set when using only 16-bit instructions. Table 2 shows the instruction formats used by SPEAR. A list of instructions is given in [3].

3.4.2 Conditional Instructions

This type of instructions is required to apply the One-Path programming paradigm [11], which allows to implement programs with a data-independent execution time. For conditional instructions the processor condition flag is used to decide if an instruction is going to be executed. If not, a NOP is inserted and executed instead of the instruction. This provides a constant execution time. The processor condition flag can only be modified by a compare or bit-test instruction and is located in the processor status register. Below an if-statement as example to show the usage of conditional instructions:

```
If (r1 == 3) then
  r2 = r3 + 1;
```

Bit Number															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode						Register					Register				
6 bits						5 bits					5 bits				
Opcode			Constant								Register				
3 bits			8 bits								5 bits				
Opcode						Constant					Register				
6 bits						5 bits					5 bits				
Opcode							Constant				Register				
7 bits							4 bits				5 bits				
Opcode											Register				
10 bits											5 bits				
Opcode					Constant										
5 bits					11 bits										

Table 2: Instruction Formats used by SPEAR

```
else
  r2 = r4;
end if;
```

with conditional instructions

```
cmpi_eq r1,  3  /*r1 equal 3?*/
mov_ct  r2, r3  /*yes => r2=r3*/
addi_ct r2,  1  /*yes => r2=r2+1*/
mov_cf  r2, r4  /*no  => r2=r4*/
```

without conditional instructions

```
cmpi_eq r7, 3
jmpi_cf +4
mov     r2, r3
addi    r2, 1
jmp     +2
mov     r2, r4
```

The implementation on the left side uses conditional instructions and requires 4 instructions. Independent whether the value of register r7 is equal 3 or not, the execution time is always the same. This data-independent constant execution time programming style requires at least a conditional move instruction, described by the One-Path programming paradigm mentioned above. However, conditional instructions can also improve the performance.

Let us take a closer look to the example:
The right side implementation requires 6 instructions and its execution time depends on the value of register r7. In the best case only 3 instructions and in the worst case 5 instructions have to be executed. Because of the 3 pipeline stages two NOPs have to be inserted during the execution of a jump. So without conditional instructions the worst case execution time is 7 cycles. To produce more efficient code with conditional instructions, the if-then-else construct has to contain only few expressions and the required instructions have to be available as conditional instructions. This was the reason, why SPEAR provides more conditional instructions than the pure conditional move instruction.

3.5 Extension Modules

Extension modules are used to adapt the processor for different requirements. To easy the integration of an extension module a generic interface has been defined to be used by all different extension modules and processor cores. The interface consists of 8 registers which are mapped to the data memory. Hence only load and store instructions are needed for accessing an extension module. The first two registers are the status and configuration registers. The status register is defined as read only. The lower 8 bit of these registers are defined by the interface specification, thus they are the same for all extension modules. The upper 8 bit are module specific. The other 6 registers are named DATA 0 to DATA 5 and are used for module specific purpose. Figure 4 shows the interface used by extension modules.
There are three types of extension modules which are classified by their functionality:

System-Extension Modules have an extended interface to the processor and improves the functionality of the processor itself.

Function-Extension Modules can be used to implement operations in hardware which otherwise have to be emulated by software.

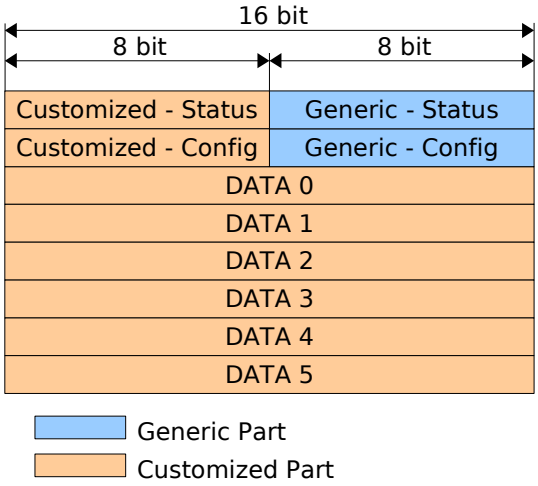


Figure 4: Interface for Extension Modules

IO-Extension Modules used to implement interfaces for the processor like RS232 or PS/2. Only this modules have I/O capability.

Two system-extension modules are worth for mentioning, because they are very important for the processor:

3.5.1 Processor Control Module

The processor control module is mandatory for the correct working of the processor, because it contains the processor status register and the interrupt handler. This module is also responsible for saving the processor status in case of an exception.

3.5.2 Programmer Module

The programmer module is used to store the program code into the processors instruction memory. The program code itself has to be provided by an EEPROM or a PC for example. Thus the programmer module can only be used in cooperation with an I/O-extension module to access the source of program code. Dividing this task to a programming and a communication part enables easy changing of the source since only the communication module has to be replaced.

4 Analysing the Old Architecture

When developing a successor, first the predecessor has to be analysed in order to identify weak points but also to identify proved features.
This chapter describes the first step of the SPEAR2 design process, namely the identification of the weaknesses of the SPEAR architecture. Finally solutions for these weaknesses will be presented.

4.1 Three Processor Cores

Beside the SPEAR processor core two further processor cores, namely NEEDLE and LANCE, were developed at the department. The motivation for the additional processor cores was to achieve scalability with respect to performance. All three cores are fully code compatible and have the same extension module interface.
NEEDLE, a small processor with less performance and small resource consumption. SPEAR, designed for medium requirements. And LANCE, a superscalar 16-bit processor with high performance.
Basically, each one of this processor cores could be used as starting point of SPEAR2. However, the experience showed, that there was no reason for using neither NEEDLE nor LANCE. Indeed, the resource consumption of NEEDLE were only two-thirds compared to SPEAR, but also the maximum clock frequency were nearly halved and executing one instruction took several clock cycles. Therefore the performance was reduced to less than one quarter. In contrast, LANCE needed much more resources than SPEAR and in return doubled the instructions executed per cycle. But through heavily reduced maximum clock frequency the performance of LANCE was not even good as the performance of SPEAR.

4.2 Analysing SPEAR

Since SPEAR is the most efficient processor core compared to NEEDLE and LANCE, it was chosen as the basis for a new processor. However, SPEAR had also several disadvantages which we tried to remove within SPEAR2.

The list below identifies the parts of SPEAR with potential for improvement:

Type of Memory: One problem of SPEAR was the use of asynchronous memory, because this type of memory is not supported by new FPGAs and has to be replaced with synchronous memory.

Additional Pipeline Stage: The maximum clock frequency could easily be improved by braking up the critical path with an additional pipeline stage. Apart from reaching higher clock frequencies an additional pipeline stage would be required anyway because of the registered output of synchronous memory.

Access to Extension Modules: In some situations the throughput between the processor and extension modules was to low. Especially if the processor had to copy data between to extension modules. The solution would be the capability of transferring more than 16 bit at once.

Computational Power: The performance of SPEAR was enough for typical microcontroller tasks. But writing complex algorithms using 32 bit variables was difficult, since 32 bit arithmetic is inefficient on a entire 16 bit processor. It would be nice if SPEAR2 could handle 32 bit values efficient.

Addressable Memory: SPEAR was able to address 128 KB of memory. SPEAR2 should be able to extend the amount of addressable memory.

Memory Organisation: Only 16 bit access was supported by SPEAR. Thus using values smaller than 16 bit was inefficient since byte values wasted half of used memory. And standard tools like the gnu compiler collection or gnu debugger have problems when the memory does not use byte addresses.

Conditional Instructions: Many but not all arithmetic operations were available as conditional instructions. But all arithmetic operations should be supported to utilize the benefit of conditional instructions with respect to performance.

Frame Pointer: They were intended for efficient access to extension modules and accomplished this task very well. But they are no replacement for stack pointers which accelerate sub-routine calls and register swapping considerable.

Write Back Bus: This bus is implemented as tri-state bus which is converted by synthesis tools to multiplexer configurations since FPGAs do not have on-chip tri-state buffers. And large multiplexer structures need much resources and slow down the design. A solution would be to merge the signals by a disjunction.

Shifting: It was only possible to shift a data word for one position. Shifting a specific number of bits at once would improve these arithmetic operations noticeably.

The solution for the first two points will be the use of synchronous memory and an additional pipeline stage. To gain more computational power, to increase the throughput to extension modules and to extend the amount of addressable memory SPEAR2 will dispose of a customizable data path. In return of higher resources usage the data path of SPEAR2 can be enlarged to 32 bit. This feature will be described in detail in Section 5.2. The memory organisation will be changed. 8 bit, 16 bit and 32 bit memory access will be supported. All arithmetic operations will be available as conditional instructions. Real stack pointer will be available for stack management. Shifting operation will support to shift several positions at once.
A more detailed description about the improvements in SPEAR2 is given in Chapter 5.

5 SPEAR2

This chapter gives an overview and treats all design decisions and improvements of SPEAR2. The customizable data path is explained in detail, due to its high impact to the processor core. Afterwards the specification and implementation details are going to be treated.

5.1 Overview

SPEAR2 is the successor of SPEAR and stands for **S**calable **P**rocessor for **E**mbedded **A**pplications in **R**eal-time environments **2**. Although it is the successor and a lot of features are adopted from SPEAR, the code required a complete redesign. SPEAR2 represents a RISC architecture which executes instructions in a pipeline. The pipeline has four stages and supports full data forwarding between stages to prevent data hazards. Also control hazards are resolved in hardware. Thus it is not necessary to insert any pipeline stalls. These feature simplifies the programming and simplifies the prediction of worst case execution time. The memory for data and instructions is separated (Harvard-architecture). The size of the memory is configurable and 1 KB of the data memory is reserved for memory mapping of the extension modules. The instruction set comprises 122 instructions. The width of every instruction is 16 bit and all of them have the same execution time of one cycle. Most of the instructions are conditional ones. The register file holds 16 registers which are split into 14 general purpose and 2 special function registers which are used to save the return address in case of an interrupt or subroutine call. SPEAR2 supports 32 exceptions. 16 of them are hardware exceptions (=interrupts) and 16 can be activated by software (=trap). For building stacks there are four stack pointer available. If the processor is idle, it can be put to sleep mode for energy saving. The processor returns from sleep mode as soon as an interrupt occurs. The most interesting feature is the capability to change the width of the data path. This enables two versions with different performance but same instruction set and interface. Figure 5 shows the structure of the processor.

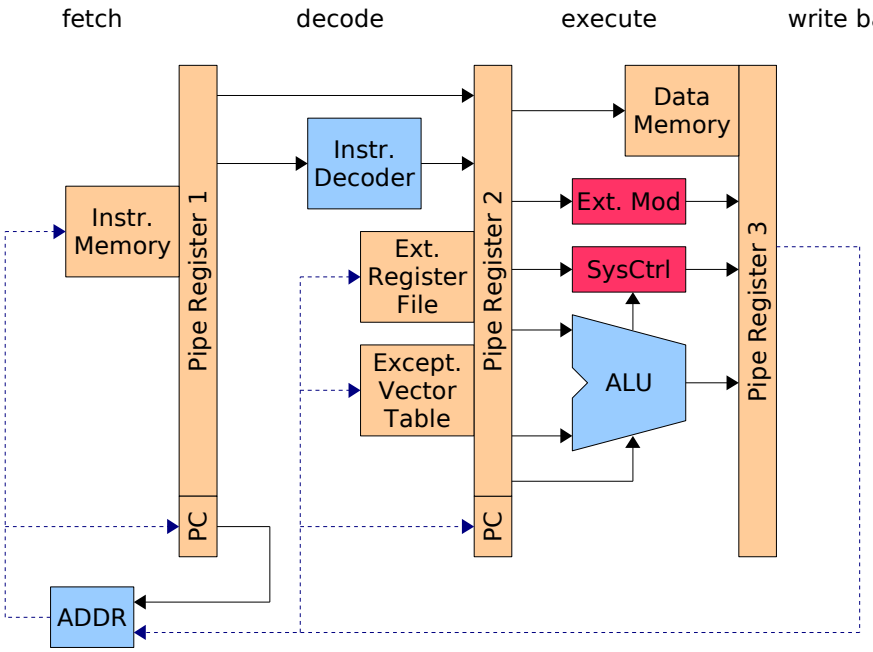


Figure 5: Block Diagram of SPEAR2

5.2 Customizable Data Path

A customizable data path enables the capability of having two versions of the same processor but with different performance. Both versions use the same toolchain, have the same features and an identical interface. Hence it is easy to switch between different configurations even during a running project. For example at the beginning of a project the small version of SPEAR2 is used. After some source code and extension modules have been developed, it becomes clear that the performance is insufficient. If the performance of the 32 bit version is enough, only the configuration of the processor has to be changed and the same source code and extension modules can be used furthermore. The following steps are required to switch the configuration and to acquire more performance:

- First the configuration has to be changed by editing the configuration file of SPEAR2. Afterwards the source code of SPEAR2 has to be

synthesized again.

- The source code of the project has to be recompiled for the 32 bit version.

Since the 32 bit version requires more resources the FPGA must has enough free resources left when switching to 32 bit.

5.2.1 Implementation Overview

Figure 6 shows which parts of SPEAR2 are modified when changing the configuration. Since the majority of parts are affected, the resource requirements are quite different. However, with an enlarged data path the resource requirements are still less compared to an entirely 32 bit implementation. The advantage is not saving some resources, but the big advantage of a customizable data path is the compatibility of both configurations. The same toolchain can be used. The same source code runs on both processors, since the instruction decoder stays unchanged. And no extension modules have to be adapted when changing the configuration.
More information about how the individual parts of the processor are affected by the customizable data path is given in Section 5.5.

5.2.2 Performance Improvement

When SPEAR2 is configured as an entirely 16 bit processor, the performance will be comparable to other 16 bit processors. With an enlarged data path the performance can be improved, because 32 bit values can be handled by the ALU without overhead and the enabled 32 bit access to extension modules. But the performance of other 32 bit processors will not be reached, since the instruction set is still less powerful.

5.2.3 Addressable Memory

Since with an enlarged data path also the addressable memory extends, data path configuration is not only a question about performance. The 16 bit

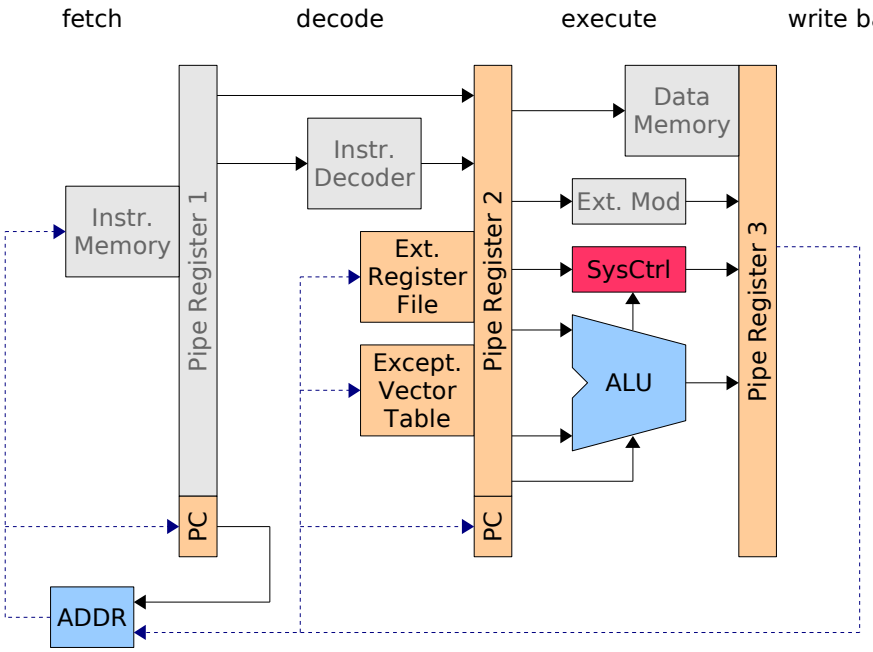


Figure 6: Parts Affected by Configuration

version can address 64 KB memory and the 32 bit version is able to address 4 GB memory.

5.3 Processor Architecture

In this section the architecture of SPEAR2 will be described in detail. The data-flow for every stage and the interaction of the architectural components.

5.3.1 First Stage

The fist stage of the pipeline is called fetch stage. Figure 7 depicts this stage. The only task of this stage is address generation for instruction memories. The program counter is always incremented by one except when a jump is performed. The boot memory is read only and is used at boot time, afterwards it is possible to switch to the instruction memory.

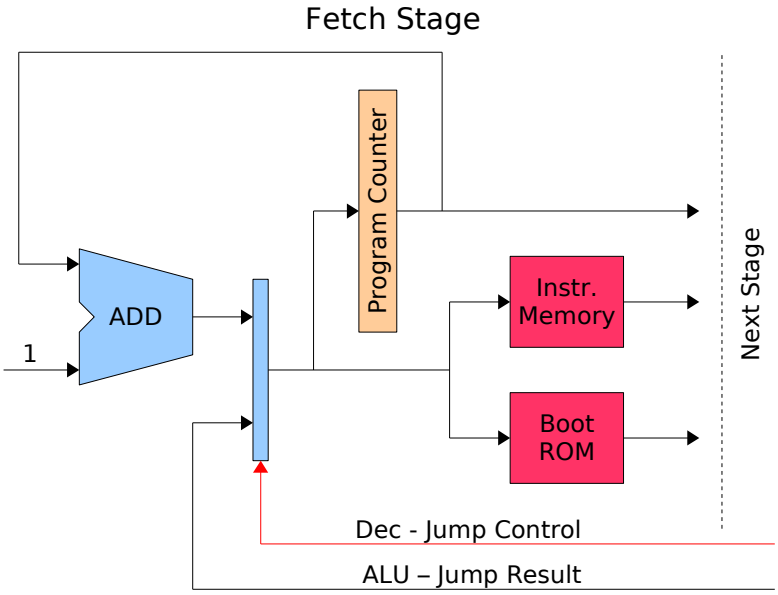


Figure 7: The Fetch Stage in More Detail

The pipe register 1 is build by the program counter an the output of the memories.

5.3.2 Second Stage

This stage is used for decoding the instruction. The stage is depicted in Figure 8. The multiplexer at the input of the decode stage determines whether the instruction to decode is taken from the bootrom or form the instruction RAM. This decision is controlled by the source-selection-signal of the programmer module. The decoding task determines the type of instruction and generates the opcode for the ALU, checks if a jump has to be performed, extracts the immediate value, and generates addresses for the extended register file and the exception vector table. If the processor is currently executing a jump in the execute stage caused by an exception or even by a jump instruction, then the instruction in the decode stage has to be flushed. However, in order to reduce resource usage not all signals are flushed, instead only the control signals are disabled.

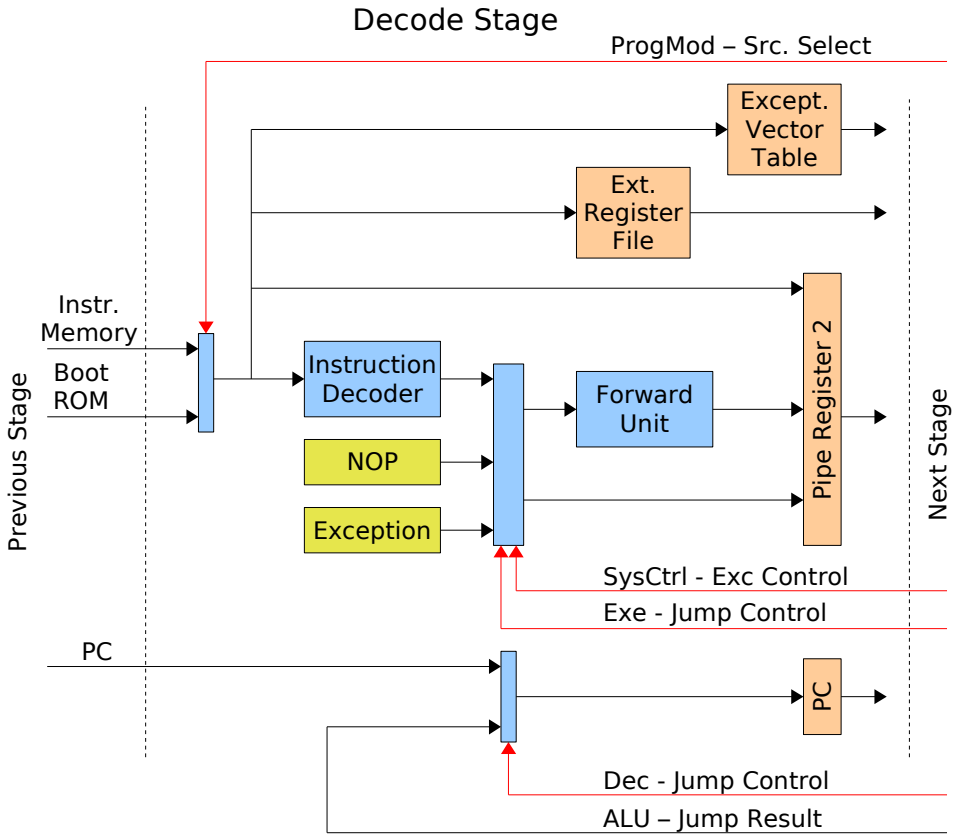


Figure 8: The Decode Stage in More Detail

Depending whether the processor executes a jump or not, either the jump destination address or the program counter of the current instruction will be saved in the pipe register 2. In this way the execution time of a program is constant and does not depend whether the exception happens during a jump or not.

The pipe register 2 consist of the control signals, the output of the exception vector table, the output of the extended register file and the saved program counter.

5.3.3 Third Stage

The third pipe stage executes the instruction. Figure 9 depicts this stage in detail. The data hazards are resolved by the forwarding unit, which is located

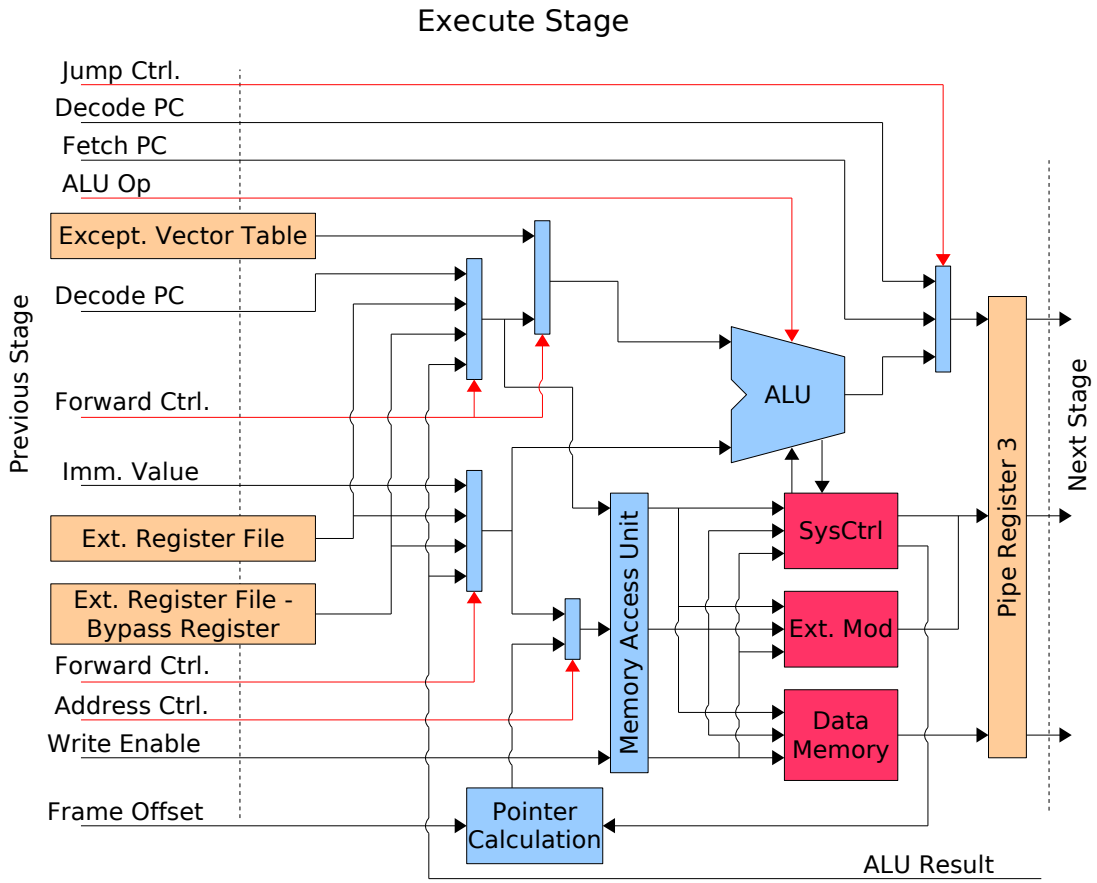


Figure 9: The Execute Stage in More Detail

in the execute stage. The forwarding unit assures that the ALU is always feed with latest information and is described in detail in Section 5.5.5. Instead of an expensive forwarding unit in terms of hardware and delays, stalls could be used to resolve the data hazards. But this solution would make it more complex to predetermine the execution time of a program.
Depending on the executed instruction, different input sources are used as operands. These sources can be: The values of a register read from the

register file, the immediate value encoded in the instruction, the destination address of a jump, or the output of the exception vector table.
In order to reduce the amount of registers in pipe register 3, a multiplexer regulates which value will be saved in pipe register 3: Normally, the result of the ALU will be used, if a jump to subroutine is performed, the program counter of the fetch stage will be used instead because this program counter holds the address from the subroutine call. Due to the fact, that in the case of an interrupt service routine the decoded instruction in replaced by a NOP, the program counter of the decode stage has to be used as return address.
Another task carried out by this stage is the access to data memory and extension modules. The data used for write access is always taken from the forward unit. The applied address can be delivered by the forward unit or frame pointer calculation. Frame pointer are described in detail in Section 5.5.7. By default, the output of an extension module is set to zero. Only when the extension module is selected, the internal signals are propagated to the output. In this way the output of all extension modules can be merged using a simple OR-gate instead of a complex multiplexer.
The pipe register 3 consist of some control signals, the result of the ALU, the merged output of extension modules and the output of data memory.

5.3.4 Fourth Stage

The last stage merges the output of ALU, extension modules, and data memory. Afterwards the result will be written back to the extended register file and the bypass register. The bypass register is used by the forward unit when the same register is read and written within one clock cycle. This makes the implementation independent of the used memory technology. Figure 10 depicts this stage.

5.4 Instruction Set Architecture

This section provides information about design decisions and instruction formats used by SPEAR2. Detailed information about individual instructions is given in Appendix B.

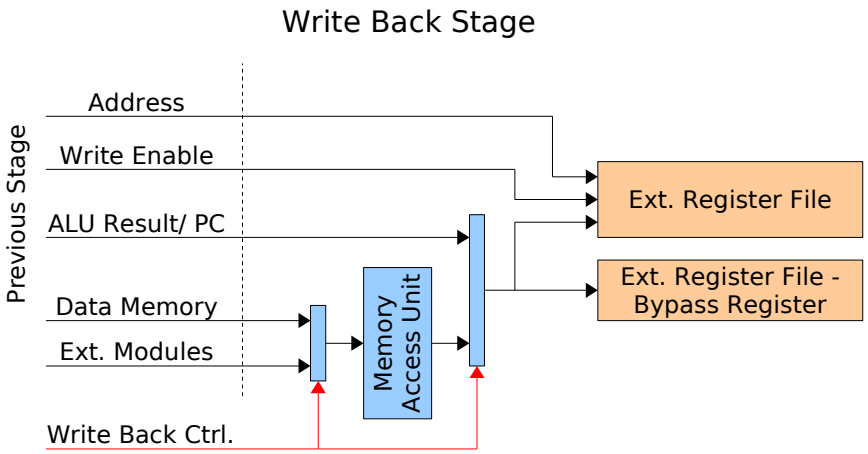


Figure 10: The Write Back Stage in More Detail

All instructions are 16 bit wide. The number of instructions increased from 80 in the original SPEAR processor to 122 in the SPEAR2 processor, whereof 40 instructions are implemented as conditional instructions.

5.4.1 Instruction Format

Five major instruction formats are used. They can be distinguished by their operands:

- Two registers
- Only one register
- One register and an immediate value
- Only one immediate value
- No operand

Normally, a regular structure of the opcodes is desired, where the width of opcode and operands are equal for all instructions. Since SPEAR2 uses only 16-bit instructions, decode space for instructions is limited. Therefore we could not use a regular structure, instead we use variable amount of bits for

opcode and immediate values. The width of an immediate value ranges from 4 to 7 bits and the opcode from 5 to 8 bits.

5.4.2 Conditional Instructions

The characteristics of conditional instructions is unchanged, but the complete instruction set was reworked for more efficient encoding of conditional instructions. Before, 2 bits were used to define one out of three possibilities:

- "00": The instruction will always be executed, independent of the condition flag.
- "11": The instruction will only be executed, if the condition flag is true.
- "10": The instruction will only be executed, if the condition flag is false.
- "01": Not used.

Using the remaining bit combination without rearranging the instruction set would have increased the complexity of the instruction decoder. By smart regrouping of the instructions, it was possible to increase the efficiency of the instruction set without complicating the decoder. Now three bits are used to determine the type of an instruction. The first bit indicates if it is a conditional instruction:

- "0": No conditional instruction
- "1": Conditional instruction

If an instruction was identified as a conditional one, the third and fourth bit are used to determine the exact condition type:

- "00": The instruction will only be executed, if the condition flag is false.
- "01": The instruction will only be executed, if the condition flag is true.

- "10" or "11": The instruction will always be executed, independent of the condition flag.

Beside the increased density of instruction set, the new organisation enables to determine the conditional character of an instruction without decoding the whole instruction and thus can be done concurrently enabling higher clock frequencies.

5.5 Implementation

In this section detailed information about the components of the processor will be given and sometimes differences between SPEAR and its successor are mentioned. The chapter closes with some information about optimization used to decrease resource usage.

5.5.1 Program Counter

Compared to the original SPEAR architecture, a significant improvement was made in SPEAR2 with respect to the program counter. Figure 11 compares both approaches.
Instead of using a dedicated program counter register, the value of the program counter is stored in the pipe register 1. Incremented by 1 it is used as address for the instruction memories. One benefit of this solution is less resource usage. And even better it reduces the latency of a jump due to the fact that the destination address is not buffered as in the original SPEAR architecture but is directly applied to the instruction memories through a multiplexer.
Since the program counter and pipe register 1 have merged, the logical path between program counter and instruction memory is extended. This has no impact to design, because synchronous memories are used for instruction memories and the fetch stage has no time critical path. For correct start up of the processor, the program counter has to be initialized with -1.
The reason why the program counter is incremented by one is given in the next section, which describes the instruction memories and their organisation.

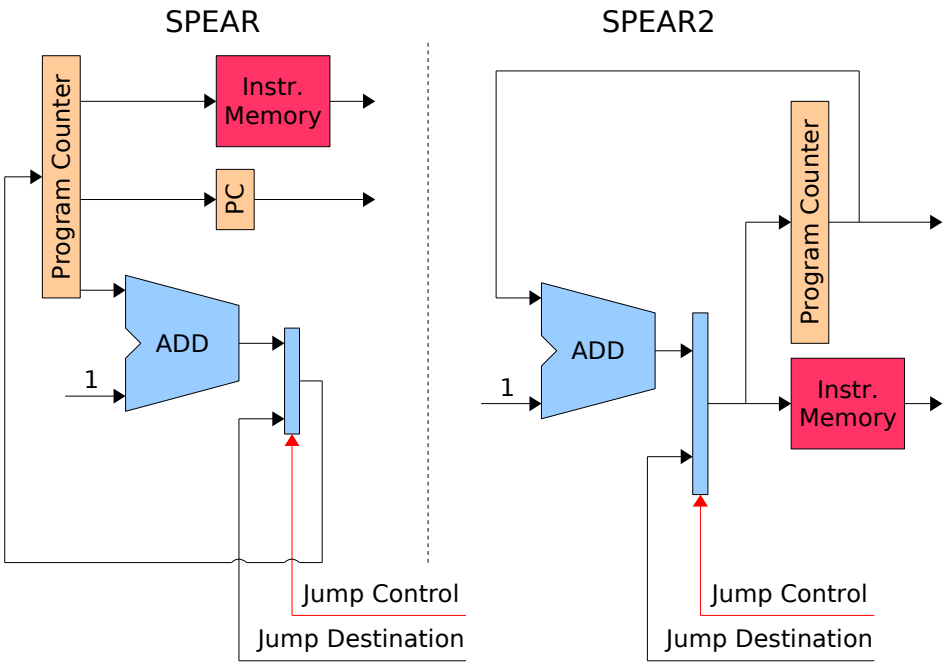


Figure 11: Two different Implementations of the Program Counter

Impact of the Customizable Data Path

The width of the program counter depends on the configuration. It can be 16 bit in the small version and 32 bit in the extended version.

5.5.2 Instruction Memories

The Spear2 architecture provides two memories for instructions: A ROM, called the bootrom because it is used by default at start up. And RAM, called the instruction memory which can be programmed by the programmer module, an extension module described in Section 5.6.2. Which memory is used for instruction fetching is controlled by software. The content of the instruction memory can be changed at run time and thus enables a flexible updating of the program code. This feature makes it possible to reduce time required for software development and testing.

Since all instructions are 16 bit wide and in order to improve the addressable memory space, we decided to use memories with 16-bit data width. Hence,

the program counter has to be incremented by one, after the fetch of a new instruction.
For small and simple applications the program code can be directly stored in the bootrom. Thus the configuration file does not only allow to set the size of the instruction memories but offers also the possibility to disable the instruction RAM, yielding to compacter processor architecture.

Impact of the Customizable Data Path
The customizable data path has no impact on this component.

5.5.3 Decoder

The decoder decodes the instructions and sets all control signals for subsequent pipe stage. For efficiency reasons, the decoder was split into two parts:

- One part of the decoder is small and sets control signals without knowledge of the decoded instruction.
- The other part has to determine which instruction is involved before the control signals can be set.

Control signals which can be set without knowing the type of an instruction are the exception vector table and register file addresses, several control signals for frame pointer and signals used for enabling conditional instructions. This part of the decoder is very simple.
The control signals which depend on the instruction type are again divided into two groups. One group contains control signal which are set and afterwards left unchanged. An example for such a signal is the immediate value. The second group of control signals comprises all control signals which are affected by a flush operation which may be initiated by a jump. In the case of normal operation, this signals are set according to the decoded instruction. If a flush takes place, only this signals have to be disabled and the remaining signals are left unchanged. Examples for such signals are the write enable

signals for the register file and data memory. This reduces the resource usage further, because fewer multiplexer are required and in turn affects the feasible clock rate of the design positively.

Impact of the Customizable Data Path

The width of the generated immediate value depends on the configuration. The 32-bit version provides three additional instructions for the data memory access: load word, load halfword unsigned, and store word. If one of the three instructions is used with the 16 bit configuration, an interrupt will be triggered.

5.5.4 Register File

Compared to the original SPEAR architecture, the number of registers was reduced from 32 to 16 registers. The register file of SPEAR2 comprises 16 registers whereof 14 registers are general purpose. This change was necessary to make the customizable data path possible on the one hand and to design a more efficient instruction set with bigger immediate values on the other hand. To compensate this reduction also the number of special function registers was reduced from 6 to 2. The frame pointer registers were moved into the system control module and SPEAR2 provides only one register to save the return address in case of a subroutine call.[1]
The resulting two special function registers are mapped to the register file of SPEAR2:

- RTS: Saves the return address in case of a subroutine call. The register r14 is used for this purpose. If nested subroutine calls are required, the return address has to be saved and restored manually.
- RTE: This register is used to save the return address in case of an exception. If nested interrupts are used, the return address has to be saved and restored manually.

[1] SPEAR supported 2 registers for saving the return address of subroutine calls, but the second register was only used by programs written in assembler because the compiler was not able to use it.

The register file is realized using two mirrored dual-port memories. Since we needed a memory with three independent ports it was the only solution which is supported by many FPGA technologies.

Impact of the Customizable Data Path
Since the width of the registers depends on the configuration, possible data widths of used memories are 16 and 32 bit.

5.5.5 Forwarding Unit

The forwarding unit was split into two parts. One part generates the control signals and is located in the decode stage. The second part performs the multiplexing and is located in the execute stage in front of the ALU. The distribution over two stages helps to improve the feasible clock rate of the design: Since the multiplexer in front of the ALU affects the critical path, the delay can be reduced if the control signals for the multiplexer are set in advance in the decode stage and afterwards saved in the pipe register 2.
The following sources may be forwarded by the forwarding unit:

- Register File Output: This is the default case, no forwarding is required.
- Pipe Register 3: If the previous instruction used one of our source registers as destination register, it is necessary to use the value stored by the pipe register 3.
- Register File - Bypass Register: This register has to be used if the same register of the register file would be used for read and write at the same time. This situation arises if the instruction two cycles ago used one of our source registers as destination register.
- Program Counter: Has to be used to calculate the destination address of a relative jump. The use of the program counter is controlled by the decoder.

- Immediate Value: Several instructions use an immediate value, which is used as operand for arithmetic operations. The use of the immediate value is also controlled by the decoder.

Usually the program counter and immediate value have nothing to do with the forwarding unit. The reason why they are handled by the forwarding unit is because we tried to conflate multiplexer where possible and to simplify the ALU. This results in a reduced number of ALU operation codes since only one operation code is needed for two versions of the same operation. For example an add operation may use either two registers or one register and an immediate value as operands. Instead of providing dedicated inputs for each operation, the immediate value is provided through the forwarding unit to the ALU. Thus only one add operation has to be implemented inside the ALU.

Impact of the Customizable Data Path
The control logic located at the decode stage is not affected. The width of the multiplexers in front of the ALU depends on the configuration.

5.5.6 ALU

The inputs of the ALU are connected to the two outputs of the forward unit and output of the exception vector table. In order to reduce the amount of different operations the ALU is able to perform, the inputs of the ALU are preprocessed: Setting the internal carry flag and the second operand accordingly or inverted, instructions such as add, add with carry, sub, and sub with carry can be reduced to a single operation inside the ALU. This preprocessing made it possible to reduce the 32 different ALU operation codes in the original SPEAR architecture to 22 operation codes in SPEAR2 which enables a smaller and faster implementation of the ALU.
A feature missed by the old ALU was the ability to shift register content by a specific number of bits at once. It was only possible to shift one bit left or right. Now a barrel shifter is used to perform this operation. This yields a more powerful instruction set for SPEAR2. The barrel shifter is implemented

as a sequence of multiplexers. For example an 8 bit barrel shifter is depicted in Figure 12.

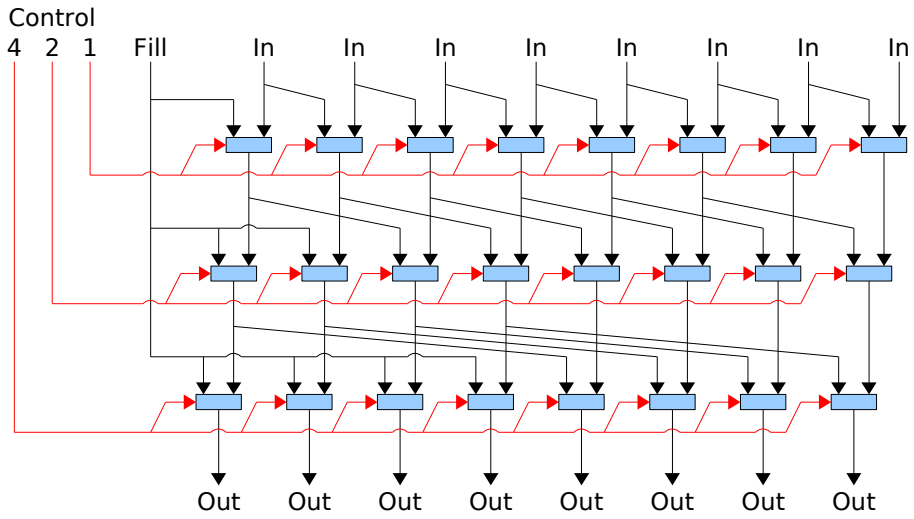


Figure 12: 8 bit Barrel Shifter

The 8 bit barrel shifter is able to shift a data word by 7 bits. To control the 8 bit barrel shifter three control signals are required, one for each multiplexer stage. Another input is used to fill up undefined bits. For shift left operations this bit is zero. For logical shift right operations the bit is zero and for arithmetic shift right operations the value of the most significant bit is used. Spear2 uses two separate barrel shifter: One for shift left and another for shift right operations. The 16-bit version of SPEAR2 requires a 16-bit barrel shifter which has 4 multiplexer stages and the 32-bit version requires a 32-bit barrel shifter with 5 stages and much higher resource usage. The number of required multiplexer can be calculated by $n * log_2(n)$, where n is the number of bits. Hence the 16 bit implementation requires 64 multiplexer, while the 32 bit barrel shifter requires 160 multiplexer.
The use of a barrel shifter is one reason for the high resource usage and lower maximum clock frequency of the 32-bit version compared to the 16-bit version of SPEAR2.

Impact of the Customizable Data Path
Since the ALU is an integral part of the data path, it is strongly affected by the configuration of the customizable data path. Depending on the configuration all operations are performed with 16 or 32-bit operands. Either two 16 bit or two 32 bit barrel shifters are used, depending on the configuration. The width and delay of the ALU is affected in great extent.

5.5.7 Frame Pointer

The frame pointer concept used by SPEAR had limited capabilities. Hence a complete redesign was required. In the following an overview what changed with the new implementation is given:

- Originally the frame pointer registers were part of the extended register file. In SPEAR2 they are located in the system control module.
- The number of independent frame pointer was increased by one. Now four registers are available for frame management.
- The size of frames was enlarged from 32 to 64 words.
- Frame pointer based memory access and concurrent manipulation of the frame pointer register is now possible within a single instruction. This allows to emulate push and pop to support efficient stack management.

There was several reasons for moving the frame pointer registers away from the extended register file. First of all the reduces number of available registers in the SPEAR2 architecture. Another problem constitutes the the additional write port which would be required when incrementing or decrementing the frame pointer automatically for stack emulation. This would yield a register file with 4 ports which have to be accessible contemporaneously.
After the completion of SPEAR2 it turned out the decision to move the frame pointer registers into the system control module was right, because the critical path was shorted and the performance of SPEAR2 increased.
The rearranged instruction set architecture provided emerged encoding space to add a fourth frame pointer and to enlarge the addressable size of frames.

Although having an efficient access to the frame pointer register, the absence of auto increment and decrement was a substantial disadvantage of SPEAR. After it got clear that the frame pointer registers will be moved to the system control module it was impossible to abandon this feature, because changing the content of frame pointer register will get more costly in terms of required instructions. The supported operations are post-increment and post-decrement. However, pop requires pre-increment instead of post-increment. The problem is solved using post-increment and an offset of -1. Push is emulated with post-decrement and an offset of zero.
Frame pointers can only be used to access the data memory by word. That means 16 bit access in the small configuration and otherwise 32 bit. Thus the constant value for auto-increment and auto-decrement is affected by the configuration: When using a 16 bit data path the value has to be two otherwise the value has to be four.

Impact of the Customizable Data Path

The width of the frame pointer is controlled by the configuration of the customizable data path. Furthermore, the value for increment and decrement depends on the configuration. Since the frame pointer registers are located at the system control module the interface of this module is also affected. More information about the impact to the system control module is given in Section 5.6.1.

5.5.8 Data Memory

In contradiction to SPEAR, which used asynchronous memory, synchronous memory is used to implement the data memory of SPEAR2. Since most FPGA platforms have only synchronous memory for implementing large memory structures. The synchronous memory implied an additional pipeline stage.
Next the changed memory organisation is explained and afterwards how this memory is implemented.

Memory Organisation

As mentioned in Paragraph 3.1.2, SPEAR was only able to access and address 16-bit data types. This restriction simplified the memory access and the memory organisation. Furthermore twice as much memory could be addressed. But this memory implementation had several drawbacks. First of all, it was hard to port the GNU C Compiler to SPEAR, since the compiler was designed for byte addressed data memories. Furthermore, memory was wasted if 8 bit values had to be stored.

The new memory organisation is depicted in Figure 13. SPEAR2 uses byte addressing for its data memory. As shown in the Figure concerning data

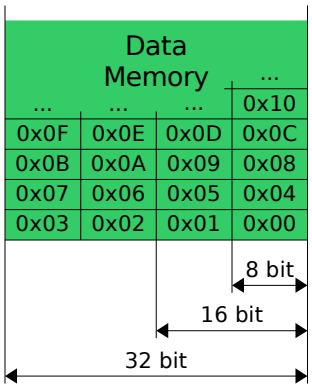


Figure 13: Organisation of Data Memory

memory organisation, every byte has its own address and so it is possible to address every byte directly. The width of the data path limits the type of memory access. Only with the 32 bit data path it is possible to read or write four bytes at once. 8 and 16 bit access is supported by both data path configurations. Every memory access has to be aligned. This means that for a 32 bit access the address has to be evenly divisible by four and for 16 bit it has to be evenly divisible by two. The SPEAR2 architecture is little endian. Words and halfwords are stored in memory with the more significant bytes at higher addresses.

Memory Architecture

Since the data memory has to enable write access to single bytes, four parallel memories were used to implement the data memory. The internal architecture is depicted in Figure 14. To connect the data memory several signal

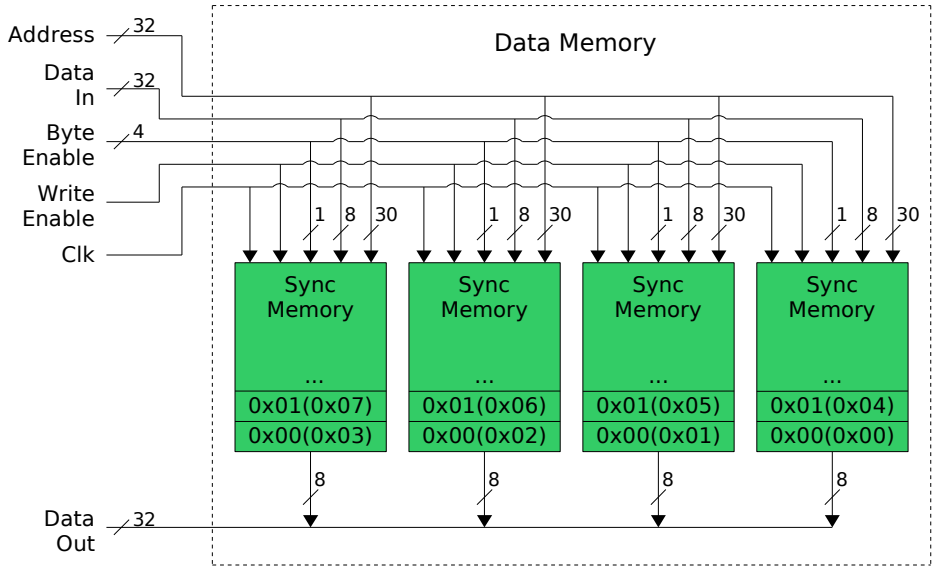


Figure 14: Architecture of Data Memory

vectors are used. The data in and out ports for example consist of 32 lines. Inside the data memory the vectors are split into 8 bit vectors and are connected to the four memories. Also the byte enable vector is split. The same address is applied to all memories but without the two least significant bits. Instead the information carried by these bits is used to derive the byte enable vector. This vector is generated by the memory access unit which is described in the next section.

Since most memories, provided by FPGAs, have only one write-enable signal, four separate memories were used to build the data memory.

Impact of the Customizable Data Path

At an early stage of development two different implementations were used for

16-bit and 32-bit data path. For the 16-bit version, only two 8-bit memories were used in parallel, acting as 16-bit memory. And for the 32-bit version an implementation similar to the current solution was used. But it turned out that a uniform interface to the data memory and also to the extension modules would be the better solution. Hence a separate memory access unit was implemented which generates the byte enable signals and aligns the data for memory access. In this way, the same memory architecture can be used for both data path configurations.

5.5.9 Memory Access Unit

To enable a configuration independent memory interface a separate memory access unit was designed. Thereby the width of data path can be changed without notice by data memory or extension modules.
The main task of this unit is the alignment of bytes and half words. The required shift is determined by the two least significant bits. Depending on these two bits, there are four possibilities to align a byte:

- Both bits are zero. No alignment is necessary and the byte enable bit for the lowest byte will be set.
- If only the least significant bit is set, the byte has to be shifted left by 8 bits. Now the second byte enable bit will be set.
- If only the higher bit is set, the byte has to be shifted left by 16 bits. Only the third byte enable bit will be set.
- If both bits are set, the byte has to be shifted left by 3 bytes. Only the highest byte enable bit will be set.

The alignment of 16 bit values is quite simple. Since only memory aligned access is allowed the least significant bit has to be zero. If the other bit is set, the half word has to be shifted left by two bytes. The two corresponding byte enable bits have to be set. For 32 bit values no alignment is required, all byte enable bits have to be set.

For correct read access the alignment of 8 and 16 bit values has to be done in reverse.
Beside the alignment for read and write the memory access unit is responsible for sign extension.
Without the memory access unit the data memory and every extension module would be responsible for alignment and sign extension. This would result in higher resource usage and more complex extension modules.

Impact of the Customizable Data Path
The impact is small: 32-bit access is not supported by the 16 bit data path.

5.5.10 Exceptions Vector Table

As before exceptions can be classified into two groups:

1. Interrupts, which are triggered by hardware.
2. Traps, which are triggered by software.

For both types 16 different sources are possible. Therefore 32 exceptions are supported. The addresses of the service routines are stored in the exception vector table. The instruction set provides special instructions to build and manipulate the exception vector table. The table contains 32 entries. The trap vectors are stored at the bottom half and interrupt vectors at the upper half of the exception vector table, which is depicted in Figure 15.

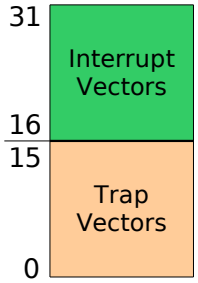


Figure 15: Exception Vector Table of SPEAR2

The trap instruction can be used to trigger the exceptions from 0 to 15. The exception vector at position 16 is used for interrupt line 0, which has the lowest priority. The higher the interrupt number the higher the priority and the position in the exception vector table.
The interrupt controller is located at the system control module and comprises an interrupt protocol and an interrupt mask register. Every interrupt will be noted and saved to the protocol register. But the interrupt service routine will only be executed if the corresponding bit in the mask register is cleared. If an interrupt occurs and the mask bit is set, the interrupt can be cleared by software or will be trigger as soon as the mask bit is cleared. What changed with SPEAR2 is how an interrupt can be cleared. Before it was necessary to load the interrupt protocol register, change the intended bits and write back the result. Thereby it was possible to miss interrupts if the interrupt had occurred between the read out and write back of the protocol register. Now the read out is not needed anymore. If now a value is written to the protocol register, the final value is build by an exclusive OR operation between the write value and the content of the protocol register. In othe! r! words, all bits which are set will be inverted. Since the bits are inverted, it is also possible to trigger hardware interrupts by software.
The new way to change the protocol register is faster and more importantly, no interrupts can be missed.

Impact of the Customizable Data Path
Only the exception vector table is affected by the configuration of the customizable data path.

5.5.11 Optimization

This section gives some information about the development and describes some design issues of SPEAR2.
The order in which the components of the processor were implemented is similar to the outline of this chapter. The program counter was implemented first. Afterwards the processor was completed incrementally component after component. For the first development step, only the functionality of the

processor was taken into account. The first implementation already included the customizable data path. After a first version with basic functionality was implemented, for further development two requirements were taken into account:

- Economical resource usage
- Improve achievable clock frequency

Often the resource usage and achievable clock frequency depend on each other. The lower the resource usage the faster the achievable clock frequency. Large multiplexer structures for example have long delays. Trying to reduce the size of multiplexer structures has positive impact to both aspects and was kept in mind during development.
In general two basic approaches were used to enable a cost effective and fast design:

- Use multiplexer only where needed.
- Calculate the control signals for multiplexer one cycle before the control signals are used.

The first method seems quite simple. The whole trick is to switch only the required signals. As mentioned in Section 5.3.2, only the crucial signals are changed when an instruction has to be flushed due an ongoing jump or exception. For example, to set the register destination address to a defined value would have no impact but requires more resources. This technique was mostly used to improve the efficiency inside the second pipeline stage.
The second method, already mentioned in Section 5.5.5, increases the resource usage minimal but can improve achievable clock frequency notable. In particular the delay of the forwarding unit was improved by this technique. The method can only be applied if it is possible to split the multiplexer structure and calculate the control logic in advance. Thereby the multiplexer can be switched without the time delay for calculating control signals.

5.6 Extension Modules

Since the requirements of every project are slightly different, extension modules are used to customize the processor. A generic interface between processor core and extension modules has been designed. The generic interface interface is build by a physical and a logical interface.
The physical interfaces of the extension modules are mapped to a unique address in the data memory address space. Hence only conventional load and store instructions are required to access an extension module. The interface provides 32 registers a 8 bit. The byte oriented description enables to use the same interface, however the data path is configured. It is possible to access single bytes or any other supported data type.
The logical interface describes the usage of the interface. The first two bytes are used to indicate the module status and are read only. The next two bytes are used for configuration. The first status and the first configuration byte are predefined. The other bytes can be used for module specific issues.

Generic Part of Interface

The generic part of the extension module interface comprises two registers. The generic status register and the generic configuration register. Both registers have a width of one byte. First the generic status byte will be explained, which is depicted in Figure 16.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
LOOR	-	-	FSS	BUSY	ERR	RDY	INT

Figure 16: Generic Status Byte

The generic status register holds the following status bits:

- INT (Interrupt): This bit is set automatically if the module triggers an interrupt. This feature can be used to determine which module triggered an interrupt, if several extension modules share the same interrupt line.

- RDY (Ready): This bit is used to indicate the ready-to-operate state of the extension module.
- ERR (Error): This bit is automatically set if an error occurs.
- BUSY: This bit indicates if the extension module is not ready for new tasks.
- FSS (Fail Safe State): FSS is used to indicate if the extension module is in a fail safe state. The extension module can switch to this state automatically or manually by software.
- LOOR (Loop Ready): In interaction with LOOW, this bit can be used to determine the presence of an extension module. The bit written to LOOW appears at LOOR after one cycle.

The generic configuration byte is depicted in Figure 17.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
LOOW	-	-	EFSS	OUTD	SRES	ID	INTA

Figure 17: Generic Config Byte

The generic configuration register holds the following configuration bits:

- INTA (Interrupt Acknowledge): Can be used to acknowledge the interrupt of an extension module and clears the INT flag in the generic status register.
- ID (Identify): If this bit is set, byte 4 to 7 can be used to read the manufacturer and version number of the extension module.
- SRES (Soft Reset): This bit can be used to reset an extension module. The extension module should make no difference between a soft and a hardware reset.

- OUTD (Output Disable): If this bit is set, the extension module shall not drive the interfaces to its environment. Thereby it should be possible to disconnect an extension module from its environment.
- EFSS (Enter Fail Safe State): It can be useful for some extension modules to define a fail safe state. For example, if the extension module is responsible for controlling a motor a fail safe state which stop the motor could be useful. By setting this bit, the extension module enters the fail safe state.
- LOOW (Loop Write): In interaction with LOOR it can be used to determine the presence of an extension module. The bit written to LOOW appears at LOOR after one cycle.

Classification of Extension Modules

As mentioned in Section 3.5, extension modules are classified by their functionality:

- System-Extension Module, which have an extended interface to the processor and improve the functionality of the processor core.
- Function-Extension Module, which can be used to implement functions in hardware which otherwise have to be emulated by software.
- IO-Extension Module, which are used to implement interfaces to the processor like RS232 or PS/2.

The next paragraphs will describe some important extension modules.

5.6.1 System Control Module

Although the system control module acts as an extension module, it can not be omitted. Since it contains the processor status register, the interrupt handler, and four registers used as frame pointer register, this extension module is an integral part of the processor. Moving all these registers into

an extension module, which in turn is memory mapped, allows simple manipulation by using conventional load and store instructions. The interface of the system control module is depicted in Figure 18.

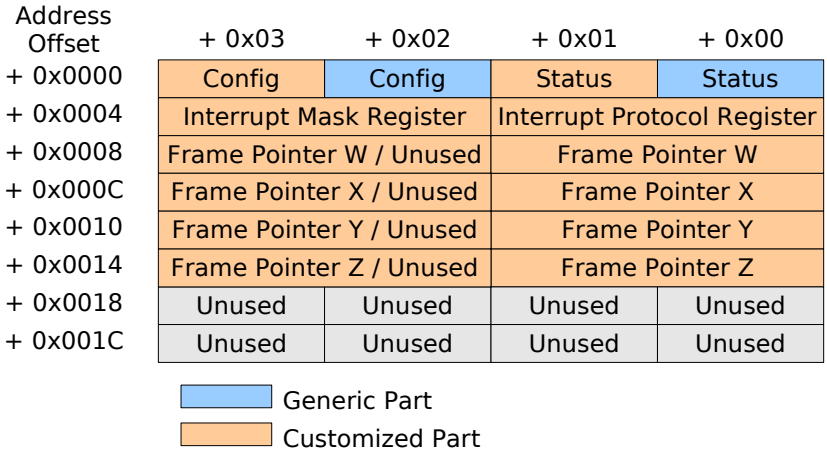


Figure 18: Interface of the System Control Module

Processor Status

The customized status byte holds the processor status flags which are manipulated by the processor. In the case of an exception, the processor status byte is copied to the processor configuration register. The processor status register is automatically restored by returning from exception. The customized status and configuration byte are nearly identical, but only the configuration byte can be overwritten manually. Thus the only way to manipulate the processor status register is to manipulate the processors configuration register and using the return from exception instruction to write to the processor status register.

Beside processor status flags, the processor status register holds some additional flags. One bit of the processor status register and one of the processor configuration register are not used.

The processor status register is depicted in Figure 19.

The processor status register holds the following flags:

Status Byte

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
GIE	0	-	COND	ZERO	NEG	CARRY	OVER

Configuration Byte

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
GIE	SLEEP	-	COND	ZERO	NEG	CARRY	OVER

Figure 19: Customized Status and Configuration Byte of the System Control Module

- OVER: The first bit is used as overflow flag. It indicates whether the result of an operation has overflowed according to the two's complement representation.
- CARRY: Bit 1 is used as carry flag. It is used to indicate when a arithmetic carry has been generated out of the most significant ALU bit position.
- NEG: The negative flag indicates if the result of the ALU operation is negative.
- ZERO: The zero flag indicates whether the result of a mathematical or logical operation was zero.
- COND: The condition flag is used to determine if a conditional instruction has to be executed or not. The conditional flag can only manipulated by special instructions.
- SLEEP: The sleep flag is used to activate the sleep mode of the processor, which is described at the end of this section. This bit can only be set in the processor configuration register and is alway zero in the processor status register.
- GIE: The global interrupt enable flag is used for interrupt controlling. Interrupts can be disabled when clearing this flag.

Interrupt Configuration Registers

Two registers of the system control module are used by the interrupt handler:

- Interrupt Protocol Register: This register is used to protocol incoming interrupts.
- Interrupt Mask Register: This register is used to mask the interrupt sources individually.

The Interrupt handler is described in more detail in Section 5.5.10.

Frame Pointer Registers

Four registers of the system control interface are used as frame pointer. Thus the amount of addressable data memory depends on the data path configuration, the width of the frame pointer registers can be 16 or 32 bit. The frame pointer concept is described in Section 5.5.7.

Sleep Mode

A new feature introduced with SPEAR2 is the sleep mode. As known by other processors it can be used to reduce power consumption, when the processor is idle. Since the processor will only wake up from sleep mode when an interrupt service routine has to be executed, the sleep mode can only be used by interrupt driven programs.
To put the processor to sleep mode, the *sleep* bit has to be set. After the bit is set, all flip-flops of the processor - except the flip-flops of the interrupt handler - are stopped.

5.6.2 Programmer Module

The programmer module provides a write access to the instruction memory. In conjunction with an extension module with a communication interface, this module can be used for downloading new program code into the instruction memory. Dividing this task into a programming and a communication part enables easy changing of the download interface since only the communication module has to be replaced. The interface of the programmer module is depicted in Figure 18.

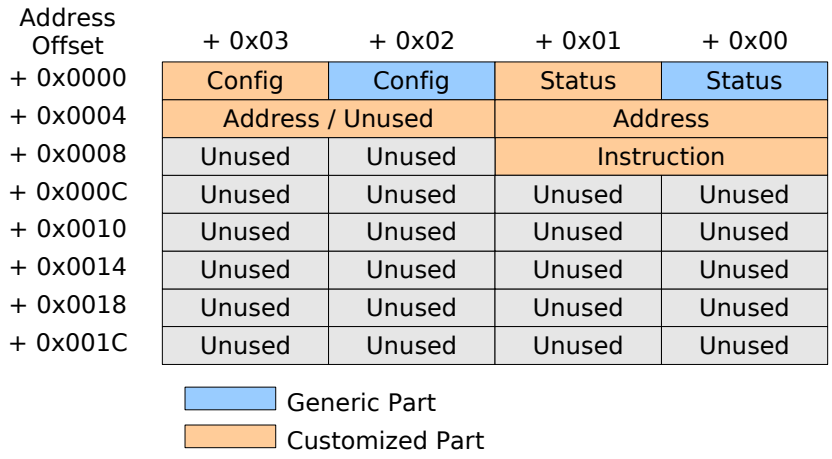


Figure 20: Interface of the Programmer Module

Only two data registers are used:

- Address Register: This register holds the destination address for the instruction. The width of this register can be 16 or 32 bit, depending on the configuration of the data path.
- Instruction Register: This register holds the instruction which has to be saved to the instruction memory.

The customized status register is not used. Only the customized configuration register, which is depicted in Figure 21.

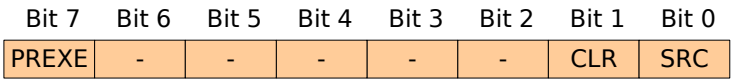


Figure 21: Customized Config Byte of the System Control Module

The customized configuration register holds the following configuration bits:

- SRC: This bit is used for source selection. After reset, this bit is zero and the boot memory is selected as instruction source. To use the instruction memory as instruction source, this bit has to be set.

- CLR: This bit can be used to trigger a soft-reset of the processor. In distinction from a normal reset, the registers of the programmer module are not affected by an soft-reset.
- PREXE: This bit is used to trigger the write to instruction memory. After the correct values are loaded to the address and instruction register, this bit has to be set. In order to provide a more efficient program download, the address register is automatically incremented by one, when the store procedure is finished.

If the program download is completed, the *src* and *crl* bit have to be set at once. Afterwards the program execution starts at address zero, and the instructions are taken from the instruction memory.

5.7 Differences: 16 vs. 32 bit Version

This paragraph will summarize the implementation differences between the 16-bit and enlarged 32-bit data path. The effects on performance and resource usage will be discussed in Chapter 7.

5.7.1 Interface

The interface to extension modules and memory is not affected by the width of the data path. This is possible, since always the 32-bit interface is used. Thereby a consistent interface is provided and the overhead for the 16-bit configuration is negligible. The interface is described in detail in Paragraph 6.2.

5.7.2 Instruction Set Architecture

The width of data path has small impact to the instruction set. Practically the same instruction set is used for both configurations.
The 32-bit version provides 3 additional instructions compared to the 16-bit version. Following instructions are only available if the extended data path is used:

- Load word - Loads 32 bit from memory
- Store word - Stores 32 bit to memory
- Load half word unsigned - Loads 16 bit from memory without sign extension

A processor with 16 bit data path triggers an exception, if such an instruction has to be executed.

Additional some instructions are different regarding width of the second operand. The affected instructions are listed below:

- Shift left - Shifts the content of a register
- Shift right - Shifts the content of a register
- Shift right arithmetic - Shifts the content of a register
- Bit clear - Clears a specific bit
- Bit set - Sets a specific bit
- Bit test - Tests if a specific bit is set

The second operand is used to specify the number of bits to shift or to specify a certain bit in a register. Therefore only 4 bits are necessary if the registers width is 16 bit and 5 bits are required for 32-bit registers.

Detailed information about individual instructions is given in Appendix B.

5.7.3 Addressable Memory

The width of data path affects also the amount of addressable memory, since the width of all address registers is the same as the width of the data path. Hence, choice of data path configuration is not only a question of performance and resource usage, but also how much memory is required. If more than 128KB program memory and 64KB data memory is required, the extended data path has to be used. With extended data path, SPEAR2 can address 8GB program and 4GB data memory.

6 Configuration and Interface Description

This chapter describes the configuration options and interfaces of SPEAR2.

6.1 Configuration

This section provides information about the configuration options of SPEAR2. The configuration is stored in a vhdl file. Five different options are supported:

- TECH_C - Specifies the target technology. If no certain technology is selected, only device independent generic components are used. Using of device specific components enables higher optimization.
- WORD_CFG_C - Specifies the width of data path. If 1, the smaller data path configuration is selected. If 2, the data path will be extended to 32 bit.
- INSTR_RAM_CFG_C - Specifies the width of instruction memory. More precisely, specifies the number of address lines used for instruction memory. Hence if *INSTR_RAM_CFG_C* is set to 5, 32 byte of instruction memory are addressable, if 6, 64 byte and so on. Without extended data path, the maximum number of address lines is 16.
- DATA_RAM_CFG_C - Specifies the size of data memory. More precisely, specifies the number of address lines used for data memory. Hence if 5, 32 byte of data memory are addressable. If 6, 64 byte and so on. Without extended data path, the maximum number of address lines is 16.
- USE_IRAM_CFG_C - The basic configuration of SPEAR2 includes only the bootrom as instruction memory. By enabling *USE_IRAM_CFG_C* the instruction RAM is included as additional instruction memory. This option also determines whether the programmer module is included. If no instruction RAM is available, the programmer module is not needed.

Assignment	*Type*	*Allowed Values*
TECH_C	String	NO_TARGET XILINX ALTERA
WORD_CFG_C	Integer	1 → 16-bit data path 2 → 32-bit data path
INSTR_RAM_CFG_C	Integer	5 to 16 (independent of data path width) 17 to 32 (only with extended data path)
DATA_RAM_CFG_C	Integer	5 to 16 (independent of data path width) 17 to 32 (only with extended data path)
USE_IRAM_CFG_C	Boolean	TRUE → Instruction RAM available FALSE → No Instruction RAM available

Table 3: Configuration Options of SPEAR2

Table 3 shows the available options and their allowed values.
The options of the AMBA module [8] will also be configured through this configuration file after the AMBA module has been integrated into SPEAR2.

6.2 Interface

The interface is divided into general signals, an input record, and an output record. Records contain several signals. The interface of SPEAR2 is depicted in Figure 22.

Global Signals

Three signals are common signals which are listed in Table 4.

Signal	*I/O*	*Description*
clk	I	Clock
extrst	I	External reset
sysrst	O	Internal reset

Table 4: General Interface of SPEAR2

clk

All operations of SPEAR2 are synchronous to this clock.

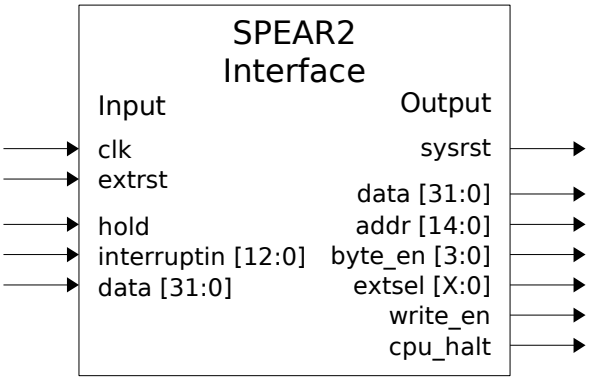


Figure 22: Interface of SPEAR2

extrst

The external reset puts SPEAR2 into initial state. The reset is synchronous and low-active.

sysrst

The internal reset indicates a reset of SPEAR2, which can be triggered by an external reset or by soft reset triggered by software. The signal is low-active.

Input Signals

The items of the input record are listed in Table 5.

Input Signal	*Description*
`hold`	Extend extension module access
`interruptin [12:0]`	Interrupt source lines
`data [31:0]`	Input from extension modules

Table 5: Input Interface of SPEAR2

hold

Is used to extend the bus cycle. If the signal is sampled high, all outputs of SPEAR stay unchanged. Hold is sampled on the rising edge of the clock.

interruptin

SPEAR2 distinguish 16 different interrupts. Thirteen interrupt lines are available, the other interrupts are used for internal interrupt sources. Interrupts are high-active.

data

The read data bus carries data read from extension modules. Due to the fact, that all extension module data outputs are set to zero when not selected, the data bus of different extension modules is merged by a simple OR-operation. Thus, SPEAR2 needs only one data input bus and no additional control logic is required.

Output Signals

An overview of the output record is given in Table 6.

Output Signal	*Description*
`data [31:0]`	Output to extension modules
`addr [14:0]`	Address for extension modules
`byte_en [3:0]`	Byte enable signals
`extsel [X:0]`	Extension module select signal
`write_en`	Write enable signal
`cpu_halt`	CPU state

Table 6: Output Interface of SPEAR2

data

The write data bus is an output of the core and contains the data that is written to extension modules.

addr

The address bus indicates the extension module address that is being accessed by the current transfer.

byte_en

The byte enable signals indicate which byte lanes of the data bus contain valid data.

extsel

The select signal indicates if an access to extension modules is in progress. This signal is generated for every extension module and can only be active for one extension module at the same time. The number of different *extsel* signals is equal to the number of attached extension modules.

write_en

The write enable signal indicates if a write or read transfer is in progress. If the extension module is selected and write enable is low: a read is in progress. If the module is selected and write enable is high: a write is in progress.

cpu_halt

The cpu-halt signal indicates if the core is in sleep state.

All extension modules get the same input signals, except the *extsel* signal is different.

7 Results

This chapter summarizes the results of this master thesis, and compares resource usage and performance of both data path configurations.

7.1 Processor Characteristics

Among the most interesting features of a soft core processor are resource usage and maximum clock frequency. Both are analysed in this section.

The toolchain of Xilinx was used for analysis, namely the Xilinx Synthesis Technology (XST) [14] in version 9.1. As FPGA a Spartan-3 was used, which is the current low-cost FPGA family of Xilinx. Before the results of synthesis will be presented, a short introduction on the internals of Spartan-3 devices is given.

Spartan-3 Devices
The resource usage of Spartan-3 devices is best measured in slices and block RAMs.
A slice is the basic element of a Xilinx device. One slice has the following elements:

- 2 Look-Up Tables (LUT) - main resource for implementing logic functions. Every LUT has 4 inputs.
- 2 Flip-Flops
- Some multiplexers to effectively combine LUTs in order to permit more complex logic operations.
- Some carry logic and various dedicated arithmetic logic gates, to support fast and efficient implementations of math operations.

All Spartan-3 devices support block RAM, which are synchronous 18Kbit memory blocks. The aspect ratio – i.e. width vs. depth – of each block

RAM is configurable. Furthermore, multiple blocks can be cascaded to create still wider and/or deeper memories.

The Spartan-3 device, which we used for analysis was the xc3s2000-fg656-5. This device has 20.480 slices and 40 block RAMs.

7.1.1 Resource Usage

We tested SPEAR2 with different options. XST provides the possibility to optimize for speed or resource usage. The results are shown in Table 7.

	Optimization	
	Speed	*Area*
16-bit data path	1132 Slices 9 RAMB16s 60 MHz	837 Slices 11 RAMB16s 54 MHz
32-bit data path	1896 Slices 9 RAMB16s 58 MHz	1455 Slices 11 RAMB16s 43 MHz

Table 7: Synthesis Results of SPEAR2

Resource usage and maximum clock frequency depend strongly on the chosen optimization.
If optimized for speed, only 9 block RAMs are used. Otherwise, two more are needed to implement the design. The reason is how the register file is implemented. There are two possible solutions:

1. Using distributed memory
2. Using block RAM

If the register file is implemented with block RAMs, two of them are needed, because the register file requires three ports and block RAMs have only two ports. Therefore two mirrored dual-port memories have to be used to implement one three-port memory.

16-bit vs. 32-bit data path

An interesting detail of Table 7 is, how much resource usage increases for the extended data path. Because this value gives information about additional costs for the extended data path.
Independent of optimization, the resource usage increases by about 70 percent. There are several reasons for the increased resource usage:

- Most parts of the processor are affected by the data path configuration, and all affected data and address registers have doubled resource consumption.
- The barrel shifter for the 32-bit data path requires 160 multiplexer, and for the 16-bit data path only 64 multiplexer are required.

Basically the decoder and control logic have nearly constant resource consumption.

7.2 Performance

In this section the performance benefits of the extended data path are analysed. Therewith and with the information about increased resource usage, a cost-benefit analysis can be done.
To analyse the performance, a face recognition program were used. The program and test conditions are listed in Appendix A.

Table 8 compares execution time of different functions and provides information about performance improvements, possible with an extended data path. The execution time is measured in clock cycles.
The results provide interesting information: The execution time of functions depend strongly on the used data types. The functions *fix_fft()* and *squareRoot()* use mostly 32-bit data types and the functions *TransposeImage()* and *findMax()* use mostly 16-bit data types. If most used variables are of 32-bit data types, the use of extended data path is recommendable. On the other side, if most used variables are of 16-bit data types, the execution takes longer if the data path is configured for 32 bit.

	SPEAR2 16-bit data path	*SPEAR2* 32-bit data-path
fix_fft()	97,329 cycles	27,085 cycles
complexPointMatrixMult()	7,472 cycles	5,736 cycles
squareRoot()	5,218 cycles	3,119 cycles
TransposeImage()	4,220 cycles	7,064 cycles
findMax()	1,961 cycles	2,793 cycles

Table 8: Execution Time of Different Functions

16-bit vs. 32-bit data path

The used data types have a massive impact to the execution time. If possible, the coder of a program should use the width of data path as size for the used variables, because processing 32-bit variables with 16-bit registers cause a big overhead and processing 16-bit variables with 32-bit registers requires to mask the upper 16 bit before the variable is used for arithmetic or compare operations.

8 Conclusion

In the course of this master thesis a processor with a configurable data path width - 16 bit or 32 bit - was proposed.
The advantages of such a processor are that it has one instruction set, one toolchain, and the same extension module interface. Regardless, the processors can be configured for different performance.

The processor was implemented successfully, which was the main objective of this master thesis. The data path can be extended to 32 bit in return for additional resource usage of about 70 percent. Developed extension modules work with both data path configurations without adaptation.
Tests with SPEAR2 showed that one condition has to be fulfilled to gain optimal performance with different data path widths:

- If possible, the data type of variables should be 16 bit if the 16-bit data path is used and 32 bit for the 32-bit data path. Otherwise some overhead is produced by the compiler to handle data type sizes different to the register widths.

Additional to the concept of configurable data path, other useful features were successfully implemented. SPEAR2 is prepared for future FPGA families and provides improved functionality compared to SPEAR.

9 Outlook

Since SPEAR2 is already used and tested in some projects, the feedback gotten gives information about improvable features.
One idea is to remove the exception vector table and to use predefined exception vectors. This would reduce resource usage and result in a simpler architecture. On the other hand, the latency of interrupt handling would be increased by a few clock cycles, applying this optimization.
It would also be possible to modify the forwarding unit of the pipeline. Using stalls instead of forwarding in some cases, would allow higher frequencies by reducing resource usage. On the other hand, some instructions would be stalled and the execution time prediction would be more difficult.

An idea with much more potential for improvements would be a fundamental change of the ISA.

Another Approach - 32-bit ISA for SPEAR2
The main disadvantage of SPEAR2 compared to SPEAR is the reduced number of general purpose registers. It is hard to handle thirty-two 32-bit registers and supporting conditional instructions with a 16-bit ISA. So we decided to reduce the number of general purpose registers.
Compared to other architectures, the main drawback of the SPEAR2 architecture is the limited 16-bit ISA. Even without conditional instructions, 16 bit for instruction coding is limited.

An idea which would noticeably improve performance would be the use of 32-bit instructions. The advantages of 32-bit ISA would be as follows:

- The number of general purpose registers could be scaled up to 32 registers.
- One instruction could determine two source and one separate destination register.

- Reduction of the number of instructions would be possible; if one register was defined to always be zero, some instructions could be removed and emulated by other instructions. The load-immediate instruction, for example, could be emulated by an OR-immediate instruction, resulting in a much simpler architecture.
- The immediate values could be 16-bit, enough for most operations.
- The decoder could be designed more primitively - only two or three different instruction formats would be required, resulting in lower resource usage and faster decoding.

The only drawback would be that the required amount of program memory would increase. But not by a factor of two, rather less. Since more registers and larger immediate values would be available, fewer instructions would be required to solve the same problem. Another reason to weaken this drawback are the characteristics of program memory. Program memory is fast and cheap.

A soft core processor like SPEAR2 with customizable data path and 32-bit ISA would be an interesting and probably highly efficient architecture. However, this optimization step would also involve a complete revision of the toolchain, especially of the C-compiler, and is therefore out of the scope of this master thesis.

A Code Listing

This appendix lists all code examples.

A.1 Face Recognition Program

As example, the face recognition program was used. This program was developed by Kristian Ambrosch and Peter Tummeltshammer, and uses a fixed-point fast fourier transform developed by Tom Roberts, Malcolm Slaney, and Dimitrios P. Bouras.
The code of the face recognition program was separated for a better analysis. This way more detailed information is provided about the performance differences between both data path configurations.
For compilation the *spear16-gcc* and *spear32-gcc* were used with optimization level 2.

A.1.1 C Code

First the C code for the fixed-point fast fourier transform function is presented. This function was used by the face recognition program. To analyse the execution time, parameter m was set to 5 and parameter *inverse* was set to zero. The other function parameters have no impact to the execution time.

```
1  /* fix_fft.c - Fixed-point in-place Fast Fourier Transform  */
2  /*
3     All data are fixed-point short integers, in which -32768
4     to +32768 represent -1.0 to +1.0 respectively. Integer
5     arithmetic is used for speed, instead of the more natural
6     floating-point.
7
8     For the forward FFT (time -> freq), fixed scaling is
9     performed to prevent arithmetic overflow, and to map a 0dB
10    sine/cosine wave (i.e. amplitude = 32767) to two -6dB freq
11    coefficients. The return value is always 0.
12
13    For the inverse FFT (freq -> time), fixed scaling cannot be
14    done, as two 0dB coefficients would sum to a peak amplitude
15    of 64K, overflowing the 32k range of the fixed-point integers.
16    Thus, the fix_fft() routine performs variable scaling, and
17    returns a value which is the number of bits LEFT by which
18    the output must be shifted to get the actual amplitude
19    (i.e. if fix_fft() returns 3, each value of fr[] and fi[]
20    must be multiplied by 8 (2**3) for proper scaling.
21    Clearly, this cannot be done within fixed-point short
22    integers. In practice, if the result is to be used as a
23    filter, the scale_shift can usually be ignored, as the
24    result will be approximately correctly normalized as is.
25
26    Written by:  Tom Roberts  11/8/89
27    Made portable:  Malcolm Slaney 12/15/94 malcolm@interval.com
28    Enhanced:  Dimitrios P. Bouras  14 Jun 2006 dbouras@ieee.org
29    Modified: Kristian Ambrosch and Peter Tummeltshammer 8/5/07
30    reduced N_WAVE from 1024 to 32
31 */
32
33 #include <stdint.h>
34
35 #define N_WAVE      32    /* full length of Sinewave[] */
36 #define LOG2_N_WAVE 5     /* log2(N_WAVE) */
37
```

```
38
39  /*
40    Henceforth "Uint16_t" implies 16-bit word. If this is not
41    the case in your architecture, please replace "Uint16_t"
42    with a type definition which *is* a 16-bit word.
43  */
44
45  /*
46    Since we only use 3/4 of N_WAVE, we define only
47    this many samples, in order to conserve data space.
48  */
49
50  int16_t Sinewave[N_WAVE-N_WAVE/4] = {
51    0, 6392,12539,18204,23169,27244,30272,32137,32767,32137,30272,27244,23169,
52    18204,12539,6392, 0, -6392, -12539,-18204,-23169,-27244,-30272,-32137
53  };
54
55  /*
56    FIX_MPY() - fixed-Point multiplication & scaling.
57    Substitute inline assembly for hardware-specific
58    optimization suited to a particluar DSP processor.
59    Scaling ensures that result remains 16-bit.
60  */
61  //inline
62  int16_t FIX_MPY(int16_t a, int16_t b)
63  {
64    /* shift right one less bit (i.e. 15-1) */
65    int32_t c = ((int32_t)a * (int32_t)b) >> 14;
66    /* last bit shifted out = rounding-bit */
67    b = c & 0x01;
68    /* last shift + rounding bit */
69    a = (c >> 1) + b;
70    return a;
71  }
72
73  /*
74    fix_fft() - perform forward/inverse fast Fourier transform.
75    fr[n],fi[n] are real and imaginary arrays, both INPUT AND
76    RESULT (in-place FFT), with 0 <= n < 2**m; set inverse to
77    0 for forward transform (FFT), or 1 for iFFT.
78  */
79  int32_t fix_fft(int16_t fr[], int16_t fi[], int16_t m, int16_t inverse)
80  {
81    int32_t mr, nn, i, j, l, k, istep, n, scale, shift;
82    int16_t qr, qi, tr, ti, wr, wi;
83
84    n = 1 << m;
85
86    /* max FFT size = N_WAVE */
87    if (n > N_WAVE)
88      return -1;
89
90    mr = 0;
91    nn = n - 1;
92    scale = 0;
93
94    /* decimation in time - re-order data */
95    for (m=1; m<=nn; ++m) {
96      l = n;
97      do {
98        l >>= 1;
99      } while (mr+l > nn);
100     mr = (mr & (l-1)) + l;
101
102     if (mr <= m)
103       continue;
104     tr = fr[m];
105     fr[m] = fr[mr];
106     fr[mr] = tr;
107     ti = fi[m];
108     fi[m] = fi[mr];
109     fi[mr] = ti;
110   }
111
112   l = 1;
113   k = LOG2_N_WAVE-1;
114   while (l < n) {
115     if (inverse) {
116       /* variable scaling, depending upon data */
117       shift = 0;
118       for (i=0; i<n; ++i) {
119         j = fr[i];
120         if (j < 0)
```

```
121             j = -j;
122           m = fi[i];
123           if (m < 0)
124             m = -m;
125           if (j > 16383 || m > 16383) {
126             shift = 1;
127             break;
128           }
129         }
130         if (shift)
131           ++scale;
132       } else {
133         /*
134           fixed scaling, for proper normalization --
135           there will be log2(n) passes, so this results
136           in an overall factor of 1/n, distributed to
137           maximize arithmetic accuracy.
138         */
139         shift = 1;
140       }
141       /*
142         it may not be obvious, but the shift will be
143         performed on each data Point exactly once,
144         during this pass.
145       */
146       istep = l << 1;
147       for (m=0; m<l; ++m) {
148         j = m << k;
149         /* 0 <= j < N_WAVE/2 */
150         wr =  Sinewave[j+N_WAVE/4];
151         wi = -Sinewave[j];
152         if (inverse)
153           wi = -wi;
154         if (shift) {
155           wr >>= 1;
156           wi >>= 1;
157         }
158         for (i=m; i<n; i+=istep) {
159           j = i + l;
160           tr = FIX_MPY(wr,fr[j]) - FIX_MPY(wi,fi[j]);
161           ti = FIX_MPY(wr,fi[j]) + FIX_MPY(wi,fr[j]);
162           qr = fr[i];
163           qi = fi[i];
164           if (shift) {
165             qr >>= 1;
166             qi >>= 1;
167           }
168           fr[j] = qr - tr;
169           fi[j] = qi - ti;
170           fr[i] = qr + tr;
171           fi[i] = qi + ti;
172         }
173       }
174       --k;
175       l = istep;
176     }
177     return scale;
178   }
```

Now the C code of the face recognition program is presented. The function *squareRoot()* was used to calculate the square root of 50,000. For the function *TransposeImage()* and *findMax()* the values of *ImageWidth* and *ImageHeight* were set to 16 and for the function *complexPointMatrixMult()* the values of *ImageWidth* and *ImageHeight* were set to 4. The other function parameters have no impact to the execution time.

```
1  ////////////////////////////////////////////////////////////////////////////
2  // Title      : Face Recognition
3  // Project    :
4  ////////////////////////////////////////////////////////////////////////////
5  // File       : frconsole.c
6  // Author     : Kristian Ambrosch, Peter Tummeltshammer
7  // Company    : TU Wien
8  // Created    : 2007/05/08
9  // Last update: 2007/05/29
10 // Platform   : SPEAR2
11 ////////////////////////////////////////////////////////////////////////////
12 // Description: This Application takes as input an image and a filter and
13 // calculates the Peak to Sidelobe Ratio.
14 // For more info see: http://ti.tuwien.ac.at/ecs/teaching/courses/hwswcode_lu
15 ////////////////////////////////////////////////////////////////////////////
16 // Copyright (c) 2007 TU Wien
17 ////////////////////////////////////////////////////////////////////////////
18
19 //squareRoot: Compute the Square Root - math.h does not exist on SPEAR
20 int32_t squareRoot(int32_t value)
21 {
22   int32_t xn,xn_old;
23   int16_t counter;
24
25   xn = value;
26   xn_old = 0;
27
28   for (counter = 0; (counter < 100) && (xn_old != xn); counter++)
29     {
30       xn_old = xn;
31       xn = (xn_old + (value/xn_old)) >> 1;
32     }
33
34   return xn;
35 }
36
37 //TransposeImage: Calculates the transposed matrix of an image
38 //necessary for rotating the image during FFT
39 void TransposeImage(int16_t *image)
40 {
41   int16_t temp[ImageWidth * ImageHeight];
42   int16_t counter1,counter2;
43
44   //Calculate Transposed Matrix
45   for (counter1 = 0; counter1 < ImageHeight; counter1 ++)
46   {
47     for (counter2 = 0; counter2 < ImageWidth; counter2++)
48     {
49       temp[counter1 + ImageWidth * counter2] = image[ImageWidth * counter1 + counter2];
50     }
51   }
52
53   //Copy Result back to Buffer
54   for (counter1 = 0; counter1 < ImageHeight; counter1 ++)
55   {
56     for (counter2 = 0; counter2 < ImageWidth; counter2++)
57     {
58       image[counter1 + ImageWidth * counter2] = temp[counter1 + ImageWidth * counter2];
59     }
60   }
61 }
62
63 //findMax: Returns the peak value of a matrix and stores its coordinates in x and y
64 int16_t findMax (int16_t *matrix, int16_t *x, int16_t *y)
65 {
66   int16_t counter1, counter2, max, akt;
67
68   *x = 0;
69   *y = 0;
70   max = matrix[0];
71
```

```
72    for (counter1 = 0; counter1 < ImageHeight; counter1 ++)
73    {
74      for (counter2 = 0; counter2 < ImageWidth; counter2++)
75      {
76        akt = matrix[ImageWidth * counter1 + counter2];
77        if (akt > max)
78        {
79          *x=counter1;
80          *y=counter2;
81          max = akt;
82        }
83      }
84    }
85    return max;
86  }
87
88  //complexMult: Performs the multiplication of two complex values
89  void complexMult(int16_t *realX, int16_t *imagX, int16_t *realY, int16_t *imagY,
90                   int16_t scale)
91  {
92    int16_t realXtemp = (*realX);
93
94    *realX = ((((int32_t) (*realX)) * ((int32_t) (*realY))) -
95              (((int32_t) (*imagX)) * ((int32_t) (*imagY)))) >> scale;
96    *imagX = ((((int32_t) realXtemp) * ((int32_t) (*imagY))) +
97              (((int32_t) (*imagX)) * ((int32_t) (*realY)))) >> scale;
98  }
99
100 //GetMulMaxBits: Eximates the maximum possible bits,
101 //that would result from a multiplication of x and y
102 int16_t getMulMaxBits(int16_t x, int16_t y)
103 {
104   int16_t counter;
105   for (counter = 0; (x > 0) && (counter < 16); counter ++)
106   {
107     x = x >> 1;
108   }
109   x = counter;
110
111   for (counter = 0; (y > 0) && (counter < 16); counter ++)
112   {
113     y = y >> 1;
114   }
115   y = counter;
116
117   if ((x==0) || (y==0))
118     return 0;
119   else if (x==1)
120     return y;
121   else if (y==1)
122     return x;
123   else
124     return x + y;
125 }
126
127 //complexPointMatrixMult: Calculates the pointwise Multiplication of two matrices
128 void complexPointMatrixMult(int16_t *realX, int16_t *imagX,
129                             int16_t *realY, int16_t *imagY)
130 {
131   int16_t counter1,counter2;
132   int16_t scale,maxbits,aktbits;
133
134   scale = 0;
135   maxbits = 0;
136
137   //Estimate the max. resulting bits after the complex multiplication
138   for (counter1 = 0; counter1 < ImageHeight; counter1 ++)
139   {
140     for (counter2 = 0; counter2 < ImageWidth; counter2++)
141     {
142       // + 1, because the addition after the multiplications
143       //can incresase value only by one bit.
144       aktbits = 1 + getMulMaxBits(realX[counter1 * ImageHeight + counter2],
145                                   realY[counter1 * ImageHeight + counter2]);
146       if (aktbits > maxbits)
147         maxbits = aktbits;
148
149       aktbits = 1 + getMulMaxBits(imagX[counter1 * ImageHeight + counter2],
150                                   imagY[counter1 * ImageHeight + counter2]);
151       if (aktbits > maxbits)
152         maxbits = aktbits;
153
154       aktbits = 1 + getMulMaxBits(realX[counter1 * ImageHeight + counter2],
```

```
155                                     imagY[counter1 * ImageHeight + counter2]);
156       if (aktbits > maxbits)
157         maxbits = aktbits;
158
159       aktbits = 1 + getMulMaxBits(imagX[counter1 * ImageHeight + counter2],
160                                     realY[counter1 * ImageHeight + counter2]);
161       if (aktbits > maxbits)
162         maxbits = aktbits;
163     }
164   }
165
166   //Dynamic Scaling if more than 16 Bits estimated
167   if (maxbits > 16)
168     scale = maxbits - 16;
169
170
171
172   //Perform the pointwise multiplication
173   for (counter1 = 0; counter1 < ImageHeight; counter1 ++)
174   {
175     for (counter2 = 0; counter2 < ImageWidth; counter2++)
176     {
177       complexMult(realX + counter1 * ImageHeight + counter2,
178         imagX + counter1 * ImageHeight + counter2,
179         realY + counter1 * ImageHeight + counter2,
180         imagY + counter1 * ImageHeight + counter2,
181         scale);
182     }
183   }
184 }
```

B Instruction Set Reference

This appendix provides a detailed guide to the instruction set architecture of SPEAR2.

B.1 Overview

This section provides an overview of available instructions.

Instruction	Operands	Description
1. ldli	rX, IMM8	Load low byte immediate
2. ldhi	rX, IMM8	Load high byte immediate
3. ldliu	rX, IMM8	Load low byte immediate without sign extension
4. ldfpw	rX, IMM6	MEM(FPTRW + IMM6) → rX
5. ldfpx	rX, IMM6	MEM(FPTRX + IMM6) → rX
6. ldfpy	rX, IMM6	MEM(FPTRY + IMM6) → rX
7. ldfpz	rX, IMM6	MEM(FPTRZ + IMM6) → rX
8. stfpw	rX, IMM6	rX → MEM(FPTRW + IMM6)
9. stfpx	rX, IMM6	rX → MEM(FPTRX + IMM6)
10. stfpy	rX, IMM6	rX → MEM(FPTRY + IMM6)
11. stfpz	rX, IMM6	rX → MEM(FPTRZ + IMM6)
12. ldfpw_inc	rX, IMM5	MEM(FPTRW + IMM5) → rX, FPTRW ++
13. ldfpx_inc	rX, IMM5	MEM(FPTRX + IMM5) → rX, FPTRX ++
14. ldfpy_inc	rX, IMM5	MEM(FPTRY + IMM5) → rX, FPTRY ++
15. ldfpz_inc	rX, IMM5	MEM(FPTRZ + IMM5) → rX, FPTRZ ++
16. stfpw_inc	rX, IMM5	rX → MEM(FPTRW + IMM5), FPTRW ++
17. stfpx_inc	rX, IMM5	rX → MEM(FPTRX + IMM5), FPTRX ++
18. stfpy_inc	rX, IMM5	rX → MEM(FPTRY + IMM5), FPTRY ++
19. stfpz_inc	rX, IMM5	rX → MEM(FPTRZ + IMM5), FPTRZ ++
20. ldfpw_dec	rX, IMM5	MEM(FPTRW + IMM5) → rX, FPTRW −−
21. ldfpx_dec	rX, IMM5	MEM(FPTRX + IMM5) → rX, FPTRX −−
22. ldfpy_dec	rX, IMM5	MEM(FPTRY + IMM5) → rX, FPTRY −−
23. ldfpz_dec	rX, IMM5	MEM(FPTRZ + IMM5) → rX, FPTRZ −−
24. stfpw_dec	rX, IMM5	rX → MEM(FPTRW + IMM5), FPTRW −−
25. stfpx_dec	rX, IMM5	rX → MEM(FPTRX + IMM5), FPTRX −−
26. stfpy_dec	rX, IMM5	rX → MEM(FPTRY + IMM5), FPTRY −−
27. stfpz_dec	rX, IMM5	rX → MEM(FPTRZ + IMM5), FPTRZ −−
28. cmpi_eq	rX, IMM7	Compare immediate equal
29. cmp_eq	rX, rY	Compare equal
30. cmpi_lt	rX, IMM7	Compare immediate less than

Instruction	Operands	Description
31. cmp_lt	rX, rY	Compare less than
32. cmpi_gt	rX, rY	Compare immediate greater than
33. cmp_gt	rX, rY	Compare greater than
34. cmpun_lt	rX, rY	Compare unsigned less than
35. cmpun_gt	rX, rY	Compare unsigned greater than
36. btest	rX, IMM5	Bit test
37. bset	rX, IMM5	Bit set
38. bset_ct	rX, IMM5	Bit set if cond-flag true
39. bset_cf	rX, IMM5	Bit set if cond-flag false
40. bclr	rX, IMM5	Bit clear
41. bclr_ct	rX, IMM5	Bit clear if cond-flag true
42. bclr_cf	rX, IMM5	Bit clear if cond-flag false
43. sl	rX, rY	Shift left
44. sl_ct	rX, rY	Shift left if cond-flag true
45. sl_cf	rX, rY	Shift left if cond-flag false
46. sli	rX, IMM4	Shift left immediate
47. sli_ct	rX, IMM4	Shift left immediate if cond-flag true
48. sli_cf	rX, IMM4	Shift left immediate if cond-flag false
49. sr	rX, rY	Shift right
50. sr_ct	rX, rY	Shift right if cond-flag true
51. sr_cf	rX, rY	Shift right if cond-flag false
52. sri	rX, IMM4	Shift right immediate
53. sri_ct	rX, IMM4	Shift right immediate if cond-flag true
54. sri_cf	rX, IMM4	Shift right immediate if cond-flag false
55. sra	rX, rY	Shift right arithmetic
56. sra_ct	rX, rY	Shift right arithmetic if cond-flag true
57. sra_cf	rX, rY	Shift right arithmetic if cond-flag false
58. srai	rX, IMM4	Shift right arithmetic immediate
59. srai_ct	rX, IMM4	Shift right arithmetic immediate if cond-flag true
60. srai_cf	rX, IMM4	Shift right arithmetic immediate if cond-flag false
61. rrc	rX	Rotate right with carry
62. rrc_ct	rX	Rotate right with carry if cond-flag true
63. rrc_cf	rX	Rotate right with carry if cond-flag false
64. mov	rX, rY	rY → rX
65. mov_ct	rX, rY	rY → rX if cond-flag true
66. mov_cf	rX, rY	rY → rX if cond-flag false
67. addi	rX, IMM6	rX + IMM6 → rX
68. addi_ct	rX, IMM6	rX + IMM6 → rX if cond-flag true

Instruction	Operands	Description
69. addi_cf	rX, IMM6	rX + IMM6 → rX if cond-flag false
70. add	rX, rY	rX + rY → rX
71. add_ct	rX, rY	rX + rY → rX if cond-flag true
72. add_cf	rX, rY	rX + rY → rX if cond-flag false
73. addc	rX, rY	rX + rY + Carry → rX
74. addc_ct	rX, rY	rX + rY + Carry → rX if cond-flag true
75. addc_cf	rX, rY	rX + rY + Carry → rX if cond-flag false
76. sub	rX, rY	rX - rY → rX
77. sub_ct	rX, rY	rX - rY → rX if cond-flag true
78. sub_cf	rX, rY	rX - rY → rX if cond-flag false
79. subc	rX, rY	rX - rY - Carry → rX
80. subc_ct	rX, rY	rX - rY - Carry → rX if cond-flag true
81. subc_cf	rX, rY	rX - rY - Carry → rX if cond-flag false
82. and	rX, rY	rX and rY → rX
83. and_ct	rX, rY	rX and rY → rX if cond-flag true
84. and_cf	rX, rY	rX and rY → rX if cond-flag false
85. or	rX, rY	rX or rY → rX
86. or_ct	rX, rY	rX or rY → rX if cond-flag true
87. or_cf	rX, rY	rX or rY → rX if cond-flag false
88. eor	rX, rY	rX xor rY → rX
89. eor_ct	rX, rY	rX xor rY → rX if cond-flag true
90. eor_cf	rX, rY	rX xor rY → rX if cond-flag false
91. not	rX	rX → ¬rX
92. not_ct	rX	rX → ¬rX if cond-flag true
93. not_cf	rX	rX → ¬rX if cond-flag false
94. neg	rX	rX → −rX
95. neg_ct	rX	rX → −rX if cond-flag true
96. neg_cf	rX	rX → −rX if cond-flag false
97. trap	IMM4	Call trap
98. trap_ct	IMM4	Call trap if cond-flag true
99. trap_cf	IMM4	Call trap if cond-flag false
100. jsr	rX	Jump to subroutine
101. jsr_ct	rX	Jump to subroutine if cond-flag true
102. jsr_cf	rX	Jump to subroutine if cond-flag false
103. jmpi	a10	Jump immediate
104. jmpi_ct	a10	Jump immediate if cond-flag true
105. jmpi_cf	a10	Jump immediate if cond-flag false
106. jmp	rX	Jump

Instruction	Operands	Description
107. jmp_ct	rX	Jump if cond-flag true
108. jmp_cf	rX	Jump if cond-flag false
109. ldw	rX, rY	Load word
110. ldh	rX, rY	Load half word
111. ldhu	rX, rY	Load half word unsigned
112. ldb	rX, rY	Load byte
113. ldbu	rX, rY	Load byte unsigned
114. stw	rX, rY	Store word
115. sth	rX, rY	Store half word
116. stb	rX, rY	Store byte
117. rts		Return from subroutine
118. rte		Return from exception
119. ldvec	rX, IMM5	Load vector
120. stvec	rX, IMM5	Store vector
121. nop		No operation
122. illop		Illegal opcode

B.2 Description

This section provides a detailed reference of the SPEAR2 instruction set. The following notations are used to describe instruction format:

- **n**: An immediate value, embedded in the instruction word
- **r**: One of the general purpose registers

The following notation conventions are used to describe instruction operation:

Notation	Meaning
`PC`	Program Counter
`SR`	Status Register
`SSR`	Saved Status Register
`VECTAB()`	The exception vector table
`cond-flag`	The condition flag of the status register
`rX, rY`	One of the 16 general purpose registers
`IMMn`	An n-bit immediate value, embedded in the instruction word
`MSB(X)`	Most Significant Bit of X
`LSB(X)`	Least Significant Bit of X
`X + Y`	Add
`X - Y`	Subtract
`X >> n`	The value X after being right-shifted n bit positions
`X << n`	The value X after being rleft-shifted n bit positions
`X & Y`	Bitwise logical AND
`X | Y`	Bitwise logical OR
`X ^ Y`	Bitwise logical XOR
`~X`	Bitwise logical NOT
`MEM32(X)`	The word located in data memory at byte-address X
`MEM16(X)`	The halfword located in data memory at bate-address X
`MEM8(X)`	The byte located in data memory at bate-address X
`sign_extend(X)`	Sign-extend X

1. **ldli**
   load low byte immediate

   Instruction Format

0	0	0	0	n	n	n	n	n	n	n	n	r	r	r	r
Opcode				IMM8								rX			

   Syntax `ldli rX, IMM8`
   Semantics *16/32-bit*
   `rX = sign_extend(IMM8)`
   Description
   Loads the IMM8 value into the register rX. IMM8 is written to the lower end of rX. The remaining bits of rX are set as the MSB of IMM8.

2. **ldhi**
   load high byte immediate

   Instruction Format

0	0	0	1	n	n	n	n	n	n	n	n	r	r	r	r
Opcode				IMM8								rX			

   Syntax `ldhi rX, IMM8`
   Semantics *16/32-bit*
   `rX = rX & 0x0f`
   `rX = rX | (sign_extend(IMM8) << 8)`
   Description
   Loads the IMM8 value into the register rX. IMM8 is written to the second byte of rX. The low byte of rX will remain unaffected. The remaining bits are set as the MSB of IMM8.

3. **ldliu**
   load low byte immediate without sign extension

   Instruction Format

0	0	1	0	n	n	n	n	n	n	n	n	r	r	r	r
Opcode				IMM8								rX			

Syntax `ldliu rX, IMM8`
Semantics
16/32-bit
```
rX = rX & ~(0x0f)
rX = rX | IMM8
```

Description
Loads the IMM8 value into the register rX. IMM8 is written to the second byte of rX. The low byte of rX will remain unaffected. The remaining bits are set as the MSB of IMM8.

4. ldfpw
load (half)word with frame pointer W

Instruction Format

0	1	1	0	0	0	n	n	n	n	n	n	r	r	r	r
Opcode						IMM6						rX			

Syntax `ldfpw rX, IMM6`
Semantics
16-bit
```
rX = MEM16(FPTRW + IMM6)
```
32-bit
```
rX = MEM32(FPTRW + IMM6)
```

Description
Loads a word or halfword form the memory location that results from adding the value in frame pointer register W and the instruction's signed 6-bit immediate value. The data is placed in register rX. The data width used for memory access is determined by the processor configuration.

5. ldfpx
load (half)word with frame pointer X

Instruction Format

0	1	1	0	0	1	n	n	n	n	n	n	r	r	r	r
Opcode						IMM6						rX			

Syntax `ldfpx rX, IMM6`
Semantics
16-bit

```
rX = MEM16(FPTRX + IMM6)
  32-bit
rX = MEM32(FPTRX + IMM6)
```

Description

Loads a word or halfword form the memory location that results from adding the value in frame pointer register X and the instruction's signed 6-bit immediate value. The data is placed in register rX. The data width used for memory access is determined by the processor configuration.

6. ldfpy

load (half)word with frame pointer Y

Instruction Format

0	1	1	0	1	0	n	n	n	n	n	n	r	r	r	r
Opcode						IMM6						rX			

Syntax

```
ldfpy rX, IMM6
```

Semantics

```
  16-bit
rX = MEM16(FPTRY + IMM6)
  32-bit
rX = MEM32(FPTRY + IMM6)
```

Description

Loads a word or halfword form the memory location that results from adding the value in frame pointer register Y and the instruction's signed 6-bit immediate value. The data is placed in register rX. The data width used for memory access is determined by the processor configuration.

7. ldfpz

load (half)word with frame pointer Z

Instruction Format

0	1	1	0	1	1	n	n	n	n	n	n	r	r	r	r
Opcode						IMM6						rX			

Syntax `ldfpz rX, IMM6`
Semantics
16-bit
`rX = MEM16(FPTRZ + IMM6)`
32-bit
`rX = MEM32(FPTRZ + IMM6)`

Description
Loads a word or halfword form the memory location that results from adding the value in frame pointer register Z and the instruction's signed 6-bit immediate value. The data is placed in register rX. The data width used for memory access is determined by the processor configuration.

8. **stfpw**
store (half)word with frame pointer W

Instruction Format

0	1	1	1	0	0	n	n	n	n	n	n	r	r	r	r
Opcode						IMM6						rX			

Syntax `stfpw rX, IMM6`
Semantics
16-bit
`MEM16(FPTRW + IMM6) = rX`
32-bit
`MEM32(FPTRW + IMM6) = rX`

Description
Stores the content of register rX into memory at the address specified by the sum of frame pointer register W and the instruction's signed 6-bit immediate value.

9. **stfpx**
store (half)word with frame pointer X

Instruction Format

0	1	1	1	0	1	n	n	n	n	n	n	r	r	r	r
Opcode						IMM6						rX			

Syntax `stfpx rX, IMM6`

Semantics
16-bit
`MEM16(FPTRX + IMM6) = rX`
32-bit
`MEM32(FPTRX + IMM6) = rX`

Description
Stores the content of register rX into memory at the address specified by the sum of frame pointer register X and the instruction's signed 6-bit immediate value.

10. stfpy
store (half)word with frame pointer Y

Instruction Format

0	1	1	1	1	0	n	n	n	n	n	n	r	r	r	r
Opcode						IMM6						rX			

Syntax `stfpy rX, IMM6`

Semantics
16-bit
`MEM16(FPTRY + IMM6) = rX`
32-bit
`MEM32(FPTRY + IMM6) = rX`

Description
Stores the content of register rX into memory at the address specified by the sum of frame pointer register Y and the instruction's signed 6-bit immediate value.

11. stfpz
store (half)word with frame pointer Z

Instruction Format

0	1	1	1	1	1	n	n	n	n	n	n	r	r	r	r
Opcode						IMM6						rX			

Syntax `stfpz rX, IMM6`

Semantics
16-bit
`MEM16(FPTRZ + IMM6) = rX`

32-bit

```
MEM32(FPTRZ + IMM6) = rX
```

Description

Stores the content of register rX into memory at the address specified by the sum of frame pointer register Z and the instruction's signed 6-bit immediate value.

12. ldfpw_inc

load (half)word with frame pointer W and increment W

Instruction Format

0	1	0	0	0	0	0	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax

```
ldfpw_inc rX, IMM5
```

Semantics

16-bit

```
rX = MEM16(FPTRW + IMM5)
FPTRW = FPTRW + 2
```

32-bit

```
rX = MEM32(FPTRW + IMM5)
FPTRW = FPTRW + 4
```

Description

Loads a word or halfword form the memory location that results from adding the value in frame pointer register W and the instruction's signed 5-bit immediate value. The data is placed in register rX. Afterwards frame pointer register W is incremented by 2 or 4. The value for incrementing the frame pointer register and data width used for memory access is determined by the processor configuration.

13. ldfpx_inc

load (half)word with frame pointer X and increment X

Instruction Format

0	1	0	0	0	1	0	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax
```
ldfpx_inc rX, IMM5
```
Semantics
```
  16-bit
rX = MEM16(FPTRX + IMM5)
FPTRX = FPTRX + 2
  32-bit
rX = MEM32(FPTRX + IMM5)
FPTRX = FPTRX + 4
```
Description

Loads a word or halfword form the memory location that results from adding the value in frame pointer register X and the instruction's signed 5-bit immediate value. The data is placed in register rX. Afterwards frame pointer register X is incremented by 2 or 4. The value for incrementing the frame pointer register and data width used for memory access is determined by the processor configuration.

14. ldfpy_inc

load (half)word with frame pointer Y and increment Y

Instruction Format

0	1	0	0	1	0	0	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax
```
ldfpy_inc rX, IMM5
```
Semantics
```
  16-bit
rX = MEM16(FPTRY + IMM5)
FPTRY = FPTRY + 2
  32-bit
rX = MEM32(FPTRY + IMM5)
FPTRY = FPTRY + 4
```
Description

Loads a word or halfword form the memory location that results from adding the value in frame pointer register Y and the instruction's signed 5-bit immediate value. The data is placed in register rX. Afterwards frame pointer register Y is incremented by 2 or 4. The value for incrementing the frame pointer register and data width used for memory access is determined by the processor configuration.

15. ldfpz_inc

load (half)word with frame pointer Z and increment Z

Instruction Format

0	1	0	0	1	1	0	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax `ldfpz_inc rX, IMM5`

Semantics

16-bit

```
rX = MEM16(FPTRZ + IMM5)
FPTRZ = FPTRZ + 2
```

32-bit

```
rX = MEM32(FPTRZ + IMM5)
FPTRZ = FPTRZ + 4
```

Description

Loads a word or halfword form the memory location that results from adding the value in frame pointer register Z and the instruction's signed 5-bit immediate value. The data is placed in register rX. Afterwards frame pointer register Z is incremented by 2 or 4. The value for incrementing the frame pointer register and data width used for memory access is determined by the processor configuration.

16. stfpw_inc

store (half)word with frame pointer W and increment W

Instruction Format

0	1	0	1	0	0	0	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax `stfpw_inc rX, IMM5`

Semantics

16-bit

```
MEM16(FPTRW + IMM5) = rX
FPTRW = FPTRW + 2
```

32-bit

```
MEM32(FPTRW + IMM5) = rX
FPTRW = FPTRW + 4
```

Description

Stores the content of register rX into memory at the address specified by the sum of frame pointer register W and the instruction's signed 6-bit immediate value. Afterwards frame pointer register W is incremented by 2 or 4, depending on the processor configuration.

17. stfpx_inc

store (half)word with frame pointer X and increment X

Instruction Format

0	1	0	1	0	1	0	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax
```
stfpx_inc rX, IMM5
```
Semantics
```
  16-bit
MEM16(FPTRX + IMM5) = rX
FPTRX = FPTRX + 2
  32-bit
MEM32(FPTRX + IMM5) = rX
FPTRX = FPTRX + 4
```

Description

Stores the content of register rX into memory at the address specified by the sum of frame pointer register X and the instruction's signed 6-bit immediate value. Afterwards frame pointer register X is incremented by 2 or 4, depending on the processor configuration.

18. stfpy_inc

store (half)word with frame pointer Y and increment Y

Instruction Format

0	1	0	1	1	0	0	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax
```
stfpy_inc rX, IMM5
```
Semantics
```
  16-bit
MEM16(FPTRY + IMM5) = rX
FPTRY = FPTRY + 2
  32-bit
MEM32(FPTRY + IMM5) = rX
FPTRY = FPTRY + 4
```

Description

Stores the content of register rX into memory at the address specified by the sum of frame pointer register Y and the instruction's signed 6-bit immediate value. Afterwards frame pointer register Y is incremented by 2 or 4, depending on the processor configuration.

19. stfpz_inc

store (half)word with frame pointer Z and increment Z

Instruction Format

0	1	0	1	1	1	0	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax
```
stfpz_inc rX, IMM5
```
Semantics
```
  16-bit
MEM16(FPTRZ + IMM5) = rX
FPTRZ = FPTRZ + 2
  32-bit
MEM32(FPTRZ + IMM5) = rX
FPTRZ = FPTRZ + 4
```
Description

Stores the content of register rX into memory at the address specified by the sum of frame pointer register Z and the instruction's signed 6-bit immediate value. Afterwards frame pointer register Z is incremented by 2 or 4, depending on the processor configuration.

20. ldfpw_dec

load (half)word with frame pointer W and decrement W

Instruction Format

0	1	0	0	0	0	1	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax
```
ldfpw_dec rX, IMM5
```
Semantics
```
  16-bit
rX = MEM16(FPTRW + IMM5)
FPTRW = FPTRW - 2
  32-bit
rX = MEM32(FPTRW + IMM5)
FPTRW = FPTRW - 4
```
Description

Loads a word or halfword form the memory location that results from adding the value in frame pointer register W and the instruction's signed 5-bit immediate value. The data is placed in register rX. Afterwards frame pointer register W is decremented by 2 or 4. The value for decrementing the frame pointer register and data width used for memory access is determined by the processor configuration.

21. ldfpx_dec

load (half)word with frame pointer X and decrement X

Instruction Format

0	1	0	0	0	1	1	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax
Semantics

```
ldfpx_dec rX, IMM5
  16-bit
rX = MEM16(FPTRX + IMM5)
FPTRX = FPTRX - 2
  32-bit
rX = MEM32(FPTRX + IMM5)
FPTRX = FPTRX - 4
```

Description

Loads a word or halfword form the memory location that results from adding the value in frame pointer register X and the instruction's signed 5-bit immediate value. The data is placed in register rX. Afterwards frame pointer register X is decremented by 2 or 4. The value for decrementing the frame pointer register and data width used for memory access is determined by the processor configuration.

22. ldfpy_dec

load (half)word with frame pointer Y and decrement Y

Instruction Format

0	1	0	0	1	0	1	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax
```
ldfpy_dec rX, IMM5
```
Semantics
```
  16-bit
rX = MEM16(FPTRY + IMM5)
FPTRY = FPTRY - 2
  32-bit
rX = MEM32(FPTRY + IMM5)
FPTRY = FPTRY - 4
```
Description

Loads a word or halfword form the memory location that results from adding the value in frame pointer register Y and the instruction's signed 5-bit immediate value. The data is placed in register rX. Afterwards frame pointer register Y is decremented by 2 or 4. The value for decrementing the frame pointer register and data width used for memory access is determined by the processor configuration.

23. ldfpz_dec

load (half)word with frame pointer Z and decrement Z

Instruction Format

0	1	0	0	1	1	1	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax
```
ldfpz_dec rX, IMM5
```
Semantics
```
  16-bit
rX = MEM16(FPTRZ + IMM5)
FPTRZ = FPTRZ - 2
  32-bit
rX = MEM32(FPTRZ + IMM5)
FPTRZ = FPTRZ - 4
```
Description

Loads a word or halfword form the memory location that results from adding the value in frame pointer register Z and the instruction's signed 5-bit immediate value. The data is placed in register rX. Afterwards frame pointer register Z is decremented by 2 or 4. The value for decrementing the frame pointer register and data width used for memory access is determined by the processor configuration.

24. stfpw_dec

store (half)word with frame pointer W and decrement W

Instruction Format

0	1	0	1	0	0	1	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax `stfpw_dec rX, IMM5`

Semantics

16-bit

```
MEM16(FPTRW + IMM5) = rX
FPTRW = FPTRW - 2
```

32-bit

```
MEM32(FPTRW + IMM5) = rX
FPTRW = FPTRW - 4
```

Description

Stores the content of register rX into memory at the address specified by the sum of frame pointer register W and the instruction's signed 6-bit immediate value. Afterwards frame pointer register W is decremented by 2 or 4, depending on the processor configuration.

25. stfpx_dec

store (half)word with frame pointer X and decrement X

Instruction Format

0	1	0	1	0	1	1	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax `stfpx_dec rX, IMM5`

Semantics

16-bit

```
MEM16(FPTRX + IMM5) = rX
FPTRX = FPTRX - 2
```

32-bit

```
MEM32(FPTRX + IMM5) = rX
FPTRX = FPTRX - 4
```

Description

Stores the content of register rX into memory at the address specified by the sum of frame pointer register X and the instruction's signed 6-bit immediate value. Afterwards frame pointer register X is decremented by 2 or 4, depending on the processor configuration.

26. stfpy_dec

store (half)word with frame pointer Y and decrement Y

Instruction Format

0	1	0	1	1	0	1	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax
```
stfpy_dec rX, IMM5
```
Semantics
```
  16-bit
MEM16(FPTRY + IMM5) = rX
FPTRY = FPTRY - 2
  32-bit
MEM32(FPTRY + IMM5) = rX
FPTRY = FPTRY - 4
```

Description

Stores the content of register rX into memory at the address specified by the sum of frame pointer register Y and the instruction's signed 6-bit immediate value. Afterwards frame pointer register Y is decremented by 2 or 4, depending on the processor configuration.

27. stfpz_dec

store (half)word with frame pointer Z and decrement Z

Instruction Format

0	1	0	1	1	1	1	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax
```
stfpz_dec rX, IMM5
```
Semantics
```
  16-bit
MEM16(FPTRZ + IMM5) = rX
FPTRZ = FPTRZ - 2
  32-bit
MEM32(FPTRZ + IMM5) = rX
FPTRZ = FPTRZ - 4
```

Description

Stores the content of register rX into memory at the address specified by the sum of frame pointer register Z and the instruction's signed 6-bit immediate value. Afterwards frame pointer register Z is decremented by 2 or 4, depending on the processor configuration.

28. **cmpi_eq**
compare equal immediate

Instruction Format

1	0	1	1	1	n	n	n	n	n	n	n	r	r	r	r
Opcode					IMM7							rX			

Syntax `cmpi_eq rX, IMM7`
Semantics
16/32-bit
```
if (rX - IMM7 == 0)
cond-flag = 1
else cond-flag = 0
```
Description
Compares the value in rX with the sign-extended immediate, setting the condition flag if they are equal, otherwise the condition flag is cleared.

29. **cmp_eq**
compare equal

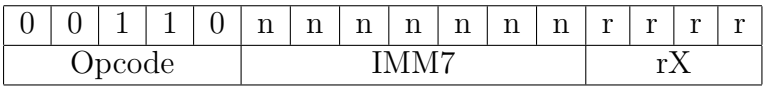
Instruction Format

1	0	1	1	0	0	0	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `cmp_eq rX, rY`
Semantics
16/32-bit
```
if (rX == rY)
cond-flag = 1
else cond-flag = 0
```
Description
Compares the value in rX with the value in rY, setting the condition flag if they are equal, otherwise the condition flag is cleared.

30. **cmpi_lt**
compare less than immediate

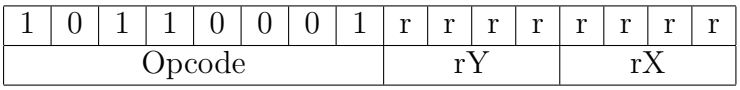
Instruction Format



Syntax `cmpi_lt rX, IMM7`
Semantics
16/32-bit
```
if (rX < IMM7)
cond-flag = 1
else cond-flag = 0
```
Description
Compares the value in rX with the sign-extended immediate, setting the condition flag if they value in IMM7 is greater than the value in rX, otherwise the condition flag is cleared.

31. cmp_lt
compare less than

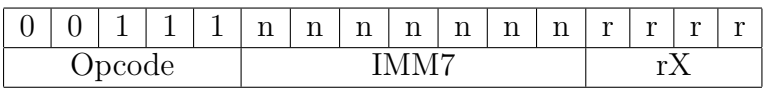
Instruction Format



Syntax `cmp_lt rX, rY`
Semantics
16/32-bit
```
if (rX < rY)
cond-flag = 1
else cond-flag = 0
```
Description
Compares the value in rX with the value in rY, setting the condition flag if they value in rY is greater than the value in rX, otherwise the condition flag is cleared.

32. cmpi_gt
compare greater than immediate

Instruction Format



Syntax `cmpi_gt rX, IMM7`
Semantics
16/32-bit
```
if (rX > IMM7)
cond-flag = 1
else cond-flag = 0
```
Description
Compares the value in rX with the sign-extended immediate, setting the condition flag if they value in rX is greater than the value in IMM7, otherwise the condition flag is cleared.

33. cmp_gt
compare greater than

Instruction Format

1	0	1	1	0	0	1	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `cmpi_gt rX, rY`
Semantics
16/32-bit
```
if (rX > rY)
cond-flag = 1
else cond-flag = 0
```
Description
Compares the value in rX with the value in rY, setting the condition flag if they value in rX is greater than the value in rY, otherwise the condition flag is cleared.

34. cmpu_lt
compare less than unsigned

Instruction Format

1	0	1	1	0	0	1	1	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `cmpu_lt rX, rY`
Semantics
16/32-bit
```
if (rX < rY)
```

```
cond-flag = 1
else cond-flag = 0
```

Description

Compares the value in rX with the value in rY, setting the condition flag if they value in rY is greater than the value in rX, otherwise the condition flag is cleared. Both operands are treated as unsigned intergers.

35. cmpu_gt

compare greater than unsigned

Instruction Format

1	0	1	1	0	1	0	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax
```
cmpu_gt rX, rY
```
Semantics
16/32-bit
```
if (rX > rY)
cond-flag = 1
else cond-flag = 0
```

Description

Compares the value in rX with the value in rY, setting the condition flag if they value in rX is greater than the value in rY, otherwise the condition flag is cleared. Both operands are treated as unsigned intergers.

36. btest

bit test

Instruction Format

1	0	1	1	0	1	1	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax
```
btest rX, IMM5
```
Semantics
16-bit
```
if (rX(IMM4) == 1)
cond-flag = 1
```

```
else cond-flag = 0
  32-bit
if (rX(IMM5) == 1)
cond-flag = 1
else cond-flag = 0
```

Description

Tests the bit in rX on specific position determined by the immediate value. The condition flag is set, if the specific bit is set. Otherwise the condition flag is cleared. The width of the immediate value depends on the processor configuration.

37. bset

bit set

Instruction Format

1	0	1	0	0	1	0	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax

```
bset rX, IMM5
```

Semantics

```
  16-bit
rX = rX | (1 << IMM4)
  32-bit
rX = rX | (1 << IMM5)
```

Description

Sets one bit in rX. The position is specified by the immediate value. The width of the immediate value depends on the processor configuration.

38. bset_ct

bit set if cond-flag true

Instruction Format

1	0	0	1	0	1	0	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax

```
bset_ct rX, IMM5
```

Semantics

```
  16-bit
```

```
if (cond-flag == 1)
rX = rX | (1 << IMM4)
  32-bit
if (cond-flag == 1)
rX = rX | (1 << IMM5)
```

Description

Sets one bit in rX, if condition flag is set. The position is specified by the immediate value. The width of the immediate value depends on the processor configuration.

39. bset_cf

bit set if cond-flag false

Instruction Format

1	0	0	0	0	1	0	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax

```
bset_cf rX, IMM5
```

Semantics

```
  16-bit
if (cond-flag == 0)
rX = rX | (1 << IMM4)
  32-bit
if (cond-flag == 0)
rX = rX | (1 << IMM5)
```

Description

Sets one bit in rX, if condition flag is cleared. The position is specified by the immediate value. The width of the immediate value depends on the processor configuration.

40. bclr

bit clear

Instruction Format

1	0	1	0	0	1	1	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax
```
bclr rX, IMM5
```
Semantics
16-bit
```
rX = rX & ~(1 << IMM4)
```
32-bit
```
rX = rX & ~(1 << IMM5)
```

Description

Clears one bit in rX. The position is specified by the immediate value. The width of the immediate value depends on the processor configuration.

41. bclr_ct

bit clear if cond-flag true

Instruction Format

1	0	0	1	0	1	1	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax
```
bclr_ct rX, IMM5
```
Semantics
16-bit
```
if (cond-flag == 1)
rX = rX & ~(1 << IMM4)
```
32-bit
```
if (cond-flag == 1)
rX = rX & ~(1 << IMM5)
```

Description

Clears one bit in rX, if condition flag is set. The position is specified by the immediate value. The width of the immediate value depends on the processor configuration.

42. bclr_cf

bit clear if cond-flag false

Instruction Format

1	0	0	0	0	1	1	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax `bclr_cf rX, IMM5`

Semantics

16-bit

```
if (cond-flag == 0)
rX = rX & ~(1 << IMM4)
```

32-bit

```
if (cond-flag == 0)
rX = rX & ~(1 << IMM5)
```

Description

Clears one bit in rX, if condition flag is cleared. The position is specified by the immediate value. The width of the immediate value depends on the processor configuration.

43. sl

shift left

Instruction Format

1	0	1	0	0	0	0	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `sl rX, rY`

Semantics

16-bit

```
rX = rX << (rY & 0x0f)
```

32-bit

```
rX = rX << (rY & 0x1f)
```

Description

Shifts the value in rX left by the number of bits specified by the value in rY.

44. sl_ct

shift left if cond-flag true

Instruction Format

1	0	0	1	0	0	0	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `sl_ct rX, rY`

Semantics

16-bit

```
if (cond-flag == 1)
rX = rX << (rY & 0x0f)
  32-bit
if (cond-flag == 1)
rX = rX << (rY & 0x1f)
```

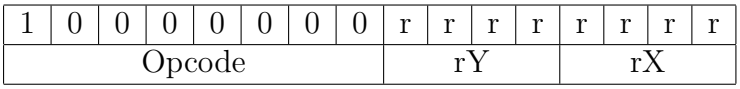
Description

Shifts the value in rX left, if condition flag is set. The number of bits is given by the value in rY.

45. sl_cf

shift left if cond-flag false

Instruction Format



1	0	0	0	0	0	0	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax

```
sl_cf rX, rY
```

Semantics

```
  16-bit
if (cond-flag == 0)
rX = rX << (rY & 0x0f)
  32-bit
if (cond-flag == 0)
rX = rX << (rY & 0x1f)
```

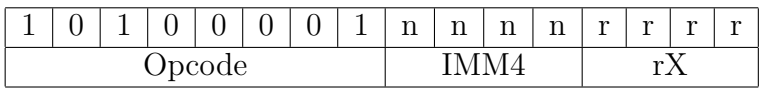
Description

Shifts the value in rX left, if condition flag is cleared. The number of bits is given by the value in rY.

46. sli

shift left immediate

Instruction Format



1	0	1	0	0	0	0	1	n	n	n	n	r	r	r	r
Opcode								IMM4				rX			

Syntax

```
sli rX, IMM4
```

Semantics

```
  16/32-bit
rX = rX << IMM4
```

Description
Shifts the value in rX left by the number of bits specified by the immediate value.

47. **sli_ct**
shift left immediate if cond-flag true

Instruction Format

1	0	0	1	0	0	0	1	n	n	n	n	r	r	r	r
Opcode								IMM4				rX			

Syntax `sli_ct rX, IMM4`
Semantics
16/32-bit
```
if (cond-flag == 1)
rX = rX << IMM4
```
Description
Shifts the value in rX left, if condition flag is set. The number of bits is given by the immediate value.

48. **sli_cf**
shift left immediate if cond-flag false

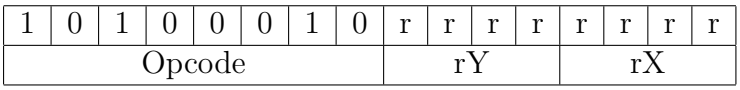
Instruction Format

1	0	0	0	0	0	0	1	n	n	n	n	r	r	r	r
Opcode								IMM4				rX			

Syntax `sli_cf rX, IMM4`
Semantics
16/32-bit
```
if (cond-flag == 0)
rX = rX << IMM4
```
Description
Shifts the value in rX left, if condition flag is cleared. The number of bits is given by the immediate value.

49. **sr**
shift right

Instruction Format



Syntax `sr rX, rY`

Semantics

16-bit

```
rX = rX >> (rY & 0x0f)
```

32-bit

```
rX = rX >> (rY & 0x1f)
```

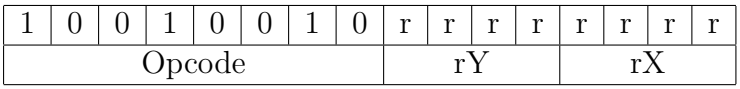
Description

Shifts the unsigned value in rX right by the number of bits specified by the value in rY.

50. **sr_ct**

shift right if cond-flag true

Instruction Format



Syntax `sr_ct rX, rY`

Semantics

16-bit

```
if (cond-flag == 1)
rX = rX >> (rY & 0x0f)
```

32-bit

```
if (cond-flag == 1)
rX = rX >> (rY & 0x1f)
```

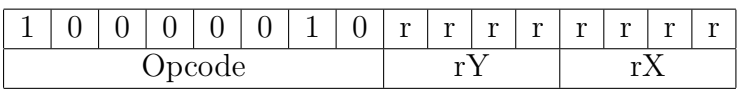
Description

Shifts the unsigned value in rX right, if condition flag is set. The number of bits is given by the value in rY.

51. **sr_cf**

shift right if cond-flag false

Instruction Format



Syntax
```
sr_cf rX, rY
```
Semantics
16-bit
```
if (cond-flag == 0)
rX = rX >> (rY & 0x0f)
```
32-bit
```
if (cond-flag == 0)
rX = rX >> (rY & 0x1f)
```
Description
Shifts the unsigned value in rX right, if condition flag is cleared. The number of bits is given by the value in rY.

52. sri
shift right immediate

Instruction Format

1	0	1	0	0	0	1	1	n	n	n	n	r	r	r	r
Opcode								IMM4				rX			

Syntax
```
sri rX, IMM4
```
Semantics
16/32-bit
```
rX = rX >> IMM4
```
Description
Shifts the unsigned value in rX right by the number of bits specified by the immediate value.

53. sri_ct
shift right immediate if cond-flag true

Instruction Format

1	0	0	1	0	0	1	1	n	n	n	n	r	r	r	r
Opcode								IMM4				rX			

Syntax
```
sri_ct rX, IMM4
```
Semantics
16/32-bit
```
if (cond-flag == 1)
rX = rX >> IMM4
```
Description

Shifts the unsigned value in rX right, if condition flag is set. The number of bits is given by the immediate value.

54. sri_cf

shift right immediate if cond-flag false

Instruction Format

1	0	0	0	0	0	1	1	n	n	n	n	r	r	r	r
Opcode								IMM4				rX			

Syntax
```
sri_cf rX, IMM4
```
Semantics
16/32-bit
```
if (cond-flag == 0)
rX = rX >> IMM4
```
Description

Shifts the unsigned value in rX right, if condition flag is cleared. The number of bits is given by the immediate value.

55. sra

shift right arithmetic

Instruction Format

1	1	1	0	1	0	0	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

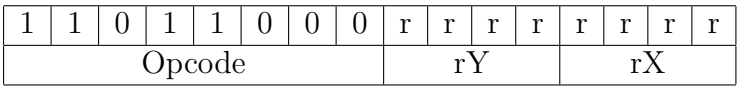
Syntax
```
sra rX, rY
```
Semantics
16-bit
```
rX = rX >> (rY & 0x0f)
```
32-bit
```
rX = rX >> (rY & 0x1f)
```
Description

Shifts the signed value in rX right by the number of bits specified by the value in rY.

56. sra_ct

shift right arithmetic if cond-flag true

Instruction Format



Syntax `sra_ct rX, rY`

Semantics

16-bit

```
if (cond-flag == 1)
rX = rX >> (rY & 0x0f)
```

32-bit

```
if (cond-flag == 1)
rX = rX >> (rY & 0x1f)
```

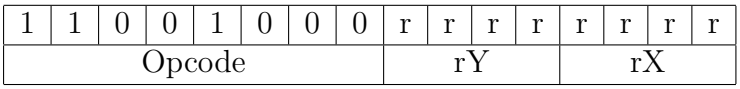
Description

Shifts the signed value in rX right, if condition flag is set. The number of bits is given by the value in rY.

57. sra_cf

shift right arithmetic if cond-flag false

Instruction Format



Syntax `sra_cf rX, rY`

Semantics

16-bit

```
if (cond-flag == 0)
rX = rX >> (rY & 0x0f)
```

32-bit

```
if (cond-flag == 0)
rX = rX >> (rY & 0x1f)
```

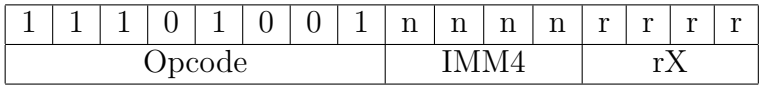
Description

Shifts the signed value in rX right, if condition flag is cleared. The number of bits is given by the value in rY.

58. srai

shift right arithmetic immediate

Instruction Format



Syntax `srai rX, IMM4`
Semantics
16/32-bit
```
rX = rX >> IMM4
```
Description
Shifts the signed value in rX right by the number of bits specified by the immediate value.

59. srai_ct
shift right arithmetic immediate if cond-flag true

Instruction Format

1	1	0	1	1	0	0	1	n	n	n	n	r	r	r	r
Opcode								IMM4				rX			

Syntax `srai_ct rX, IMM4`
Semantics
16/32-bit
```
if (cond-flag == 1)
rX = rX >> IMM4
```
Description
Shifts the signed value in rX right, if condition flag is set. The number of bits is given by the immediate value.

60. srai_cf
shift right arithmetic immediate if cond-flag false

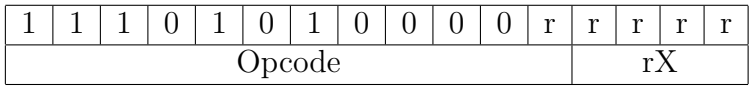
Instruction Format

1	1	0	0	1	0	0	1	n	n	n	n	r	r	r	r
Opcode								IMM4				rX			

Syntax `srai_cf rX, IMM4`
Semantics
16/32-bit
```
if (cond-flag == 0)
rX = rX >> IMM4
```
Description
Shifts the signed value in rX right, if condition flag is cleared. The number of bits is given by the immediate value.

61. **rrc**
rotate right with carry

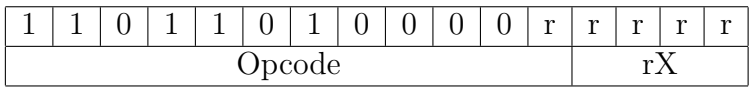
Instruction Format



Syntax
```
rrc rX
```
Semantics
16/32-bit
```
rX = rX >> 1
MSB(rX) = carry
carry = LSB(rX)
```
Description
Rotates the value in rX right by one bit. At the same time the MSB of rX is written with the carry value and the LSB of rX is stored to carry.

62. **rrc_ct**
rotate right with carry if cond-flag true

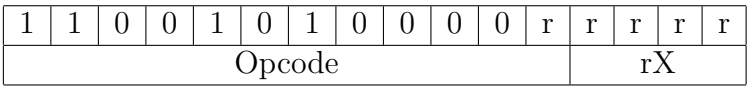
Instruction Format



Syntax
```
rrc_ct rX
```
Semantics
16/32-bit
```
if (cond-flag == 1)
rX = rX >> 1
MSB(rX) = carry
carry = LSB(rX)
```
Description
Rotates the value in rX right by one bit, if condition flag is set. At the same time the MSB of rX is written with the carry value and the LSB of rX is stored to carry.

63. **rrc_cf**
rotate right with carry if cond-flag true

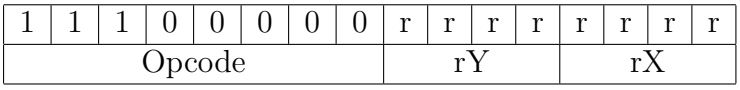
Instruction Format



Syntax `rrc_cf rX`
Semantics *16/32-bit*
```
if (cond-flag == 0)
rX = rX >> 1
MSB(rX) = carry
carry = LSB(rX)
```
Description
Rotates the value in rX right by one bit, if condition flag is cleared. At the same time the MSB of rX is written with the carry value and the LSB of rX is stored to carry.

64. mov
move

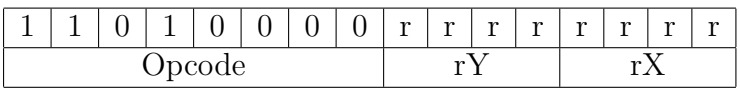
Instruction Format



Syntax `mov rX, rY`
Semantics *16/32-bit*
```
rX = rX
```
Description
Moves the value in rY to rX.

65. mov_ct
move if cond-flag true

Instruction Format



Syntax `mov_ct rX, rY`
Semantics *16/32-bit*
```
if (cond-flag == 1)
```

```
rX = rX
```

Description

Moves the value in rY to rX, if condition flag is set.

66. mov_cf

move if cond-flag false

Instruction Format

1	1	0	0	0	0	0	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `mov_cf rX, rY`

Semantics

16/32-bit

```
if (cond-flag == 0)
rX = rX
```

Description

Moves the value in rY to rX, if condition flag is cleared.

67. addi

add immediate

Instruction Format

1	0	1	0	1	0	n	n	n	n	n	n	r	r	r	r
Opcode						IMM6						rX			

Syntax `addi rX, IMM6`

Semantics

16/32-bit

```
rX = rX + sign_extend(IMM6)
```

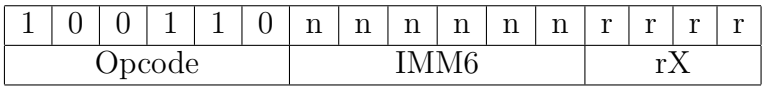
Description

Calculates the sum of rX and the sign-extended immediate. Stores the result in rX.

68. addi_ct

add immediate if cond-flag true

Instruction Format



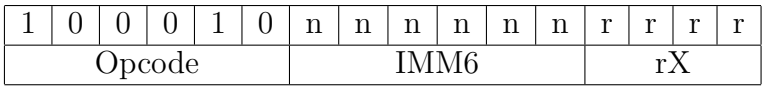
Syntax `addi_ct rX, IMM6`
Semantics *16/32-bit*

```
if (cond-flag == 1)
rX = rX + sign_extend(IMM6)
```

Description
Calculates the sum of rX and the sign-extended immediate, if condition flag is set. Stores the result in rX.

69. addi_cf
add immediate if cond-flag false

Instruction Format



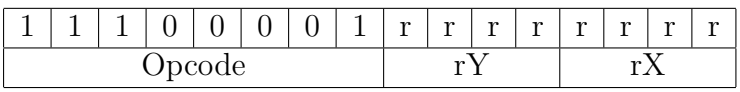
Syntax `addi_cf rX, IMM6`
Semantics *16/32-bit*

```
if (cond-flag == 0)
rX = rX + sign_extend(IMM6)
```

Description
Calculates the sum of rX and the sign-extended immediate, if condition flag is cleared. Stores the result in rX.

70. add
add

Instruction Format



Syntax `add rX, rY`
Semantics *16/32-bit*

```
rX = rX + rY
```

Description
Calculates the sum of rX and rY. Stores the result in rX.

71. add_ct

add if cond-flag true

Instruction Format

1	1	0	1	0	0	0	1	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `add_ct rX, rY`

Semantics
16/32-bit

```
if (cond-flag == 1)
rX = rX + rY
```

Description

Calculates the sum of rX and rY, if condition flag is set. Stores the result in rX.

72. add_cf

add if cond-flag false

Instruction Format

1	1	0	0	0	0	0	1	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `add_cf rX, rY`

Semantics
16/32-bit

```
if (cond-flag == 0)
rX = rX + rY
```

Description

Calculates the sum of rX and rY, if condition flag is cleared. Stores the result in rX.

73. addc

add with carry

Instruction Format

1	1	1	0	0	0	1	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `addc rX, rY`
Semantics *16/32-bit*
`rX = rX + rY + carry`
Description
Calculates the sum of rX, rY, and carry. Stores the result in rX.

74. addc_ct
add with carry if cond-flag true

Instruction Format

1	1	0	1	0	0	1	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `addc_ct rX, rY`
Semantics *16/32-bit*
```
if (cond-flag == 1)
rX = rX + rY + carry
```
Description
Calculates the sum of rX, rY, and carry, if condition flag is set. Stores the result in rX.

75. addc_cf
add with carry if cond-flag false

Instruction Format

1	1	0	0	0	0	1	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `addc_cf rX, rY`
Semantics *16/32-bit*
```
if (cond-flag == 0)
rX = rX + rY + carry
```
Description
Calculates the sum of rX, rY, and carry, if condition flag is cleared. Stores the result in rX.

76. sub

subtract

Instruction Format

1	1	1	0	0	0	1	1	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `sub rX, rY`

Semantics

16/32-bit

```
rX = rX - rY
```

Description

Subtracts rY from rX. Stores the result in rX.

77. sub_ct

subtract if cond-flag true

Instruction Format

1	1	0	1	0	0	1	1	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `sub_ct rX, rY`

Semantics

16/32-bit

```
if (cond-flag == 1)
rX = rX - rY
```

Description

Subtracts rY from rX, if condition flag is set. Stores the result in rX.

78. sub_cf

subtract if cond-flag false

Instruction Format

1	1	0	0	0	0	1	1	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `sub_cf rX, rY`
Semantics *16/32-bit*
```
if (cond-flag == 0)
rX = rX - rY
```
Description
Subtracts rY from rX, if condition flag is cleared. Stores the result in rX.

79. subc
subtract with carry

Instruction Format

1	1	1	0	0	1	0	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `subc rX, rY`
Semantics *16/32-bit*
```
rX = rX - rY - carry
```
Description
Subtracts rY from rX. Considers the carry flag and stores the result in rX.

80. subc_ct
subtract with carry if cond-flag true

Instruction Format

1	1	0	1	0	1	0	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `subc_ct rX, rY`
Semantics *16/32-bit*
```
if (cond-flag == 1)
rX = rX - rY - carry
```
Description
Subtracts rY from rX, if condition flag is set. Considers the carry flag and stores the result in rX.

81. subc_cf

subtract with carry if cond-flag false

Instruction Format

1	1	0	0	0	1	0	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `subc_cf rX, rY`

Semantics

16/32-bit

```
if (cond-flag == 0)
rX = rX - rY - carry
```

Description

Subtracts rY from rX, if condition flag is cleared. Considers the carry flag and stores the result in rX.

82. and

bitwise logical and

Instruction Format

1	1	1	0	0	1	0	1	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `and rX, rY`

Semantics

16/32-bit

```
rX = rX & rY
```

Description

Caluclates the bitwise logical AND of rX and rY. Stores the result in rX.

83. and_ct

bitwise logical and if cond-flag true

Instruction Format

1	1	0	1	0	1	0	1	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `and_ct rX, rY`
Semantics *16/32-bit*
```
if (cond-flag == 1)
rX = rX & rY
```
Description
Caluclates the bitwise logical AND of rX and rY, if condition flag is set. Stores the result in rX.

84. and_cf
bitwise logical and if cond-flag false

Instruction Format

1	1	0	0	0	1	0	1	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `and_cf rX, rY`
Semantics *16/32-bit*
```
if (cond-flag == 0)
rX = rX & rY
```
Description
Caluclates the bitwise logical AND of rX and rY, if condition flag is cleared. Stores the result in rX.

85. or
bitwise logical or

Instruction Format

1	1	1	0	0	1	1	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `or rX, rY`
Semantics *16/32-bit*
```
rX = rX | rY
```
Description
Caluclates the bitwise logical OR of rX and rY. Stores the result in rX.

86. **or_ct**
bitwise logical or if cond-flag true

Instruction Format

1	1	0	1	0	1	1	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `or_ct rX, rY`
Semantics
16/32-bit
```
if (cond-flag == 1)
rX = rX | rY
```
Description
Caluclates the bitwise logical OR of rX and rY, if condition flag is set. Stores the result in rX.

87. **or_cf**
bitwise logical or if cond-flag false

Instruction Format

1	1	0	0	0	1	1	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `or_cf rX, rY`
Semantics
16/32-bit
```
if (cond-flag == 0)
rX = rX | rY
```
Description
Caluclates the bitwise logical OR of rX and rY, if condition flag is cleared. Stores the result in rX.

88. **eor**
bitwise logical exclusive or

Instruction Format

1	1	1	0	0	1	1	1	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `eor rX, rY`
Semantics *16/32-bit*
```
rX = rX ^ rY
```
Description
Caluclates the bitwise logical exclusive OR of rX and rY. Stores the result in rX.

89. eor_ct
bitwise logical exclusive or if cond-flag true

Instruction Format

1	1	0	1	0	1	1	1	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `eor_ct rX, rY`
Semantics *16/32-bit*
```
if (cond-flag == 1)
rX = rX ^ rY
```
Description
Caluclates the bitwise logical exclusive OR of rX and rY, if condition flag is set. Stores the result in rX.

90. eor_cf
bitwise logical exclusive or if cond-flag false

Instruction Format

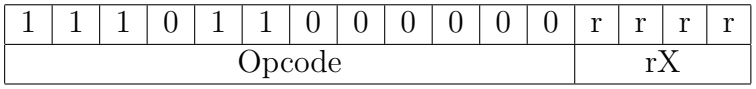
1	1	0	0	0	1	1	1	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `eor_cf rX, rY`
Semantics *16/32-bit*
```
if (cond-flag == 0)
rX = rX ^ rY
```
Description
Caluclates the bitwise logical exclusive OR of rX and rY, if condition flag is cleared. Stores the result in rX.

91. not

bitwise logical not

Instruction Format



Syntax `not rX`

Semantics

16/32-bit

```
rX = ~rX
```

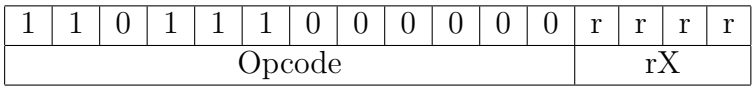
Description

Caluclates the bitwise complement of rX. Stores the result in rX.

92. not_ct

bitwise logical not if cond-flag true

Instruction Format



Syntax `not_ct rX`

Semantics

16/32-bit

```
if (cond-flag == 1)
rX = ~rX
```

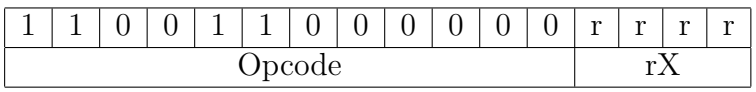
Description

Caluclates the bitwise complement of rX, if condition flag is set. Stores the result in rX.

93. not_cf

bitwise logical not if cond-flag false

Instruction Format



Syntax `not_cf rX`
Semantics *16/32-bit*
```
if (cond-flag == 0)
rX = ~rX
```
Description
Caluclates the bitwise complement of rX, if condition flag is cleared. Stores the result in rX.

94. neg
negative

Instruction Format

1	1	1	0	1	1	0	1	0	0	0	0	r	r	r	r
Opcode												rX			

Syntax `neg rX`
Semantics *16/32-bit*
```
rX = ~rX + 1
```
Description
Caluclates the negative of rX. Stores the result in rX.

95. neg_ct
negative if cond-flag true

Instruction Format

1	1	0	1	1	1	0	1	0	0	0	0	r	r	r	r
Opcode												rX			

Syntax `neg_ct rX`
Semantics *16/32-bit*
```
if (cond-flag == 1)
rX = ~rX + 1
```
Description
Caluclates the negative of rX, if condition flag is set. Stores the result in rX.

96. neg_cf

negative if cond-flag false

Instruction Format

1	1	0	0	1	1	0	1	0	0	0	0	r	r	r	r
Opcode												rX			

Syntax
```
neg_cf rX
```
Semantics
16/32-bit
```
if (cond-flag == 0)
rX = ~rX + 1
```

Description

Caluclates the negative of rX, if condition flag is cleared. Stores the result in rX.

97. trap

trap

Instruction Format

1	1	1	0	1	0	1	1	n	n	n	n	r	r	r	r
Opcode								IMM4				rX			

Syntax
```
trap rX, IMM4
```
Semantics
16/32-bit
```
r15 = PC
PC = VECTAB(IMM4)
SSR = SR
SR = 0
```

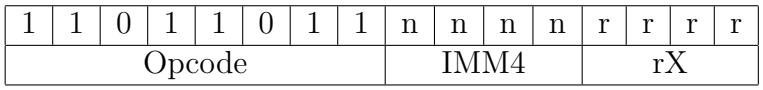
Description

Raises a system call exception, saves the status register, and saves the address of the next instruction in register r15. Afterwards the status register is cleared and thereby interrupts are disabled.

98. trap_ct

trap if cond-flag true

Instruction Format



Syntax `trap_ct rX, IMM4`

Semantics

16/32-bit

```
if (cond-flag == 1)
r15 = PC
PC = VECTAB(IMM4)
SSR = SR
SR = 0
```

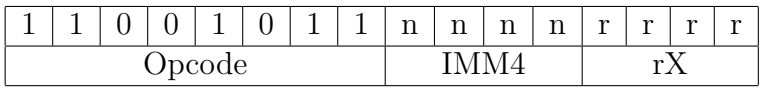
Description

Raises a system call exception, if condition flag is set, saves the status register, and saves the address of the next instruction in register r15. Afterwards the status register is cleared and thereby interrupts are disabled.

99. trap_cf

trap if cond-flag false

Instruction Format



Syntax `trap_cf rX, IMM4`

Semantics

16/32-bit

```
if (cond-flag == 0)
r15 = PC
PC = VECTAB(IMM4)
SSR = SR
SR = 0
```

Description

Raises a system call exception, if condition flag is cleared, saves the status register, and saves the address of the next instruction in register r15. Afterwards the status register is cleared and thereby interrupts are disabled.

100. jsr

jump to subroutine

Instruction Format

1	1	1	0	1	1	1	0	0	0	0	0	r	r	r	r
Opcode												rX			

Syntax `jsr rX`
Semantics *16/32-bit*
`r14 = PC`
`PC = rX`

Description
Saves the address of the next instruction in register r14. Loads the address in rX to the program counter.

101. jsr_ct
jump to subroutine if cond-flag true

Instruction Format

1	1	0	1	1	1	1	0	0	0	0	0	r	r	r	r
Opcode												rX			

Syntax `jsr_ct rX`
Semantics *16/32-bit*
`if (cond-flag == 1)`
`r14 = PC`
`PC = rX`

Description
Saves the address of the next instruction in register r14, if condition flag is set. And loads the address in rX to the program counter.

102. jsr_cf
jump to subroutine if cond-flag false

Instruction Format

1	1	0	0	1	1	1	0	0	0	0	0	r	r	r	r
Opcode												rX			

Syntax `jsr_cf rX`
Semantics *16/32-bit*
`if (cond-flag == 0)`

```
r14 = PC
PC = rX
```

Description

Saves the address of the next instruction in register r14, if condition flag is cleared. And loads the address in rX to the program counter.

103. jmpi

jump immediate

Instruction Format

1	0	1	0	1	1	n	n	n	n	n	n	n	n	n	n
Opcode						IMM10									

Syntax `jmpi IMM10`

Semantics *16/32-bit*

```
PC = PC + IMM10
```

Description

Performs a branch to the address given by the sum of the program counter and the sign-extended immediate.

104. jmpi_ct

jump immediate if cond-flag true

Instruction Format

1	0	0	1	1	1	n	n	n	n	n	n	n	n	n	n
Opcode						IMM10									

Syntax `jmpi_ct IMM10`

Semantics *16/32-bit*

```
if (cond-flag == 1)
PC = PC + IMM10
```

Description

Performs a branch, if condition flag is set. The destination address is given by the sum of the program counter and the sign-extended immediate.

105. jmpi_cf
jump immediate if cond-flag false

Instruction Format

1	0	0	0	1	1	n	n	n	n	n	n	n	n	n	n
Opcode						IMM10									

Syntax
```
jmpi_cf IMM10
```
Semantics
16/32-bit
```
if (cond-flag == 0)
PC = PC + IMM10
```
Description
Performs a branch, if condition flag is cleared. The destination address is given by the sum of the program counter and the sign-extended immediate.

106. jmp
jump

Instruction Format

1	1	1	0	1	1	1	1	0	0	0	0	r	r	r	r
Opcode												rX			

Syntax
```
jmp rX
```
Semantics
16/32-bit
```
PC = rX
```
Description
Loads the address in rX to the program counter.

107. jmp_ct
jump if cond-flag true

Instruction Format

1	1	0	1	1	1	1	1	0	0	0	0	r	r	r	r
Opcode												rX			

Syntax `jmp_ct rX`
Semantics *16/32-bit*
`if (cond-flag == 1)`
`PC = rX`

Description
Loads the address in rX to the program counter, if condition flag is set.

108. jmp_cf
jump if cond-flag false

Instruction Format

1	1	0	0	1	1	1	1	0	0	0	0	r	r	r	r
Opcode												rX			

Syntax `jmp_cf rX`
Semantics *16/32-bit*
`if (cond-flag == 0)`
`PC = rX`

Description
Loads the address in rX to the program counter, if condition flag is cleared.

109. ldw
load word

Instruction Format

1	1	1	1	0	0	0	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `ldw rX, rY`
Semantics *16/32-bit*
`rX = MEM32(rY)`

Description
Loads a word from memory and stores the result in rX. The address is given by rY. This instruction is only available, with enlarged data path.

110. ldh
load halfword

Instruction Format

1	1	1	1	0	0	0	1	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `ldh rX, rY`
Semantics *16/32-bit*
`rX = MEM16(rY)`
Description
Loads a halfword from memory and stores the sign-extended result in rX. The address is given by rY.

111. ldhu
load halfword unsigned

Instruction Format

1	1	1	1	0	0	1	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `ldhu rX, rY`
Semantics *16/32-bit*
`rX = MEM16(rY)`
Description
Loads a halfword from memory and stores the zero-extended result in rX. The address is given by rY. This instruction is only available, with enlarged data path.

112. ldb
load byte

Instruction Format

1	1	1	1	0	0	1	1	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `ldb rX, rY`
Semantics *16/32-bit*
`rX = MEM8(rY)`
Description
Loads a byte from memory and stores the sign-extended result in rX. The address is given by rY.

113. ldbu
load byte unsigned

Instruction Format

1	1	1	1	0	1	0	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `ldbu rX, rY`
Semantics *16/32-bit*
`rX = MEM8(rY)`
Description
Loads a byte from memory and stores the zero-extended result in rX. The address is given by rY.

114. stw
store word

Instruction Format

1	1	1	1	0	1	0	1	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `stw rX, rY`
Semantics *16/32-bit*
`MEM32(rY) = rX`
Description
Stores rX to memory. The address is given by rY. This instruction is only available, with enlarged data path.

115. sth
store halfword

Instruction Format

1	1	1	1	0	1	1	0	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `sth rX, rY`
Semantics *16/32-bit*
`MEM16(rY) = rX`
Description
Stores the low halfword of rX to memory. The address is given by rY.

116. stb
store byte

Instruction Format

1	1	1	1	0	1	1	1	r	r	r	r	r	r	r	r
Opcode								rY				rX			

Syntax `stb rX, rY`
Semantics *16/32-bit*
`MEM8(rY) = rX`
Description
Stores the low byte of rX to memory. The address is given by rY.

117. rts
return from subroutine

Instruction Format

1	1	1	1	1	0	0	0	0	0	0	0	1	1	1	0
Opcode												r14			

Syntax `rts`
Semantics *16/32-bit*
`PC = r14`
Description
Performs a jump to the address saved in r14.

118. rte

return from exception

Instruction Format

1	1	1	1	1	0	0	1	0	0	0	0	1	1	1	1
Opcode												r14			

Syntax `rte`

Semantics

16/32-bit

```
PC = r15
SR = SSR
```

Description

Performs a jump to the address saved in r15 and restores the status register.

119. ldvec

load vector

Instruction Format

1	1	1	1	1	0	1	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax `ldvec rX, IMM5`

Semantics

16/32-bit

```
rX = VECTAB(IMM5)
```

Description

Reads the vector on position IMM5 from the vector table and stores the result in rX.

120. stvec

store vector

Instruction Format

1	1	1	1	1	0	0	n	n	n	n	n	r	r	r	r
Opcode							IMM5					rX			

Syntax `stvec rX, IMM5`
Semantics *16/32-bit*
`VECTAB(IMM5) = rX`
Description
Stores a vector to the vector table.

121. nop
no operation

Instruction Format

1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
Opcode															

Syntax `nop`
Semantics *16/32-bit*
`-`
Description
nop does nothing.

122. illop
illegal operation

Instruction Format

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Opcode															

Syntax `illop`
Semantics *16/32-bit*
`-`
Description
Raises the illop exception.

References

[1] Altera Corporation, 101 Innovation Drive - San Jose, CA 95134, USA. *Nios II©Processor Reference Handbook*, 2007.

[2] ARM Ltd., 110 Fulbourn Road - Cambridge, CB1 9NJ, United Kingdom. *AMBA Specification rev 2.0*, 1999.

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

[4] 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.

[5] M. Delvai, W. Huber, B. Rahbaran, and A. Steininger. SPEAR - Designentscheidungen für den Scalable Processor for Embedded Applications in Real-time Environments. *In Austrochip 2001*, Tagungsband:25–32, October 2001.

[6] W. Huber. Spezifikation der Schnittstelle zwischen Extension-Modulen und SPEAR. Technical report, TU Wien, Institut für Technische Informatik, Vienna, 2001.

[7] Lattice Semiconductor Corporation, 5555 NE Moore Court - Hillsboro, OR 97124, USA. *LatticeMico32©Processor Reference Manual*, 2007.

[8] J. Mosser. AMBA4SPEAR2 - An AMBA extension module for the SPEAR2 processor core. Master's thesis, TU Wien, Institut für Technische Informatik, 2008.

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

[10] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design*. Morgan Kaufman Publishers, 3rd edition, 2005.

[11] P. Puschner and A. Burns. Writing Temporally Predictable Code. *In Proc. 7th IEEE International Workshop on Object-Oriented Real-time Dependable Systems*, pages 85–91, Januar 2002.

[12] J. Reichardt and B. Schwarz. *VHDL-Synthese*. Oldenbourg, 1st edition, 2000.

[13] Xilinx, Inc., 2100 Logic Drive - San Jose, CA 95124-3400, USA. *MicroBlaze©Processor Reference Guide*, 2007.

[14] Xilinx, Inc., 2100 Logic Drive - San Jose, CA 95124-3400, USA. *Xilinx©Synthesis Technology (XST) User Guide*, 2007.