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
Kreuzgasse 6A
2130 Mistelbach

Für meine Eltern
Danksagung
Kurzfassung
Abstract

A soft core processor is a configurable microcontroller defined in software. Such processors may 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 course of this master thesis, the soft core processor SPEAR2 has been developed. The SPEAR2 architecture is a 16/32-bit processor and based on SPEAR (Scalable Processor for Embedded Applications in Real-time Environments).

The motives for developing an improved version are versatile. Fitting the code to new target technologies, eliminating some disadvantages of SPEAR, enabling configurability, or just adding useful features like byte addressed memory.

To satisfy this goals, SPEAR2 was written from scratch. To provide adjustable memory sizes and the option to change the size of the data path, a configuration framework has been created. 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 sizes with other components of the processor. The chief difficulty was memory access and developing a consistent bus interface. For both configurations the same instruction set is used, enabling to use the same toolchain for both configurations.

SPEAR2 was developed with the aim to be an efficient 16-bit processor. If required, more computational power can be provided by extending the data path. The gained experience showed, a 16-bit processor with extended data path is not able to provide the performance of 32-bit processors, because of the limited instruction set. Without extended data path SPEAR2 acts as small and efficient processor, already used by some projects. The experience shows, that the 16-bit configuration is able to compete with other soft core processors.
Contents

1 Introduction 1
  1.1 Motivation ............................................. 1
  1.2 Outline ................................................. 3

2 State of the Art 4
  2.1 MicroBlaze .............................................. 4
    2.1.1 Overview ........................................... 5
    2.1.2 Instruction Set Architecture ...................... 5
    2.1.3 Registers ........................................... 6
    2.1.4 Pipeline Architecture ............................ 6
    2.1.5 Memory Architecture ............................. 7
    2.1.6 Exceptions ......................................... 8
  2.2 Nios II ................................................ 9
    2.2.1 Overview ........................................... 9
    2.2.2 Instruction Set Architecture ...................... 10
    2.2.3 Registers ........................................... 10
    2.2.4 Pipeline Architecture ............................ 10
    2.2.5 Memory Architecture ............................. 11
    2.2.6 Exceptions ......................................... 13
  2.3 LatticeMico32 .......................................... 14
    2.3.1 Overview ........................................... 14
    2.3.2 Instruction Set Architecture ...................... 15
    2.3.3 Registers ........................................... 16
    2.3.4 Pipeline Architecture ............................ 16
    2.3.5 Memory Architecture ............................. 17
    2.3.6 Exceptions ......................................... 18

3 SPEAR - Basis for a new Architecture 20
  3.1 Overview ................................................ 20
    3.1.1 Pipeline ........................................... 21
    3.1.2 Memory Architecture ............................. 22
  3.2 Exceptions ............................................. 22
CONTENTS

3.3 Register File ........................................... 22
   3.3.1 Frame Pointer Registers .......................... 23
   3.3.2 RTSX- and RTSY-Register .......................... 23
   3.3.3 RTE-Register ...................................... 24
3.4 Instruction Set Architecture .......................... 24
   3.4.1 Structure of Instructions .......................... 24
   3.4.2 Conditional Instructions ........................... 24
3.5 Extension Modules ................................... 26
   3.5.1 Processor Control Module ......................... 27
   3.5.2 Programmer Module ................................. 27

4 Analysing the Old Architecture ........................ 28
4.1 Three Processor Cores .................................. 28
4.2 Analysing SPEAR ........................................ 29

5 SPEAR2 .................................................. 31
5.1 Overview .............................................. 31
5.2 Customizable Data Path .................................. 32
   5.2.1 Implementation Overview ......................... 33
   5.2.2 Performance Improvement .......................... 33
   5.2.3 Addressable Memory ................................ 33
5.3 Processor Architecture ................................ 34
   5.3.1 First Stage ....................................... 34
   5.3.2 Second Stage ..................................... 35
   5.3.3 Third Stage ....................................... 36
   5.3.4 Fourth Stage ..................................... 39
5.4 Instruction Set Architecture .......................... 39
   5.4.1 Instruction Format ................................ 40
   5.4.2 Conditional Instructions ......................... 40
5.5 Implementation ........................................ 41
   5.5.1 Program Counter .................................. 41
   5.5.2 Instruction Memories .............................. 42
   5.5.3 Decoder ........................................... 43
## CONTENTS

5.5.4 Register File ......................................... 44
5.5.5 Forwarding Unit ....................................... 45
5.5.6 ALU ...................................................... 46
5.5.7 Frame Pointer .......................................... 47
5.5.8 Data Memory ........................................... 49
5.5.9 Memory Access ......................................... 52
5.5.10 Exceptions ............................................. 53
5.5.11 Sleep Mode ............................................ 54
5.5.12 Implementation Details ................................. 55
5.6 Extension Modules ........................................ 56
  5.6.1 System Control Module ................................. 58
  5.6.2 Programmer Module ..................................... 61
5.7 Differences: 16 vs. 32 bit Version ......................... 62
  5.7.1 Interface .............................................. 63
  5.7.2 Instruction Set Architecture ............................ 63
  5.7.3 Addressable Memory ................................... 64

6 Specification .................................................. 65
  6.1 Configuration ........................................... 65
  6.2 Interface ................................................ 66

7 Results ......................................................... 67

8 Conclusion ...................................................... 68

A Appendix - Instruction Set Reference .................... 69
  A.1 Overview ............................................... 69
  A.2 Description ............................................. 74
List of Figures

1. Block Diagram of SPEAR ........................................ 21
2. Exception Vector Table of SPEAR ............................... 22
3. Organization of a Frame ........................................ 23
4. Interface for Extension Modules ................................. 26
5. Block Diagram of SPEAR2 ....................................... 32
6. Parts Affected by Configuration ................................. 34
7. Performance of SPEAR2 .......................................... 35
8. The Fetch Stage in More Detail ................................. 36
9. The Decode Stage in More Detail ............................... 37
10. The Execute Stage in More Detail .............................. 38
11. The Write Back Stage in More Detail .......................... 39
12. The old Implementation of the Program Counter ............ 41
13. 8 bit Barrel Shifter ............................................ 47
14. Organization of Data Memory .................................... 50
15. Architecture of Data Memory .................................... 51
16. Generic Status Byte ............................................ 57
17. Generic Config Byte ............................................ 57
18. Interface of the System Control Module ....................... 59
19. Customized Status Byte of the System Control Module .... 60
20. Interface of the Programmer Module ........................... 61
21. Customized Config Byte of the System Control Module .... 62
List of Tables

1  Instruction Formats used by SPEAR  . . . . . . . . . 25
2  Configuration Options of SPEAR2  . . . . . . . . . . . 66
1 Introduction

In our life embedded systems play an important role. They are used inside products even where we do not expect them. Often a compound of embedded systems is used to enable the features of products. A car for example contains already many embedded systems and the usage is rapidly growing.

An embedded system comprises of hardware (e.g. processing unit, sensors, communication interfaces, etc.) and software. The operational areas for embedded systems are very different and so there is great diversity of embedded systems and there components. A lot of different products try to satisfy the market.

1.1 Motivation

The requirements to embedded systems are very different and so the solutions too. When starting a new embedded systems project an appropriate processor have to be chosen. For small applications a cheap 8-bit 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 as usual), then a dedicated processing unit (e.g. 32-bit embedded processors or digital signal processors (DSP)) may satisfy the needs. And between this two extremes different requirements specifications are possible. But dedicated hard cores have limited flexibility.

The opposite of a dedicated hard core is to use 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 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 take more time, but ASICs provide much more performance than an FPGA.

The concept of using a soft core processor has several advantages such as 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 diploma thesis is the soft core processor SPEAR2. The processor bases on its predecessor SPEAR [3, 4, 5] and is written from scratch. There was some reasons why we developed a sucessor. 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 slow Input/Output capability of SPEAR. Since elementary parts of the processor have 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

Chapter two gives an overview of state of the art soft core processors. For this purpose three different processor architectures were presented.

Chapter three gives detailed information about SPEAR, the predecessor of SPEAR2.

Chapter four is used to analyse the drawbacks of SPEAR and provides possible solutions.

Chapter five is the main part of this document. This chapter explains the architecture and implementation of SPEAR2 in detail.

Chapter six focuses on the specification of SPEAR2. The specification comprises the instruction set architecture and the hardware interface.

Chapter seven presents the results of this master thesis.

Chapter eight covers the conclusion.
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 choosen since they are widely used:

**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, with its advantages and disadvantages. As a result it is nearly impossible to use a soft core processor 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 configureable. For every processor a toolchain is available.

The reason why no 16-bit processor was choosen for introduction is simple: 16-bit soft core processors are hardly available. Only 8-bit and 32-bit processors. Every company mentioned above has also an 8-bit soft core processor in its product line. Since SPEAR2 is positioned bewtween 16-bit and 32-bit processors only 32-bit soft core processors were choosen.


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 MicroBlaze

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 busses 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 reg-
ister 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 instruc-
tions 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 up to eighteen 32-bit special purpose
registers, depending on configuration options.

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 vec-
tores.

- 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). The instruction and data memory ranges can be made to overlap by mapping them both to the same physical memory. The latter is useful for software debugging.

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 Rset 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, ”configureable” 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.

### 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 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.
2.2 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 tradeoff 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 using branch offset direction. A negative offset (backward branch) is predicted as taken, and a positive offset (forward branch) is predicted as not-taken. 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 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 more 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. 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. Include this custom instruction to reduce your program’s interrupt latency.

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. The worst case is a system with 32 interrupts.

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. By combining a 32-bit wide instruction set with 32 general-purpose registers, the LatticeMico32 provides the performance and flexibility suitable for a wide variety of markets, including communications, consumer, computer, medical, industrial, and automotive.

#### 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 hazzard 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 from 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 with 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 tp 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.
3 SPEAR - Basis for a new Architecture

SPEAR stands for Scalable Processor for Embedded Applications in Real-time environments. A detailed description is given in [3, 5, 4]. SPEAR was designed for embedded systems with real-time requirements. Thus two important features for such a processor are customizability and real-time capability.

Embedded systems are used in different applications with varying requirements. So it is impossible to have a processor which provide all requested features on the one hand and without wasting much resources on the other hand. To overcome this problem SPEAR is able to use extension modules for customization. The extension modules are mapped to uppermost data memory and can be accessed like normal memory with load/store operations.

Regarding real-time capability there are different abstraction levels for defining real-time. SPEAR supports real-time prediction for the lowest level, in particular for one clock cycle exactly. The duration of all operations inside the processor are known and all instructions have constant execution time. The instruction set comprises conditional instructions which have always the same execution time independent if they are executed or not. Also the reaction to interrupts is well defined.

3.1 Overview

SPEAR is an entirely 16 bit architecture. That means that all buses and registers are 16 bit in size. The design represents a RISC architecture which executes instructions by a 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 holds 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 Overview

3.1.1 Pipeline

The pipeline is build 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 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. Thereby the programming and the prediction of worst case execution time are simplified.

Figure 1: Block Diagram of SPEAR
3.1.2 Memory Architecture

The memory architecture was designed to be simple and the memory can only be addressed by word. This restriction simplifies the memory access and the memory organization. Furthermore twice as much memory can be addressed.

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 shown in figure 2. There are special instructions to build and manipulate the exception vector table. The table contains 32 entries. The trap vectors are stored at positive and interrupt vectors are stored at negative positions.

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. These special function registers are described below.
3.3 Register File

3.3.1 Frame Pointer Registers

These registers are used to build Frames. Frames are similar to stacks with the difference that data can be accessed at any memory location inside the frame without emptying the stack. An offset gives the final memory address. This feature is useful for accessing extension modules where some adjacent addresses will be accessed several times. The registers r26, r27 and r28 are used for this purpose whereby three independent frames can be managed. For using frame pointers, first the corresponding register has to be set as base address of the frame. After that the frame can be used with the intended instructions. The instructions can hold a 5 bit offset. Hence a frame comprises 32 words. Figure 3 shows a frame.

![Figure 3: Organization of a Frame](image)

3.3.2 RTSX- and RTSY-Register

Both registers are used to save the return address in case of a subroutine call. The registers r29 and r30 are used for this purpose whereby two nested subroutine calls are possible. 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 in size 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 [8, 9]. 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 for encoding instructions. Table 1 shows the instruction formats used by SPEAR. A list of instructions is given in [3].

3.4.2 Conditional Instructions

This type of instructions help to design embedded systems with real-time quality while allowing to utilize the One-Path programming paradigm [10]. 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 to provide 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;
### 3.4 Instruction Set Architecture

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Opcode</th>
<th>Register</th>
<th>Register</th>
<th>Constant</th>
<th>Register</th>
<th>Register</th>
<th>Constant</th>
<th>Register</th>
<th>Constant</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td>8 bits</td>
<td>5 bits</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

Table 1: Instruction Formats used by SPEAR

```plaintext
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*/
```

```plaintext
without conditional instructions

```
cmpi_eq r7, -3
jmp +4
mov r2, r3
mov r2, r4
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 -3 or not, the execution time is always the same. This data independent constant execution time is only possible with conditional instructions and is described by the One-Path programming paradigm mentioned above. 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 is possible when the if-the-else construct contains only few expressions.

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 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.

![Figure 4: Interface for Extension Modules](image-url)
There are three types of extension modules. The extension modules are classified by their functionality:

**System-Extension Module** which have an extended interface to the processor and provides functionality of the processor like a programmer.

**Function-Extension Module** which can be used to implement functions in hardware which otherwise have to be emulated by software.

**IO-Extension Module** is used to implement interfaces to the processor like RS232 or PS/2.

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 for downloading the program code to the instruction memory. In addition an extension module for communication is needed. Dividing this task to a programming and a communication part enables easy changing of the download interface 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 to identify the improvable respectively already proven features. Thus we obtain the needful information which features have to be improved. On the other side proven features should be taken over to the successor.

This chapter describes the first step of the SPEAR2 design process. In particular the reason for choosing neither NEEDLE nor LANCE but SPEAR as basis for a new architecture. Afterwards the weaknesses of SPEAR are analysed. Finally solutions for these weaknesses will be presented.

4.1 Three Processor Cores

The old approach for building a flexible architecture was based on three different processor cores. Namely NEEDLE, SPEAR and LANCE. All processor cores provide nearly the same features, execute the same source code and use the same extension modules.

The distinction between the three processor cores was done by different resource requirements, maximum clock frequency a core can operate and clock cycles per instruction. NEEDLE, a small processor with less performance and small resource consumption. SPEAR, designed for medium requirements.

And LANCE, a big processor with high performance.

But the problem of this approach were the fact 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. Whereas 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 processore core compared to NEEDLE and LANCE, it was selected as the basis for a new processor. But SPEAR had also several disadvantages which we tried to remove with 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 transfering 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.
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.

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 noticeable.

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 several positions at once will be possible.
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. Also one major feature, the customizable data path, is explained in detail. Afterwards the specification and implementation details are going to be treated.

5.1 Overview

SPEAR2 is the successor of SPEAR and stands for **Scalable Processor for Embedded Applications in Real-time environments**. Although it is the successor and a lot of features are adopted from SPEAR, the code is entirely written from scratch. The design represents a RISC architecture which executes instructions by a pipeline. The pipeline has four stages and supports full data forwarding between stages to prevent data hazards. In addition 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 different instructions. The size 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 size 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.
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.
5.2 Customizable Data Path

- 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 have 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. Figure 7 shows the performance of SPEAR2 compared to other architectures.

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 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 first stage of the pipeline is called fetch stage. Figure 8 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. The pipe register 1 is build by the program counter an the output of the memories.
5.3 Processor Architecture

Figure 7: Performance of SPEAR2

5.3.2 Second Stage

This stage is used for decoding the instruction. The stage is depicted by figure 9. First the decision to choose one instruction memory has to be made. This decision is controlled by the source-selection-signal of the programer module. The decoding task covers to determine the type of instruction and generate the opcode for the ALU, check if a jump has to be performed, extract the immediate value, generate addresses for the extended register file and exception vector table and according to this to set several control signals. If the previous instruction caused a jump, the decoded instruction will be discarded and the control signals will be set as if a NOP has been decoded. The case with the highest priority is an exception. Then the control signals are set to save the program counter and program status word. Only crucial control signals like the write enable signals are affected by a previous jump or an exception. For example the addresses for the extended register file are not affected signals. To change only crucial signals help to reduce the resource usage.

Beside decoding the instruction the program counter has to be saved for later use. As described in the next section the program counter of the decode stage will be used as return address in case of an exception. For correct functionality the jump destination calculated by the ALU has to be saved in case of a jump. If always the program counter of the fetch stage would be used, the execution time of a program would varies dependently if the exception happens during a jump or not.

The pipe register 2 is build by 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 second last stage executes the instruction. Figure 10 depicts this stage in detail. Because the processor has serveral pipeline stage it takes some time till the result of an operation is saved to the extended register file. To solve this problem a forwarding unit is used. The forwarding unit assures that the ALU is always feed with latest information and is described in detail in section 5.5.5. Another possibility instead of forwarding would be the insertion of stalls until the required information is available in the extended register file. But this solution would make it hard to predetermine the execution time of a program.

Different sources are possible for the ALU. The most common case is the use of the extended register file as input. If the instruction contains an immediate value, it will be used as operand. If the destination for a relativ jump has
5.3 Processor Architecture

Figure 9: The Decode Stage in More Detail

to be calculated, the program counter of the decode stage will be used as operand. If an interrupt service routine is going to be executed, the output of the exception vector table will be used as input for the ALU. After the ALU has completed, it has to be decided which value will be used for write back. Usually the result of the ALU will be used. But 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 of the subsequent instruction after the subroutine call. If a interrupt service routine is going to be executed the program counter of the decode stage has to be used. Because it is the address of the instruction which has been replaced by the call of the interrupt service routine.

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. After a read access the output of all extension modules is merged by a disjunction. This is possible since the output of an extension module is unequal zero only if the module is selected. This solution requires no multiplexer and thus reduces the resource consumption.

The pipe register 3 is build by some control signals, the result of the ALU, the merged output of extension modules and the output of data memory.

Figure 10: The Execute Stage in More Detail
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 its bypass register. The bypass register is used by the forward unit. When the same memory address is read and written at the same cycle, the result of the read operation is undefined. Since the register file is implemented with memory, a bypass register is needed to solve this problem. Figure 11 depicts this stage.

![Write Back Stage](image)

Figure 11: The Write Back Stage in More Detail

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 A.

All instructions are 16 bit in size. The number of instructions increased from 80 to 122, whereof 40 instructions are implemented as conditional instructions.
5.4 Instruction Set Architecture

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, the size of opcode and operands are equal for one instruction format. This simplifies the decoder, but may waste space. Since SPEAR2 uses only 16-bit instructions, space is luxuriously. Therefore we use different count of bits for opcode and immediate values. Only the instruction format using one register and one immediate value is affected by this design decision. The size of an immediate value can be from 4 to 7 bits. Resulting in an opcode from 5 to 8 bits. The remaining 4 bits are used for register operand.

5.4.2 Conditional Instructions

The characteristics of conditional instructions is unchanged. But we changed the way, how conditional instructions are coded by the instruction. Before, 2 bits were necessary 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.

5.5 Implementation

In this section detailed information about the components of the processor will be given. Sometimes differences between SPEAR and its successor will be mentioned.

5.5.1 Program Counter

First the old implementation of the program counter will be explained. It is depicted by figure 12. The old version was improveable, regarding the behaviour at a jump. The address for the instruction memory was taken directly from the program counter. At a jump, the jump destination was first saved to the program counter and after the next cycle the new address was available for the instruction memory. But if the destination address

![Figure 12: The old Implementation of the Program Counter](image-url)
could be used without buffering, the delay of a jump would be reduced by one cycle.

This study was considered when implementing the new program counter, which is depicted by figure 8. Now the destination address is multiplexed directly to the instruction memory and the program counter and pipe register 1 have merged. Since both registers have merged, the logical path between program counter and instruction memory is extended. But without impact to design, because 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.

**Impact of the Customizable Data Path**

The size of 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

Spea 2 possess two memories for instructions. The boot memory which is read only and used at start up. The instruction memory which can be programmed by the programmer module. 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 replacing of the program code without time-consuming recompile of the whole processor. This feature makes it possible to reduce the time required for software development and testing. If this flexibility is not needed, the instruction memory and programmer module can be turned off.

Since all instructions are 16 bit in size, the data size of these memories is 16 bit. Hence the program counter has to be incremented by one. The configuration file can be used to set the size of the memories and if the instruction memory and programmer module are included for the synthesis or are going be removed.
Impact of the Customizable Data Path
The customizable data path has no impact on this component.

5.5.3 Decoder
The job of the decoder is the decoding of instructions and setting control signals for the processor. The decoder was split into two parts:

- One part of the decoder is small and sets control signals without knowing which instruction is decoded.
- 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. SPEAR required to determine the type of decoded instruction to set the control signals for conditional instructions accordingly. This changed with the redesign of the instruction set architecture. The coding of conditional attributes is explained in detail in section 5.4.

The control signals which depend on the instruction type are 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 other control signals are set to predefined values dependent on the occurrence of an exception, insertion of a NOP or in case of normal operation to values conforming to the instruction. More information about setting this control signals is given in section 5.3.2. Examples for such signals are the write enable signals for the register file and data memory. By dividing this control signals into two groups, the resource usage can be reduced since fewer multiplexer are required and in turn affects the feasible clock rate of the design positive.

Impact of the Customizable Data Path
The size of the generated immediate value depends on the configuration. Three instructions used for memory access are only available for the 32 bit
version. If one of the three instructions is used with the 16 bit configuration, an interrupt will be triggered.

### 5.5.4 Register File

The number of registers changed with SPEAR2. The register file now holds 16 registers whereof 14 registers are for general purpose. This change was necessary to make the customizable data path possible. More information about the reason and effects is given in section 5.4.

The number of special function registers has reduced to, since with 6 special function registers only 10 registers would be left for general purpose. The frame pointer registers are now located in the system control module. The registers used to save the return address in case of a subroutine call are reduced to one register. Anyhow, if at all the second register was only used by programs written in assembler, because the compiler was not able to use it. Now only two special function registers are left:

- **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.

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 wich is supported by many FPGA technologies.

**Impact of the Customizable Data Path**

Since the size of the registers depends on the configuration, possible data sizes of the memories are 16 and 32 bit.
5.5 Implementation

5.5.5 Forwarding Unit

The forwarding unit was split into two parts. First the control signals are generated. This part 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 is located on the critical path, time can be saved if the control signals are set in advance and saved by the pipe register 2.

Several sources are possible to serve as input for the forwarding unit:

- **Register File**: To use the output of the register file is the default case. No forwarding is required.

- **Pipe Register 2**: Sometimes it is necessary to use the value stored by the pipe register 2. Namely if the previous instruction uses the same register as destination register.

- **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 uses the same register 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 reduced number of ALU operation codes since only one operation code is needed for two versions of the same operation. Such
5.5 Implementation

46

operations are add and compare which use either two registers or one register and an immediate value as operands.

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

5.5.6 ALU

The control signals for the ALU are generated by the decoder. The processor flags which are manipulated by the ALU are located in the system control module. Namely the zero, overflow, negativ, carry and condition flag. The inputs of the ALU are connected to the two outputs of the forward unit and output of the exception vector table.

First the inputs are preprocessed. Separate control signals determine if an input has to be negated. Also the carry flag is preprocessed. There are four possibilities for the carry flag. Fix the value to zero or one, invert the flag or left it unchanged. Thereby no separate operation code for add and add with carry, sub and sub with carry and some other instructions are needed. The preprocessing enables a reduced number of operation codes for the ALU which minimize the delay of the ALU and additional reduces the resource usage. SPEAR used 32 different operation codes for the ALU. This number was reduced to 22, 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. Thereby the instruction set is more powerful. The barrel shifter is implemented as a sequence of multiplexers. For example an 8 bit barrel shifter is depicted by figure 13. 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 16 and 32 bit barrel shifter. Always two barrel shifter are required, for shift left and another for shift right. The 16 bit has 4 multiplexer stages and the 32 bit barrel shifter 5 stages and much higher resource usage. The number of required multiplexer can be calculated by $n \times \log_2(n)$, where $n$ is the number of bits. Hence the 16 bit implementation requires 64 multiplexer. But the 32 bit barrel shifter requires 160 multiplexer which is more as twice as many.

**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. Either two 16 bit or two 32 bit barrel shifter are used, depending on the configuration. The size and delay of the ALU is affected in great extent.

### 5.5.7 Frame Pointer

The frame pointer used by SPEAR had limited capabilities. Hence a complete redesign was required. First an overview what changed with the new
5.5 Implementation

implemenatation:

- Before the frame pointer registers were part of the extended register file. Now they are located at the system control module.
- The number of independent frame pointer increased by one. Now four registers are available for frame management.
- The size of frames was enlarged.
- Memory access by use of frame pointer and concurrent manipulation of the frame pointer register is now possible with a single instruction. Now it is possible to use the push and pop operation normally used by stack pointer.

There was several reasons for moving the frame pointer registers away from the extended register file. First of all because after halving the number of available registers the number of special function registers had to be reduced too. Another problem with the register file would have be the required additional write port when incrementing or decrementing the frame pointer automatic. Then the register file would have required four access ports. With the additional pipeline stage the forwarding from the data path to the frame pointer registers would have got more complex, and the frame pointer calculation was already a time critical problem for SPEAR. The reasons to choose the system control module as new location for the frame pointer registers are simple. It has to be possible to access the frame pointer registers and the interface of the system control module had enough space left. After the completion of SPEAR2 it turned out the decision about the relocation of the frame pointer registers was right, because the critical path was shorted and the performance of SPEAR2 increased.

The reason to extend the number of available frame pointer by one was quite simple. Enough space of the instruction set and system control module. Enough space of the instruction set is also the reason why the size of frames was enlarged.

The absence of auto increment and decrement was a substantial disadvan-
tage of SPEAR. With the reduced number of registers it was even more
important to implement this feature, since the need of swapping registers will be more probable. After it got clear 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. But 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 also the value for increment and 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 doubled. The reason is the changed memory organisation which is described in the next section.

The result of our changes introduced with SPEAR2 is a flexible and powerful frame pointer architecture.

Impact of the Customizable Data Path
The size of the frame pointer is controlled by the configuration of the customizable data path. As mentioned above, also 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 5.6.1.

5.5.8 Data Memory
In contradiction to SPEAR, which used asynchronous memory, now synchronous memory is used to implement the data memory. Because current FPGA platforms only provide synchronous memory if more than several hundred bits are required. This design decision was one reason for an additional pipeline stage.
Next the changed memory organisation is explained and afterwards how the memory is implemented.
5.5 Implementation

Memory Organisation

As mentioned in paragraph 3.1.2, SPEAR used word addressed memory and only 16 bit access was supported. 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. For one thing it was hard to port the GNU C Compiler to SPEAR, since the compiler was designed for byte addressed data memory. And for another thing memory was wasted if 8 bit values had to be stored.

The new memory organisation is depicted by figure 14. SPEAR2 uses byte addressing for its data memory. As shown by the figure about data memory organisation, every byte has its own address and so it is possible to address any byte directly. The size 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.
5.5 Implementation

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 by figure 15. To connect the data memory several signal 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 splitted. The address lines are used by all memories but without the two least significant bits. Because the information carried by these bits is equal to the byte enable vector. This information is transformed by the memory access unit wich is described in the next section.

To build the data memory by four separate memories enables a target technology independent implementation.

Impact of the Customizable Data Path
The data memory is not affected by the configuration. At an early develop-
ment stage two different implementations were used. One with two memories in parallel and another similar to the current implementation. 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 is described in the next section.

5.5.9 Memory Access

To enable a configuration independent memory interface a separate memory access unit was designed. Thereby the size 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 the two bits, for a write access there are four possible cases to align a byte:

- Both bits are zero. Now 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. Only 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. If a value smaller than the data path size has to be read,
the used instruction determines if the value has to be sign extended or filled up with zeros.
Without the memory access unit the data memory and every extension module would be responsible for alignment and sign extension. Which would result in higher resource usage and more complex extension modules.

**Impact of the Customizable Data Path**
The impact is small: The 32 bit access will not be used by the 16 bit data path.

### 5.5.10 Exceptions

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. There are special instructions to build and manipulate the exception vector table. The table contains 32 entries. The trap vectors are stored at positive and interrupt vectors are stored at negative positions. The table is identical to the one depicted by figure 2 in section 3.2.

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 other 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.

SPEAR2 introduces a new sleep mode which is described in the next section. If the processor is in sleep mode, the interrupt controller is still active. The response time of an interrupt is not affected by the running mode of the processor.

**Impact of the Customizable Data Path**

Only the exception vector table is affected by the configuration of the customizable data path.

### 5.5.11 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, located at the system control module, has to be set. After the bit is set, all flip-flops of the processor - beside the flip-flops of the interrupt handlers - are stopped.

The sleep signal can be accessed through the processor interface, thus the signal can be used by extension modules. Another helpful usage of the led out sleep signal is for indication of the processor utilization. The signal can be connected to an accessible pin of the FPGA, and an oscilloscope or LED can be used to determine the processor utilization.

**Impact of the Customizable Data Path**

The sleep mode is completely unaffected by the customizable data path.
5.5.12 Implementation Details

This section gives some information about the development and describes some implementation details of SPEAR2.

The order 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 by 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 additional attributes were taken into account:

- To be cost effective, economical resource usage is required.
- An important attribute of every processor is the 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.
- Split multiplexer structures into control and execution logic.

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 a NOP or call of an exception service routine is executed instead of the fetched instruction. To set the register destination address to a defined value for example has no impact but would require more resources. This technique was mostly used for implementation of 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 delay and the time delay decreases.

### 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. Therefore the access to extension modules work the same for all of them. The interfaces of the extension modules are mapped to a unique address at the data memory. Hence only load and store instructions are used for access to the interface. The size of the interface is 32 byte. Using bytes for interface description enables the same interface independent of the data path size. It is possible to access single bytes or any other supported data type.

The first two bytes are used to indicate the module status and are read only. The second two bytes are used for configuration. The first status and config byte are defined and the same for all extension modules. 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 config register. Both registers have a size of one byte. First the generic status byte will be explained, which is depicted by figure 16.

The generic status register holds the following status bits:

- **INT (Interrupt):** If the extension module uses interrupts, this bit is set automatically if the module triggers an interrupt. This feature can be used to poll the interrupt of the extension module instead of using an interrupt line.
5.6 Extension Modules

Figure 16: Generic Status Byte

- 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 ready for new tasks.
- FSS (Fail Safe State): FSS is used to indicate if the extension module is in fail safe state. The extension module can switch to this state automatically or manually by software.
- LOOR (Loop Ready): In interaction with LOOW it can be used to determine the presence of an extension module. The bit written to LOOW appears at LOOR after one cycle.

The generic config byte is depicted by figure 17.

Figure 17: Generic Config Byte

The generic config register holds the following configuration bits:

- INTA (Interrupt Acknowledge): Can be used to acknowledge the interrupt of an extension module.
- 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 its interface 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, which can be activated with this configuration bit. For example if the extension module is responsible for controlling a motor. And in case of switching to fail safe state the motor has to be stopped.

• 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 provides functionality of the processor like a programmer.

• Function-Extension Module, which can be used to implement functions in hardware which otherwise have to be emulated by software.

• IO-Extension Module, is 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
Altough the system control module acts as an extension module, it can not be omitted. Since it contains the processor status register, the interrupt
5.6 Extension Modules

handler, and four registers used as frame pointer register, this extension module is an integral part of the processor. The processor status register, the interrupt handler, and the frame pointer registers have been moved to an extension module to enable memory mapping of these important registers. The interface of the system control module is depicted by figure 18.

![Figure 18: Interface of the System Control Module](image)

**Processor Status**

The customized status byte holds the processor status register, and is manipulated by the processor. If the processor status has to be saved because of an exception or trap, the processor status register is copied to the processor config register. The processor status register is automatically restored by returning from exception. The customized status and config byte are identical, but only the config byte can be written. Thus the only way to manipulate the processor status register is to manipulate the processor config register and using the return from exception instruction to write to the processor status register. The processor status register is depicted by figure 19. Beside the processor status flags also known by other processor architectures, the processor status register holds some SPEAR2 specific flags. One bit of the processor status register is not used.
The processor status register holds the following flags:

- **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 an 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 be manipulated by special instructions.

- **SLEEP**: The sleep flag is used to activate the sleep mode of the processor.

- **GIE**: The global interrupt enable flag is used for interrupt controlling. Interrupts can be disabled when clearing this flag.

**Interrupt Registers**

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

- **Interrupt Protocol Register**: This register is used to protocol the occurred 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. The size of the registers can be 16 or 32 bit, depending on the configuration of the data path. The frame pointer concept is described in section 5.5.7.

### 5.6.2 Programmer Module

The programmer module is used for downloading the program code to the instruction memory. In addition an extension module for communication is needed. Dividing this task to 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 by figure 18.

![Programmer Module Interface](image)

**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 size 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 config register, which is depicted by figure 21.

```
Bit 7  Bit 6  Bit 5  Bit 4  Bit 3  Bit 2  Bit 1  Bit 0
PREXE - - - - - - CLR SRC
```

Figure 21: Customized Config Byte of the System Control Module

The customized config 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 instruction memory will be used as instruction source.

• 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. Since the most instructions are stored at successive addresses, the address register is automatically incremented by one.

### 5.7 Differences: 16 vs. 32 bit Version

This paragraph will summarise 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 Differences: 16 vs. 32 bit Version

5.7.1 Interface

The interface to extension modules and memory is not affected by the size 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 size of data path has small impact to the instruction set. Practically the same instruction set is used for both configurations. But only a subset is used, if the data path size is 16 bit. Affected instructions are used for memory access. Following instructions are only available if the extended data path is used:

- Load word - Loads 32 bit from memory
- Store word - Strores 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 size 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 register size is 16 bit and 5 bits are required for 32-bit registers.

Detailed information about individual instructions is given in appendix A.

5.7.3 Addressable Memory

The size of data path affects also the amount of addressable memory. 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 64KB of program or data memory are required, the extended data path has to be used. With extended data path, SPEAR2 can address 4GB of memory.
6 Specification

This chapter provides information about handling of SPEAR2. In particular how SPEAR2 can be configured and the interface is specified in detail.

6.1 Configuration

This section provides information about the configuration of SPEAR2. The configuration is stored in a text 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 size 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 size of instruction memory. More precisely, specifies the number of address lines used for instruction memory. Hence if 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** - Specifies if the boot memory is used with or without an additional instruction RAM. If false, only boot memory is available which is read only. Since no instruction RAM is available, the programmer module is discarded. If true, in addition to the boot memory the instruction RAM is available and consequently the programmer module too.
Table 2 shows the available options and their allowed values.

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Type</th>
<th>Allowed Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>TECH_C</td>
<td>String</td>
<td>NO_TARGET, XILINX, ALTERA</td>
</tr>
<tr>
<td>WORD_CFG_C</td>
<td>Integer</td>
<td>1 → 16-bit data path, 2 → 32-bit data path</td>
</tr>
<tr>
<td>INSTR_RAM_CFG_C</td>
<td>Integer</td>
<td>5 to 16 (independent of data path size), 17 to 32 (only with extended data path)</td>
</tr>
<tr>
<td>DATA_RAM_CFG_C</td>
<td>Integer</td>
<td>5 to 16 (independent of data path size), 17 to 32 (only with extended data path)</td>
</tr>
<tr>
<td>USE_IRAM_CFG_C</td>
<td>Boolean</td>
<td>True → Instruction RAM available, False → No Instruction RAM available</td>
</tr>
</tbody>
</table>

Table 2: Configuration Options of SPEAR2

6.2 Interface
7 Results

Hauptschlich 16 und 32 Bit Version vergleichen
## Conclusion

<table>
<thead>
<tr>
<th></th>
<th>Cyclone II</th>
<th>Stratix II</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>16 bit</strong></td>
<td>1300 CFs/ 370 FFs</td>
<td>1100 ALUTs/ 370 FFs</td>
</tr>
<tr>
<td></td>
<td>65 MHz</td>
<td>110 MHz</td>
</tr>
<tr>
<td><strong>32 bit</strong></td>
<td>2300 CFs/ 650 FFs</td>
<td>1950 ALUTs/ 650 FFs</td>
</tr>
<tr>
<td></td>
<td>55 MHz</td>
<td>90 MHz</td>
</tr>
</tbody>
</table>
## A Instruction Set Reference

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

### A.1 Overview

This section provides an overview of available instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ldlr</td>
<td>rX, IMM8</td>
<td>Load low byte immediate</td>
</tr>
<tr>
<td>2. ldhhi</td>
<td>rX, IMM8</td>
<td>Load high byte immediate</td>
</tr>
<tr>
<td>3. ldlu</td>
<td>rX, IMM8</td>
<td>Load low byte immediate without sign extension</td>
</tr>
<tr>
<td>4. ldflw</td>
<td>rX, IMM6</td>
<td>MEM(FPTRW + IMM6) → rX</td>
</tr>
<tr>
<td>5. ldflx</td>
<td>rX, IMM6</td>
<td>MEM(FPTRX + IMM6) → rX</td>
</tr>
<tr>
<td>6. ldflpy</td>
<td>rX, IMM6</td>
<td>MEM(FPTRY + IMM6) → rX</td>
</tr>
<tr>
<td>7. ldflpz</td>
<td>rX, IMM6</td>
<td>MEM(FPTRZ + IMM6) → rX</td>
</tr>
<tr>
<td>8. stdlw</td>
<td>rX, IMM6</td>
<td>rX → MEM(FPTRW + IMM6)</td>
</tr>
<tr>
<td>9. stdlx</td>
<td>rX, IMM6</td>
<td>rX → MEM(FPTRX + IMM6)</td>
</tr>
<tr>
<td>10. stdlpy</td>
<td>rX, IMM6</td>
<td>rX → MEM(FPTRY + IMM6)</td>
</tr>
<tr>
<td>11. stdlpz</td>
<td>rX, IMM6</td>
<td>rX → MEM(FPTRZ + IMM6)</td>
</tr>
<tr>
<td>12. ldflw+</td>
<td>rX, IMM5</td>
<td>MEM(FPTRW + IMM5) → rX, FPTRW ++</td>
</tr>
<tr>
<td>13. ldflx+</td>
<td>rX, IMM5</td>
<td>MEM(FPTRX + IMM5) → rX, FPTRX ++</td>
</tr>
<tr>
<td>14. ldflpy+</td>
<td>rX, IMM5</td>
<td>MEM(FPTRY + IMM5) → rX, FPTRY ++</td>
</tr>
<tr>
<td>15. ldflpz+</td>
<td>rX, IMM5</td>
<td>MEM(FPTRZ + IMM5) → rX, FPTRZ ++</td>
</tr>
<tr>
<td>16. stdlw+</td>
<td>rX, IMM5</td>
<td>rX → MEM(FPTRW + IMM5), FPTRW ++</td>
</tr>
<tr>
<td>17. stdlx+</td>
<td>rX, IMM5</td>
<td>rX → MEM(FPTRX + IMM5), FPTRX ++</td>
</tr>
<tr>
<td>18. stdlpy+</td>
<td>rX, IMM5</td>
<td>rX → MEM(FPTRY + IMM5), FPTRY ++</td>
</tr>
<tr>
<td>19. stdlpz+</td>
<td>rX, IMM5</td>
<td>rX → MEM(FPTRZ + IMM5), FPTRZ ++</td>
</tr>
<tr>
<td>20. ldflw-</td>
<td>rX, IMM5</td>
<td>MEM(FPTRW + IMM5) → rX, FPTRW --</td>
</tr>
<tr>
<td>21. ldflx-</td>
<td>rX, IMM5</td>
<td>MEM(FPTRX + IMM5) → rX, FPTRX --</td>
</tr>
<tr>
<td>22. ldflpy-</td>
<td>rX, IMM5</td>
<td>MEM(FPTRY + IMM5) → rX, FPTRY --</td>
</tr>
<tr>
<td>23. ldflpz-</td>
<td>rX, IMM5</td>
<td>MEM(FPTRZ + IMM5) → rX, FPTRZ --</td>
</tr>
</tbody>
</table>
### A.1 Overview

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24. stfpw_dec</td>
<td>rX, IMM5</td>
<td>rX → MEM(FPTRW + IMM5), FPTRW --</td>
</tr>
<tr>
<td>25. stfpx_dec</td>
<td>rX, IMM5</td>
<td>rX → MEM(FPTRX + IMM5), FPTRX --</td>
</tr>
<tr>
<td>26. stfpy_dec</td>
<td>rX, IMM5</td>
<td>rX → MEM(FPTRY + IMM5), FPTRY --</td>
</tr>
<tr>
<td>27. stfpz_dec</td>
<td>rX, IMM5</td>
<td>rX → MEM(FPTRZ + IMM5), FPTRZ --</td>
</tr>
<tr>
<td>28. cmpi_eq</td>
<td>rX, IMM7</td>
<td>Compare immediate equal</td>
</tr>
<tr>
<td>29. cmp_eq</td>
<td>rX, rY</td>
<td>Compare equal</td>
</tr>
<tr>
<td>30. cmpi_lt</td>
<td>rX, IMM7</td>
<td>Compare immediate less than</td>
</tr>
<tr>
<td>31. cmp_lt</td>
<td>rX, rY</td>
<td>Compare less than</td>
</tr>
<tr>
<td>32. cmpi_gt</td>
<td>rX, rY</td>
<td>Compare immediate greater than</td>
</tr>
<tr>
<td>33. cmp_gt</td>
<td>rX, rY</td>
<td>Compare greater than</td>
</tr>
<tr>
<td>34. cmpun_lt</td>
<td>rX, rY</td>
<td>Compare unsigned less than</td>
</tr>
<tr>
<td>35. cmpun_gt</td>
<td>rX, rY</td>
<td>Compare unsigned greater than</td>
</tr>
<tr>
<td>36. btest</td>
<td>rX, IMM5</td>
<td>Bit test</td>
</tr>
<tr>
<td>37. bset</td>
<td>rX, IMM5</td>
<td>Bit set</td>
</tr>
<tr>
<td>38. bset_ct</td>
<td>rX, IMM5</td>
<td>Bit set if cond-flag true</td>
</tr>
<tr>
<td>39. bset_cf</td>
<td>rX, IMM5</td>
<td>Bit set if cond-flag false</td>
</tr>
<tr>
<td>40. bclr</td>
<td>rX, IMM5</td>
<td>Bit clear</td>
</tr>
<tr>
<td>41. bclr_ct</td>
<td>rX, IMM5</td>
<td>Bit clear if cond-flag true</td>
</tr>
<tr>
<td>42. bclr_cf</td>
<td>rX, IMM5</td>
<td>Bit clear if cond-flag false</td>
</tr>
<tr>
<td>43. sl</td>
<td>rX, rY</td>
<td>Shift left</td>
</tr>
<tr>
<td>44. sl_ct</td>
<td>rX, rY</td>
<td>Shift left if cond-flag true</td>
</tr>
<tr>
<td>45. sl_cf</td>
<td>rX, rY</td>
<td>Shift left if cond-flag false</td>
</tr>
<tr>
<td>46. sli</td>
<td>rX, IMM4</td>
<td>Shift left immediate</td>
</tr>
<tr>
<td>47. sli_ct</td>
<td>rX, IMM4</td>
<td>Shift left immediate if cond-flag true</td>
</tr>
<tr>
<td>48. sli_cf</td>
<td>rX, IMM4</td>
<td>Shift left immediate if cond-flag false</td>
</tr>
<tr>
<td>49. sr</td>
<td>rX, rY</td>
<td>Shift right</td>
</tr>
<tr>
<td>50. sr_ct</td>
<td>rX, rY</td>
<td>Shift right if cond-flag true</td>
</tr>
<tr>
<td>51. sr Cf</td>
<td>rX, rY</td>
<td>Shift right if cond-flag false</td>
</tr>
<tr>
<td>52. sri</td>
<td>rX, IMM4</td>
<td>Shift right immediate</td>
</tr>
<tr>
<td>53. sri_ct</td>
<td>rX, IMM4</td>
<td>Shift right immediate if cond-flag true</td>
</tr>
<tr>
<td>54. sri_cf</td>
<td>rX, IMM4</td>
<td>Shift right immediate if cond-flag false</td>
</tr>
</tbody>
</table>
### A.1 Overview

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>55. sra</td>
<td>rX, rY</td>
<td>Shift right arithmetic</td>
</tr>
<tr>
<td>56. sra_ct</td>
<td>rX, rY</td>
<td>Shift right arithmetic if cond-flag true</td>
</tr>
<tr>
<td>57. sra_cf</td>
<td>rX, rY</td>
<td>Shift right arithmetic if cond-flag false</td>
</tr>
<tr>
<td>58. srai</td>
<td>rX, IMM4</td>
<td>Shift right arithmetic immediate</td>
</tr>
<tr>
<td>59. srai_ct</td>
<td>rX, IMM4</td>
<td>Shift right arithmetic immediate if cond-flag true</td>
</tr>
<tr>
<td>60. srai_cf</td>
<td>rX, IMM4</td>
<td>Shift right arithmetic immediate if cond-flag false</td>
</tr>
<tr>
<td>61. rrc</td>
<td>rX</td>
<td>Rotate right with carry</td>
</tr>
<tr>
<td>62. rrc_ct</td>
<td>rX</td>
<td>Rotate right with carry if cond-flag true</td>
</tr>
<tr>
<td>63. rrc_cf</td>
<td>rX</td>
<td>Rotate right with carry if cond-flag false</td>
</tr>
<tr>
<td>64. mov</td>
<td>rX, rY</td>
<td>rY → rX</td>
</tr>
<tr>
<td>65. mov_ct</td>
<td>rX, rY</td>
<td>rY → rX if cond-flag true</td>
</tr>
<tr>
<td>66. mov_cf</td>
<td>rX, rY</td>
<td>rY → rX if cond-flag false</td>
</tr>
<tr>
<td>67. addi</td>
<td>rX, IMM6</td>
<td>rX + IMM6 → rX</td>
</tr>
<tr>
<td>68. addi_ct</td>
<td>rX, IMM6</td>
<td>rX + IMM6 → rX if cond-flag true</td>
</tr>
<tr>
<td>69. addi_cf</td>
<td>rX, IMM6</td>
<td>rX + IMM6 → rX if cond-flag false</td>
</tr>
<tr>
<td>70. add</td>
<td>rX, rY</td>
<td>rX + rY → rX</td>
</tr>
<tr>
<td>71. add_ct</td>
<td>rX, rY</td>
<td>rX + rY → rX if cond-flag true</td>
</tr>
<tr>
<td>72. add_cf</td>
<td>rX, rY</td>
<td>rX + rY → rX if cond-flag false</td>
</tr>
<tr>
<td>73. addc</td>
<td>rX, rY</td>
<td>rX + rY + Carry → rX</td>
</tr>
<tr>
<td>74. addc_ct</td>
<td>rX, rY</td>
<td>rX + rY + Carry → rX if cond-flag true</td>
</tr>
<tr>
<td>75. addc_cf</td>
<td>rX, rY</td>
<td>rX + rY + Carry → rX if cond-flag false</td>
</tr>
<tr>
<td>76. sub</td>
<td>rX, rY</td>
<td>rX - rY → rX</td>
</tr>
<tr>
<td>77. sub_ct</td>
<td>rX, rY</td>
<td>rX - rY → rX if cond-flag true</td>
</tr>
<tr>
<td>78. sub_cf</td>
<td>rX, rY</td>
<td>rX - rY → rX if cond-flag false</td>
</tr>
<tr>
<td>79. subc</td>
<td>rX, rY</td>
<td>rX - rY - Carry → rX</td>
</tr>
<tr>
<td>80. subc_ct</td>
<td>rX, rY</td>
<td>rX - rY - Carry → rX if cond-flag true</td>
</tr>
<tr>
<td>81. subc_cf</td>
<td>rX, rY</td>
<td>rX - rY - Carry → rX if cond-flag false</td>
</tr>
<tr>
<td>82. and</td>
<td>rX, rY</td>
<td>rX and rY → rX</td>
</tr>
<tr>
<td>83. and_ct</td>
<td>rX, rY</td>
<td>rX and rY → rX if cond-flag true</td>
</tr>
<tr>
<td>84. and_cf</td>
<td>rX, rY</td>
<td>rX and rY → rX if cond-flag false</td>
</tr>
<tr>
<td>85. or</td>
<td>rX, rY</td>
<td>rX or rY → rX</td>
</tr>
<tr>
<td>Instruction</td>
<td>Operands</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>86. or_ct</td>
<td>rX, rY</td>
<td>rX or rY → rX if cond-flag true</td>
</tr>
<tr>
<td>87. or_cf</td>
<td>rX, rY</td>
<td>rX or rY → rX if cond-flag false</td>
</tr>
<tr>
<td>88. eor</td>
<td>rX, rY</td>
<td>rX xor rY → rX</td>
</tr>
<tr>
<td>89. eor_ct</td>
<td>rX, rY</td>
<td>rX xor rY → rX if cond-flag true</td>
</tr>
<tr>
<td>90. eor_cf</td>
<td>rX, rY</td>
<td>rX xor rY → rX if cond-flag false</td>
</tr>
<tr>
<td>91. not</td>
<td>rX</td>
<td>rX → ¬rX</td>
</tr>
<tr>
<td>92. not_ct</td>
<td>rX</td>
<td>rX → ¬rX if cond-flag true</td>
</tr>
<tr>
<td>93. not_cf</td>
<td>rX</td>
<td>rX → ¬rX if cond-flag false</td>
</tr>
<tr>
<td>94. neg</td>
<td>rX</td>
<td>rX → ¬rX</td>
</tr>
<tr>
<td>95. neg_ct</td>
<td>rX</td>
<td>rX → ¬rX if cond-flag true</td>
</tr>
<tr>
<td>96. neg_cf</td>
<td>rX</td>
<td>rX → ¬rX if cond-flag false</td>
</tr>
<tr>
<td>97. trap</td>
<td>IMM4</td>
<td>Call trap</td>
</tr>
<tr>
<td>98. trap_ct</td>
<td>IMM4</td>
<td>Call trap if cond-flag true</td>
</tr>
<tr>
<td>99. trap_cf</td>
<td>IMM4</td>
<td>Call trap if cond-flag false</td>
</tr>
<tr>
<td>100. jsr</td>
<td>rX</td>
<td>Jump to subroutine</td>
</tr>
<tr>
<td>101. jsr_ct</td>
<td>rX</td>
<td>Jump to subroutine if cond-flag true</td>
</tr>
<tr>
<td>102. jsr_cf</td>
<td>rX</td>
<td>Jump to subroutine if cond-flag false</td>
</tr>
<tr>
<td>103. jmp</td>
<td>a10</td>
<td>Jump immediate</td>
</tr>
<tr>
<td>104. jmp_ct</td>
<td>a10</td>
<td>Jump immediate if cond-flag true</td>
</tr>
<tr>
<td>105. jmp_cf</td>
<td>a10</td>
<td>Jump immediate if cond-flag false</td>
</tr>
<tr>
<td>106. jmp</td>
<td>rX</td>
<td>Jump</td>
</tr>
<tr>
<td>107. jmp_ct</td>
<td>rX</td>
<td>Jump if cond-flag true</td>
</tr>
<tr>
<td>108. jmp_cf</td>
<td>rX</td>
<td>Jump if cond-flag false</td>
</tr>
<tr>
<td>109. ldw</td>
<td>rX, rY</td>
<td>Load word</td>
</tr>
<tr>
<td>110. ldh</td>
<td>rX, rY</td>
<td>Load half word</td>
</tr>
<tr>
<td>111. ldhu</td>
<td>rX, rY</td>
<td>Load half word unsigned</td>
</tr>
<tr>
<td>112. ldb</td>
<td>rX, rY</td>
<td>Load byte</td>
</tr>
<tr>
<td>113. ldbu</td>
<td>rX, rY</td>
<td>Load byte unsigned</td>
</tr>
<tr>
<td>114. stw</td>
<td>rX, rY</td>
<td>Store word</td>
</tr>
<tr>
<td>115. sth</td>
<td>rX, rY</td>
<td>Store half word</td>
</tr>
<tr>
<td>116. stb</td>
<td>rX, rY</td>
<td>store byte</td>
</tr>
</tbody>
</table>
## A.1 Overview

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>117. rts</td>
<td></td>
<td>Return from subroutine</td>
</tr>
<tr>
<td>118. rte</td>
<td></td>
<td>Return from exception</td>
</tr>
<tr>
<td>119. ldvec</td>
<td>rX, IMM5</td>
<td>Load vector</td>
</tr>
<tr>
<td>120. stvec</td>
<td>rX, IMM5</td>
<td>Store vector</td>
</tr>
<tr>
<td>121. nop</td>
<td></td>
<td>No operation</td>
</tr>
<tr>
<td>122. illop</td>
<td></td>
<td>Illegal opcode</td>
</tr>
</tbody>
</table>
A.2 Description

This section provides a detailed reference of the SPEAR2 instruction set. The following notation conventions are used to describe instruction operation:

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

1. **ldli**
   
   **load low byte immediate**

   **Instruction Format**

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM8</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   **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**

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM8</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   **Syntax**  
   ldhi rX, 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.
**ldhi**

`ldhi rX, IMM8`

**Semantics**

16/32-bit

\[
x = x \& 0x0f \\
x = x | (\text{sign\_extend}(\text{IMM8}) \ll 8)
\]

**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.

---

**ldliu**

**load low byte immediate without sign extension**

**Instruction Format**

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM8</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Syntax**

`ldliu rX, IMM8`

**Semantics**

16/32-bit

\[
x = x \& ~(0x0f) \\
x = x | \text{IMM8}
\]

**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.

---

**ldfpw**

**load (half)word with frame pointer W**

**Instruction Format**

| 0 | 1 | 1 | 0 | 0 | n | n | n | n | n | n | r | r | r | r |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Opcode | IMM6 | rX |

**Syntax**

`ldfpw rX, IMM8`

**Semantics**

16/32-bit

\[
x = x \& (0x0f) \\
x = x | \text{sign\_extend}(\text{IMM8}) \ll 8)
\]

**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.
5. **ldfpx**

load (half)word with frame pointer X

**Instruction Format**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM6</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Syntax**  
 ldfpw rX, IMM6

**Semantics**  
rX = MEM(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 size used for memory access is determined by the processor configuration.

6. **ldfpy**

load (half)word with frame pointer Y

**Instruction Format**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM6</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A.2 Description

Syntax  ldfpy rX, IMM6
Semantics  rX = MEM(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 size used for memory access is determined by the processor configuration.

7. ldfpz
load (half)word with frame pointer Z

Instruction Format

<table>
<thead>
<tr>
<th>Opcode</th>
<th>IMM6</th>
<th>rX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 1 1 n n n n n n r r r r</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax  ldfpz rX, IMM6
Semantics  rX = MEM(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 size used for memory access is determined by the processor configuration.

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

Instruction Format

<table>
<thead>
<tr>
<th>Opcode</th>
<th>IMM6</th>
<th>rX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 1 0 0 n n n n n n r r r r</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax  stfpw rX, IMM6
Semantics  rX = MEM(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 size used for memory access is determined by the processor configuration.
A.2 Description

Syntax
stfpw rX, IMM6

Semantics
MEM(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. stfp
store (half)word with frame pointer X

Instruction Format

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM6</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax
stfpw rX, IMM6

Semantics
MEM(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. stfp
store (half)word with frame pointer Y

Instruction Format

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM6</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax
stfpw rX, IMM6

Semantics
MEM(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. stfp
store (half)word with frame pointer Z

Instruction Format

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM6</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax
stfpz rX, IMM6

Semantics
MEM(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.
A.2 Description

Instruction Format

<table>
<thead>
<tr>
<th>Opcode</th>
<th>IMM6</th>
<th>rX</th>
</tr>
</thead>
</table>

Syntax    stfpz rX, IMM6  
Semantics MEM(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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>IMM5</th>
<th>rX</th>
</tr>
</thead>
</table>

Syntax    ldfpw_inc rX, IMM5  
Semantics  
16-bit:  
rX = MEM(FPTRW + IMM5)  
FPTRW = FPTRW + 2  
32-bit:  
rX = MEM(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 size 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 1 0 n n n n n r r r r
```

*Syntax*  
`ldfpx_inc rX, IMM5`

*Semantics*  
**16-bit:**

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

**32-bit:**

- `rX = MEM(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 size 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
```

*Syntax*  
`ldfpy_inc rX, IMM5`

*Semantics*  
**16-bit:**

- `rX = MEM(FPTRY + IMM5)`
- `FPTRY = FPTRY + 2`

**32-bit:**

- `rX = MEM(FPTRY + IMM5)`
- `FPTRY = FPTRY + 4`
A.2 Description

**Syntax**
```
ldfpy_inc rX, IMM5
```

**Semantics**

**16-bit:**
- \( rX = \text{MEM}(\text{FPTRY} + \text{IMM5}) \)
- \( \text{FPTRY} = \text{FPTRY} + 2 \)

**32-bit:**
- \( rX = \text{MEM}(\text{FPTRY} + \text{IMM5}) \)
- \( \text{FPTRY} = \text{FPTRY} + 4 \)

**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 size 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**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM5</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Syntax**
```
ldfpz_inc rX, IMM5
```

**Semantics**

**16-bit**
- \( rX = \text{MEM}(\text{FPTRZ} + \text{IMM5}) \)
- \( \text{FPTRZ} = \text{FPTRZ} + 2 \)

**32-bit**
- \( rX = \text{MEM}(\text{FPTRZ} + \text{IMM5}) \)
- \( \text{FPTRZ} = \text{FPTRZ} + 4 \)

**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 size 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 r r r r
```

**Syntax**

`stfpw_inc rX, IMM5`

**Semantics**

**16-bit:**

\[ \text{MEM}(\text{FPTRW} + \text{IMM5}) = rX \]

\[ \text{FPTRW} = \text{FPTRW} + 2 \]

**32-bit:**

\[ \text{MEM}(\text{FPTRW} + \text{IMM5}) = rX \]

\[ \text{FPTRW} = \text{FPTRW} + 4 \]

**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 r r r r
```

**Syntax**

`stfpx_inc rX, IMM5`

**Semantics**

**16-bit:**

\[ \text{MEM}(\text{FPTRX} + \text{IMM5}) = rX \]

\[ \text{FPTRX} = \text{FPTRX} + 2 \]

**32-bit:**

\[ \text{MEM}(\text{FPTRX} + \text{IMM5}) = rX \]

\[ \text{FPTRX} = \text{FPTRX} + 4 \]

**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.
### Syntax

\texttt{stfp\_inc} rX, IMM5

### Semantics

**16-bit:**

\[ \text{MEM}(\text{FPTRX} + \text{IMM5}) = rX \]
\[ \text{FPTRX} = \text{FPTRX} + 2 \]

**32-bit:**

\[ \text{MEM}(\text{FPTRX} + \text{IMM5}) = rX \]
\[ \text{FPTRX} = \text{FPTRX} + 4 \]

### 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. **\texttt{stfpy\_inc}**

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

### Instruction Format

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM5</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Syntax

\texttt{stfpy\_inc} rX, IMM5

### Semantics

**16-bit:**

\[ \text{MEM}(\text{FPTRY} + \text{IMM5}) = rX \]
\[ \text{FPTRY} = \text{FPTRY} + 2 \]

**32-bit:**

\[ \text{MEM}(\text{FPTRY} + \text{IMM5}) = rX \]
\[ \text{FPTRY} = \text{FPTRY} + 4 \]

### 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**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM5</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Syntax**  
```
stfpz_inc rX, IMM5
```

**Semantics**

16-bit:  
\[
\text{MEM}(\text{FPTRZ} + \text{IMM5}) = rX \\
\text{FPTRZ} = \text{FPTRZ} + 2
\]

32-bit:  
\[
\text{MEM}(\text{FPTRZ} + \text{IMM5}) = rX \\
\text{FPTRZ} = \text{FPTRZ} + 4
\]

**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**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM5</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A.2 Description

Syntax

ldfpw_dec rX, IMM5

Semantics

16-bit:

rX = MEM(FPTRW + IMM5)
FPTRW = FPTRW - 2

32-bit:

rX = MEM(FPTRW + IMM5)
FPTRW = FPTRW - 4

Description

Loads a word or halfword from 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 size 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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM5</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Syntax
`ldfpx_dec rX, IMM5`

### Semantics
**16-bit:**
- \( rX = MEM(FPTRX + IMM5) \)
- \( FPTRX = FPTRX - 2 \)

**32-bit:**
- \( rX = MEM(FPTRX + IMM5) \)
- \( FPTRX = FPTRX - 4 \)

### Description
Loads a word or halfword from 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 size 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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM5</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Syntax
```
ldfpy_dec rX, IMM5
```

### Semantics
**16-bit:**
- \( rX = \text{MEM}(\text{FPTRY} + \text{IMM5}) \)
- \( \text{FPTRY} = \text{FPTRY} - 2 \)

**32-bit:**
- \( rX = \text{MEM}(\text{FPTRY} + \text{IMM5}) \)
- \( \text{FPTRY} = \text{FPTRY} - 4 \)

### Description
Loads a word or halfword from 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 size 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 | r | r | r | r |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Opcode | IMM5 | \( rX \) |
A.2 Description

**Syntax**
```
ldfpz_dec rX, IMM5
```

**Semantics**

16-bit:
```
rX = MEM(FPTRZ + IMM5)
FPTRZ = FPTRZ - 2
```

32-bit:
```
rX = MEM(FPTRZ + IMM5)
FPTRZ = FPTRZ - 4
```

**Description**

Loads a word or halfword from 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 size 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**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM5</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### A.2 Description

**Syntax**

stfpw_dec rX, IMM5

**Semantics**

16-bit:

\[ \text{MEM}(\text{FPTRW} + \text{IMM5}) = rX \]
\[ \text{FPTRW} = \text{FPTRW} - 2 \]

32-bit:

\[ \text{MEM}(\text{FPTRW} + \text{IMM5}) = rX \]
\[ \text{FPTRW} = \text{FPTRW} - 4 \]

**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**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opcode</td>
<td>IMM5</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Syntax**

stfpx_dec rX, IMM5

**Semantics**

16-bit:

\[ \text{MEM}(\text{FPTRX} + \text{IMM5}) = rX \]
\[ \text{FPTRX} = \text{FPTRX} - 2 \]

32-bit:

\[ \text{MEM}(\text{FPTRX} + \text{IMM5}) = rX \]
\[ \text{FPTRX} = \text{FPTRX} - 4 \]

**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**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM5</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Syntax**

stfpy_dec rX, IMM5

**Semantics**

16-bit:
MEM(FPTRY + IMM5) = rX
FPTRY = FPTRY - 2
32-bit:
MEM(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**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM5</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A.2 Description

**Syntax**

\[ \text{stfz}_{-\text{dec}} \ rX, \ \text{IMM5} \]

**Semantics**

**16-bit:**

\[ \text{MEM} (\text{FPTRZ} + \text{IMM5}) = rX \]
\[ \text{FPTRZ} = \text{FPTRZ} - 2 \]

**32-bit:**

\[ \text{MEM} (\text{FPTRZ} + \text{IMM5}) = rX \]
\[ \text{FPTRZ} = \text{FPTRZ} - 4 \]

**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 | r | r | r | r |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Opcode | IMM7 | rX |

**Syntax**

\[ \text{cmpl}_{-\text{eq}} \ rX, \ \text{IMM7} \]

**Semantics**

if \( rX - \text{IMM7} == 0 \)

then 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**

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>rY</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**A.2 Description**

**Syntax**

cmp_eq rX, rY

**Semantics**

if (rX == rY)
then 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**

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode IMM7 rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Syntax**

cmpl t rX, IMM7

**Semantics**

if (rX < IMM7)
then 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**

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode rY rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A.2 Description

Syntax \texttt{cmp}_{lt} \ rX, \ rY

Semantics \begin{align*}
\text{if } (rX < rY) \\
\text{then cond-flag} &= 1 \\
\text{else cond-flag} &= 0
\end{align*}

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. \texttt{cmpi}_{gt}

compare greater than immediate

Instruction Format

\[ \begin{array}{ccccccccccc}
0 & 0 & 1 & 1 & 1 & n & n & n & n & n & r & r & r & r \\
\hline
\text{Opcode} & \text{IMM7} & \text{rX}
\end{array} \]

Syntax \texttt{cmpi}_{gt} \ rX, \ IMM7

Semantics \begin{align*}
\text{if } (rX > \text{IMM7}) \\
\text{then cond-flag} &= 1 \\
\text{else cond-flag} &= 0
\end{align*}

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. \texttt{cmp}_{gt}

compare greater than

Instruction Format

\[ \begin{array}{ccccccccccc}
1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & r & r & r & r & r & r \\
\hline
\text{Opcode} & \text{rY} & \text{rX}
\end{array} \]
A.2 Description

Syntax: \texttt{cmp\_gt rX, rY}

Semantics:

\begin{verbatim}
if (rX > rY)
then cond-flag = 1
else cond-flag = 0
\end{verbatim}

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. \texttt{cmpun\_lt}

compare less than unsigned

Instruction Format

\begin{verbatim}

\begin{tabular}{cccccccc}
1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
Opcode & r & r & r & r & r & r & r
\end{tabular}
\end{verbatim}

Syntax: \texttt{cmpun\_lt rX, rY}

Semantics:

\begin{verbatim}
if (rX < rY)
then cond-flag = 1
else cond-flag = 0
\end{verbatim}

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 integers.

35. \texttt{cmpun\_gt}

compare greater than unsigned

Instruction Format

\begin{verbatim}

\begin{tabular}{cccccccc}
1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
Opcode & r & r & r & r & r & r & r
\end{tabular}
\end{verbatim}

Syntax: \texttt{cmpun\_gt rX, rY}

Semantics:

\begin{verbatim}
if (rX < rY)
then cond-flag = 1
else cond-flag = 0
\end{verbatim}

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 integers.
A.2 Description

Syntax
cmpun\_gt rX, rY

Semantics
if (rX > rY)
then 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 integers.

36. btest

bit test

Instruction Format

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM5</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax
btest rX, IMM5

Semantics
16-bit:
if (rX(IMM4) == 1)
then cond\_flag = 1
else cond\_flag = 0

32-bit:
if (rX(IMM5) == 1)
then 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 size of the immediate value depends on the processor configuration.

37. bset

bit set
### Instruction Format

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>IMM5</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Syntax

`bset rX, IMM5`

#### Semantics

**16-bit:**

rX = rX — (1 << IMM4)

**32-bit:**

rX = rX — (1 << IMM5)

#### Description

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

39. **bset_cf**

bit set if cond-flag false
### Instruction Format

<table>
<thead>
<tr>
<th>Opcode</th>
<th>IMM5</th>
<th>rX</th>
</tr>
</thead>
</table>

### Syntax

`bset_cfrX, IMM5`

### Semantics

**16-bit:**

\[
\text{if (cond-flag == 0)} \\
\text{rX = rX — (1 } \ll \text{ IMM4)}
\]

**32-bit:**

\[
\text{if (cond-flag == 0)} \\
\text{rX = rX — (1 } \ll \text{ IMM5)}
\]

### Description

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

---

### 40. bclr

**bit clear**

### Instruction Format

<table>
<thead>
<tr>
<th>Opcode</th>
<th>IMM5</th>
<th>rX</th>
</tr>
</thead>
</table>

### Syntax

`bclr rX, IMM5`

### Semantics

**16-bit:**

\[
rX = rX \& \sim (1 \ll \text{ IMM4})
\]

**32-bit:**

\[
rX = rX \& \sim (1 \ll \text{ IMM5})
\]

### Description

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

---

### 41. bclr_ct

**bit clear if cond-flag true**
A.2 Description

Instruction Format

<table>
<thead>
<tr>
<th>Opcode</th>
<th>IMM5</th>
<th>rX</th>
</tr>
</thead>
</table>

Syntax  
bclr_c\_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 size of the immediate value depends on the processor configuration.

42. bclr\_cf

bit clear if cond\_flag false

Instruction Format

<table>
<thead>
<tr>
<th>Opcode</th>
<th>IMM5</th>
<th>rX</th>
</tr>
</thead>
</table>

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 size of the immediate value depends on the processor configuration.
43. **sl**
shift left

**Instruction Format**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rY</td>
<td>rX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Syntax**

bclr rX, IMM5

**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 r1,r2**

32 Bit:

r1[31] → Carry-Flag
r1[n] → r1[n+r2]
0 → r1[0..r2-1]

16 Bit:

r1[15] → Carry-Flag
r1[n] → r1[n+r2]
0 → r1[0..r2-1]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

**Befehlsformat:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SL</td>
<td>r2 (4 Bit)</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Beschreibung:**

“‘Shift Left’” verschiebt den Inhalt vom Register r1 um eine bestimmte
A.2 Description

Anzahl von Stellen, angegeben in r2, nach links. In der 32 Bit Konfiguration werden 5 Bit und in der 16 Bit Konfiguration werden nur die unteren 4 Bit von Register r2 verwendet. Das MSB von r1 wird im Carry-Flag gespeichert, die unteren Bits werden mit einer 0 aufgefüllt. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

45. **SL_CT r1,r2**

wenn Condition-Flag "‘TRUE’" ist

32 Bit:

\[
\begin{align*}
\text{r1}[31] & \rightarrow \text{Carry-Flag} \\
\text{r1}[n] & \rightarrow \text{r1}[n+r2] \\
0 & \rightarrow \text{r1}[0..r2-1]
\end{align*}
\]

16 Bit:

\[
\begin{align*}
\text{r1}[15] & \rightarrow \text{Carry-Flag} \\
\text{r1}[n] & \rightarrow \text{r1}[n+r2] \\
0 & \rightarrow \text{r1}[0..r2-1]
\end{align*}
\]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

wenn Condition-Flag "‘FALSE’" ist

\[
\begin{align*}
\text{r1} & \rightarrow \text{r1}
\end{align*}
\]

Befehlsformat:

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 0 & 0 & 0 & r \\
r & r & r & r & r & r & 100
\end{array}
\]

Beschreibung:

"‘Shift Left when Cond-flag True’" verhält sich genau so wie der Befehl SL, allerdings nur, wenn das Condition-Flag "‘TRUE’" ist. Wenn nicht, dann bleibt das Register r1 unverändert.

46. **SL_CF r1,r2**

wenn Condition-Flag "‘FALSE’" ist
32 Bit:
   r1[31] → Carry-Flag
   r1[n] → r1[n+r2]
   0 → r1[0..r2-1]

16 Bit:
   r1[15] → Carry-Flag
   r1[n] → r1[n+r2]
   0 → r1[0..r2-1]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

wenn Condition-Flag ”‘TRUE’” ist
   r1 → r1

Befehlsformat:

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL</td>
<td>CF</td>
<td>r2 (4 Bit)</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Beschreibung:
”‘Shift Left when Cond-flag False’” verhilt sich genau so wie der Befehl SL, allerdings nur, wenn das Condition-Flag ”‘FALSE’” ist. Wenn nicht, dann bleibt das Register r1 unverndert.

47. SLI r1,n4

32 Bit:
   r1[31] → Carry-Flag
   r1[n] → r1[n+n4]
   0 → r1[0..n4-1]

16 Bit:
   r1[15] → Carry-Flag
   r1[n] → r1[n+n4]
   0 → r1[0..n4-1]
Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

```
1 0 1 0 0 0 1 n n n r r r r
```

SLI n4 (4 Bit) r1 (4 Bit)

Beschreibung:

"‘Shift Left Immediate’" verschiebt den Inhalt vom Register r1 um eine bestimmte Anzahl von Stellen, angegeben mit n4, nach links. Das MSB von r1 wird im Carry-Flag gespeichert, die unteren Bits werden mit einer 0 aufgefüllt. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

48. SLI_CT r1,n4

wenn Condition-Flag "‘TRUE’" ist

32 Bit:

- r1[31] → Carry-Flag
- r1[n] → r1[n+n4]
- 0 → r1[0..n4-1]

16 Bit:

- r1[15] → Carry-Flag
- r1[n] → r1[n+n4]
- 0 → r1[0..n4-1]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

wenn Condition-Flag "‘FALSE’" ist

r1 → r1

Befehlsformat:

```
1 0 0 1 0 0 0 1 n n n n r r r r
```

SLI_CT n4 (4 Bit) r1 (4 Bit)
Beschreibung:

“Shift Left Immediate when Cond-flag True”’ verhält sich genau so wie
der Befehl SLI, allerdings nur, wenn das Condition-Flag ”‘TRUE’” ist.
Wenn nicht, dann bleibt das Register r1 unverändert.

49. **SLI_CF r1,n4**

wenn Condition-Flag ”‘FALSE’” ist

32 Bit:

\[
\begin{align*}
    r1[31] & \rightarrow \text{Carry-Flag} \\
    r1[n] & \rightarrow r1[n+n4] \\
    0 & \rightarrow r1[0..n4-1]
\end{align*}
\]

16 Bit:

\[
\begin{align*}
    r1[15] & \rightarrow \text{Carry-Flag} \\
    r1[n] & \rightarrow r1[n+n4] \\
    0 & \rightarrow r1[0..n4-1]
\end{align*}
\]

Carry-, Neg-, Overflow- und Zero-Flag werden ma-
nipuliert.

wenn Condition-Flag ”‘TRUE’” ist

\[
    r1 \rightarrow r1
\]

Befehlsformat:

\[
\begin{array}{cccccccc}
    1 & 0 & 0 & 1 & 0 & 0 & 1 & n & n & n & r & r & r & r \\
\end{array}
\]

SLI_CF | n4 (4 Bit) | r1 (4 Bit)

Beschreibung:

”‘Shift Left Immediate when Cond-flag False’”’ verhält sich genau so wie
der Befehl SLI, allerdings nur, wenn das Condition-Flag ”‘FALSE’” ist.
Wenn nicht, dann bleibt das Register r1 unverändert.

50. **SR r1,r2**

\[
\begin{align*}
    r1[0] & \rightarrow \text{Carry-Flag} \\
    r1[n+r2] & \rightarrow r1[n]
\end{align*}
\]
0 \rightarrow r1[31..31-r2+1] 

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

\[
\begin{array}{c|cccc|ccc|ccc|cc}
& 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & r & r & r & r & r \\
\hline
SR & r2 & (4 \text{ Bit}) & r1 & (4 \text{ Bit}) \\
\end{array}
\]

Beschreibung:

"'Shift Right'" verschiebt den Inhalt vom Register r1 um eine bestimmte Anzahl von Stellen, angegeben in r2, nach rechts. In der 32 Bit Konfiguration werden 5 Bit und in der 16 Bit Konfiguration werden nur die unteren 4 Bit von Register r2 verwendet. Das LSB von r1 wird im Carry-Flag gespeichert, die oberen Bits werden mit einer 0 aufgefüllt. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

51. **SR_CT r1,r2**

wenn Condition-Flag "'TRUE'" ist

\[
\begin{align*}
r1[0] & \rightarrow \text{Carry-Flag} \\
r1[n+r2] & \rightarrow r1[n] \\
0 & \rightarrow r1[31..31-r2+1] \\
\end{align*}
\]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

wenn Condition-Flag "'FALSE'" ist

r1 \rightarrow r1

Befehlsformat:

\[
\begin{array}{c|cccc|ccc|ccc|cc}
& 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & r & r & r & r & r \\
\hline
SR_CT & r2 & (4 \text{ Bit}) & r1 & (4 \text{ Bit}) \\
\end{array}
\]

Beschreibung:

"'Shift Right when Cond-flag True'" verhält sich genau so wie der Befehl SR, allerdings nur, wenn das Condition-Flag "'TRUE'" ist. Wenn nicht, dann bleibt das Register r1 unverändert.
52. **SR_CF r1,r2**

   wenn Condition-Flag "'FALSE'" ist
   
   \[
   \begin{align*}
   r1[0] & \rightarrow \text{Carry-Flag} \\
   r1[n+r2] & \rightarrow r1[n] \\
   0 & \rightarrow r1[31..31-r2+1] \\
   \end{align*}
   \]
   
   Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

   wenn Condition-Flag "'TRUE'" ist
   
   \[
   r1 \rightarrow r1
   \]

   Befehlsformat:

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR_CF</td>
<td>r2 (4 Bit)</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   Beschreibung:
   "'Shift Right when Cond-flag False'" verhält sich genau so wie der Befehl SR, allerdings nur, wenn das Condition-Flag "'False'" ist. Wenn nicht, dann bleibt das Register r1 unverändert.

53. **SRI r1,n4**

   \[
   \begin{align*}
   r1[0] & \rightarrow \text{Carry-Flag} \\
   r1[n+n4] & \rightarrow r1[n] \\
   0 & \rightarrow r1[31..31-n4+1] \\
   \end{align*}
   \]
   
   Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

   Befehlsformat:

   | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | n | n | n | r | r | r | r |
   |---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
   | SRI | n4 (4 Bit) | r1 (4 Bit) |

   Beschreibung:
   "'Shift Right Immediate'" verschiebt den Inhalt vom Register r1 um eine bestimmte Anzahl von Stellen, angegeben mit n4, nach rechts. Das
A.2 Description

LSB von r1 wird im Carry-Flag gespeichert, die oberen Bits werden mit einer 0 aufgefüllt. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

54. **SRI_CT r1,n4**

wenn Condition-Flag ’‘TRUE’’ ist

\[ r1[0] \rightarrow \text{Carry-Flag} \]
\[ r1[n+n4] \rightarrow r1[n] \]
\[ 0 \rightarrow r1[31..31-n4+1] \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

wenn Condition-Flag ’‘FALSE’’ ist

\[ r1 \rightarrow r1 \]

Befehlsformat:

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRI_CT</td>
<td>n4 (4 Bit)</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Beschreibung:

’‘Shift Right Immediate when Cond-flag True’’ verhält sich genau so wie der Befehl SRI, allerdings nur, wenn das Condition-Flag ’‘TRUE’’ ist. Wenn nicht, dann bleibt das Register r1 unverändert.

55. **SRI_CF r1,n4**

wenn Condition-Flag ’‘FALSE’’ ist

\[ r1[0] \rightarrow \text{Carry-Flag} \]
\[ r1[n+n4] \rightarrow r1[n] \]
\[ 0 \rightarrow r1[31..31-n4+1] \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

wenn Condition-Flag ’‘TRUE’’ ist

\[ r1 \rightarrow r1 \]
Befehlsformat:

```
1 0 0 0 0 0 1 1 n n n n r r r r
SRI_CT n4 (4 Bit) r1 (4 Bit)
```

Beschreibung:

"'Shift Right Immediate when Cond-flag False'" verhält sich genau so wie der Befehl SRI, allerdings nur, wenn das Condition-Flag "'FALSE'"
ist. Wenn nicht, dann bleibt das Register r1 unverändert.

56. **SRA r1,r2**

32 Bit:

- r1[0] → Carry-Flag
- r1[n+r2] → r1[n]
- r1[31] → r1[31..31-r2+1]

16 Bit:

- r1[0] → Carry-Flag
- r1[n+r2] → r1[n]
- r1[15] → r1[15..15-r2+1]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

```
1 1 1 0 1 0 0 0 r r r r r
SRA r2 (4 Bit) r1 (4 Bit)
```

Beschreibung:

"'Shift Right Arithmetic'" verschiebt den Inhalt vom Register r1 um eine bestimmte Anzahl von Stellen, angegeben in r2, nach rechts. In der 32 Bit Konfiguration werden 5 Bit und in der 16 Bit Konfiguration werden nur die unteren 4 Bit von Register r2 verwendet. Das LSB von r1 wird im Carry-Flag gespeichert, die oberen Bits werden vorzeichenrichtig aufgefüllt. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.
57. **SRA_CT r1,r2**

wenn Condition-Flag ”‘TRUE’” ist

32 Bit:

\[
\begin{align*}
& r1[0] \rightarrow \text{Carry-Flag} \\
& r1[n+r2] \rightarrow r1[n] \\
& r1[31] \rightarrow r1[31..31-r2+1]
\end{align*}
\]

16 Bit:

\[
\begin{align*}
& r1[0] \rightarrow \text{Carry-Flag} \\
& r1[n+r2] \rightarrow r1[n] \\
& r1[15] \rightarrow r1[15..15-r2+1]
\end{align*}
\]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

wenn Condition-Flag ”‘FALSE’” ist

\[r1 \rightarrow r1\]

Befehlsformat:

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 0 & r & r & r & r & r & r \\
\hline
\text{SRA_CT} & r2 (4 Bit) & r1 (4 Bit)
\end{array}
\]

Beschreibung:

”‘Shift Right Arithmetic when Cond-flag True’’ verschiebt den Inhalt vom Register r1 um eine bestimmte Anzahl von Stellen, angegeben in r2, nach rechts. Allerdings nur, wenn das Condition-Flag ”‘TRUE’” ist. In der 32 Bit Konfiguration werden 5 Bit und in der 16 Bit Konfiguration werden nur die unteren 4 Bit von Register r2 verwendet. Das LSB von r1 wird im Carry-Flag gespeichert, die oberen Bits werden vorzeichenrichtig aufgefüllt. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

58. **SRA_CF r1,r2**

wenn Condition-Flag ”‘FALSE’” ist
A.2 Description

32 Bit:
\[ r1[0] \rightarrow \text{Carry-Flag} \]
\[ r1[n+r2] \rightarrow r1[n] \]
\[ r1[31] \rightarrow r1[31..31-r2+1] \]

16 Bit:
\[ r1[0] \rightarrow \text{Carry-Flag} \]
\[ r1[n+r2] \rightarrow r1[n] \]
\[ r1[15] \rightarrow r1[15..15-r2+1] \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

wenn Condition-Flag "‘TRUE’" ist
\[ r1 \rightarrow r1 \]

Befehlsformat:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRA</td>
<td>CF</td>
<td>r2 (4 Bit)</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Beschreibung:

"‘Shift Right Arithmetic when Cond-flag False’" verschiebt den Inhalt vom Register r1 um eine bestimmte Anzahl von Stellen, angegeben in r2, nach rechts. Allerdings nur, wenn das Condition-Flag "‘FALSE’" ist. In der 32 Bit Konfiguration werden 5 Bit und in der 16 Bit Konfiguration werden nur die unteren 4 Bit von Register r2 verwendet. Das LSB von r1 wird im Carry-Flag gespeichert, die oberen Bits werden vorzeichenrichtig aufgefüllt. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

59. SRAI r1,n4

32 Bit:
\[ r1[0] \rightarrow \text{Carry-Flag} \]
\[ r1[n+n4] \rightarrow r1[n] \]
\[ r1[31] \rightarrow r1[31..31-n4+1] \]
16 Bit:

\[
\begin{align*}
& r1[0] \rightarrow \text{Carry-Flag} \\
& r1[n+n4] \rightarrow r1[n] \\
& r1[15] \rightarrow r1[15..15-n4+1]
\end{align*}
\]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 0 & 1 & 0 & 1 & n & n & n & r & r & r & r \\
\hline
\text{SRAI} & n4 (4 Bit) & r1 (4 Bit)
\end{array}
\]

Beschreibung:

"‘Shift Right Arithmetic Immediate’" verschiebt den Inhalt vom Register r1 um eine bestimmte Anzahl von Stellen, angegeben mit n4, nach rechts. Das LSB von r1 wird im Carry-Flag gespeichert, die oberen Bits werden vorzeichenrichtig aufgefüllt. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

60. SRAI_CT r1, n4

wenn Condition-Flag "‘TRUE’" ist

32 Bit:

\[
\begin{align*}
& r1[0] \rightarrow \text{Carry-Flag} \\
& r1[n+n4] \rightarrow r1[n] \\
& r1[31] \rightarrow r1[31..31-n4+1]
\end{align*}
\]

16 Bit:

\[
\begin{align*}
& r1[0] \rightarrow \text{Carry-Flag} \\
& r1[n+n4] \rightarrow r1[n] \\
& r1[15] \rightarrow r1[15..15-n4+1]
\end{align*}
\]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

wenn Condition-Flag "‘FALSE’" ist

\[
\begin{align*}
& r1 \rightarrow r1
\end{align*}
\]
Befehlsformat:

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
\hline
\text{SRAI\_CT} & n4 \ (4 \text{ Bit}) & r1 \ (4 \text{ Bit})
\end{array}
\]

Beschreibung:

"'Shift Right Arithmetic Immedate when Cond-flag True'" verschiebt den Inhalt vom Register r1 um eine bestimmte Anzahl von Stellen, angegeben mit n4, nach rechts. Allerdings nur, wenn das Condition-Flag "'TRUE'" ist. Das LSB von r1 wird im Carry-Flag gespeichert, die oberen Bits werden vorzeichenrichtig aufgefüllt. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

61. \textbf{SRAI\_CF r1,n4}

wenn Condition-Flag "'FALSE'" ist

32 Bit:
\begin{align*}
\text{r1}[0] & \rightarrow \text{Carry-Flag} \\
\text{r1}[n+n4] & \rightarrow \text{r1}[n] \\
\text{r1}[31] & \rightarrow \text{r1}[31..31-n4+1]
\end{align*}

16 Bit:
\begin{align*}
\text{r1}[0] & \rightarrow \text{Carry-Flag} \\
\text{r1}[n+n4] & \rightarrow \text{r1}[n] \\
\text{r1}[15] & \rightarrow \text{r1}[15..15-n4+1]
\end{align*}

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

wenn Condition-Flag "'TRUE'" ist

\begin{align*}
\text{r1} & \rightarrow \text{r1}
\end{align*}
Beschreibung:

"’Shift Right Arithmetic Immedate when Cond-flag False’" verschiebt den Inhalt vom Register r1 um eine bestimmte Anzahl von Stellen, angegeben mit n4, nach rechts. Allerdings nur, wenn das Condition-Flag "’FALSE’" ist. Das LSB von r1 wird im Carry-Flag gespeichert, die oberen Bits werden vorzeichenrichtig aufgefüllt. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

62. **RRC** r1

\[
\begin{align*}
\text{r1}[0] & \rightarrow \text{Temp} \\
\text{r1}[31..1] & \rightarrow \text{r1}[30..0] \\
\text{Carry-Flag} & \rightarrow \text{r1}[31] \\
\text{Temp} & \rightarrow \text{Carry-Flag}
\end{align*}
\]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRC</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Beschreibung:

"’Rotate Right with Carry’" verschiebt den Inhalt vom Register r1 um eine Stelle nach rechts. Das Carry-Flag wird in das MSB geschrieben. Das LSB von r1 wird anschließend im Carry-Flag gespeichert. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

63. **RRC_CT** r1

wenn Condition-Flag "’TRUE’" ist

\[
\begin{align*}
\text{r1}[0] & \rightarrow \text{Temp} \\
\text{r1}[31..1] & \rightarrow \text{r1}[30..0] \\
\text{Carry-Flag} & \rightarrow \text{r1}[31] \\
\text{Temp} & \rightarrow \text{Carry-Flag}
\end{align*}
\]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.
wenn Condition-Flag ”‘FALSE’” ist

\[ r1 \rightarrow r1 \]

Befehlsformat:

\[
\begin{array}{cccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & r & r & r & r \\
\end{array}
\]

Beschreibung:
”‘Rotate Right with Carry when Cond-flag True’” verschiebt den Inhalt vom Register r1 um eine Stelle nach rechts. Das Carry-Flag wird in das MSB geschrieben. Das LSB von r1 wird anschließend im Carry-Flag gespeichert. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt. Ist das Condition-Flag ”‘FALSE’”, so bleibt das Register r1 unverändert.

64. **RRC_CF r1**

wenn Condition-Flag ”‘FALSE’” ist

\[ r1[0] \rightarrow \text{Temp} \]
\[ r1[31..1] \rightarrow r1[30..0] \]
Carry-Flag \[\rightarrow r1[31] \]
Temp \[\rightarrow \text{Carry-Flag} \]
Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

wenn Condition-Flag ”‘TRUE’” ist

\[ r1 \rightarrow r1 \]

Befehlsformat:

\[
\begin{array}{cccccccccccc}
1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & r & r & r & r \\
\end{array}
\]

Beschreibung:
”‘Rotate Right with Carry when Cond-flag False’” verschiebt den Inhalt vom Register r1 um eine Stelle nach rechts. Das Carry-Flag wird in
das MSB geschrieben. Das LSB von r1 wird anschließend im Carry-Flag gespeichert. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt. Ist das Condition-Flag ’TRUE’, so bleibt das Register r1 unverändert.

65. **MOV r1,r2**

\[ r2 \rightarrow r1 \]

Befehlsformat:

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 0 & 0 & 0 & 0 & \color{red}{r} & \color{red}{r} & \color{red}{r} & \color{red}{r} & \color{red}{r} & \color{red}{r} \\
\text{MOV} & r2 \ (4 \text{ Bit}) & r1 \ (4 \text{ Bit})
\end{array}
\]

Beschreibung:

’MOVe’ kopiert den Inhalt von Register r2 in das Register r1.

66. **MOV_CT r1,r2**

\[ r2 \rightarrow r1, \text{ wenn das Condition-Flag } 'TRUE' \text{ ist.} \]

Befehlsformat:

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 1 & 0 & 0 & 0 & \color{red}{r} & \color{red}{r} & \color{red}{r} & \color{red}{r} & \color{red}{r} & \color{red}{r} \\
\text{MOV_CT} & r2 \ (4 \text{ Bit}) & r1 \ (4 \text{ Bit})
\end{array}
\]

Beschreibung:

’MOVe when Condition-flag True’ kopiert den Inhalt von Register r2 in das Register r1, wenn das Condition-Flag ’TRUE’ ist.

67. **MOV_CF r1,r2**

\[ r2 \rightarrow r1, \text{ wenn das Condition-Flag } 'FALSE' \text{ ist.} \]

Befehlsformat:

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 0 & 0 & 0 & 0 & \color{red}{r} & \color{red}{r} & \color{red}{r} & \color{red}{r} & \color{red}{r} & \color{red}{r} \\
\text{MOV_CF} & r2 \ (4 \text{ Bit}) & r1 \ (4 \text{ Bit})
\end{array}
\]
Beschreibung:
"‘MOVe when Condition-flag False’" kopiert den Inhalt von Register r2 in das Register r1, wenn das Condition-Flag "‘FALSE’" ist.

68. ADDI r1,n6

\[ r1 + n6 \rightarrow r1 \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 0 & 1 & 0 & n & n & n & n & n & n & r & r & r & r \\
\hline
\text{ADDI} & n6 (6 \text{ Bit}) & r1 (4 \text{ Bit})
\end{array}
\]

Beschreibung:
"‘ADDition Immediate’" addiert eine 6 Bit Konstante zu Register r1 hinzu. Es ist somit möglich eine Zahl zwischen -32 und +31 zu addieren. Das Carry-Flag wird nicht berücksichtigt.

69. ADDI_CTOR1,n6

\[ r1 + n6 \rightarrow r1, \text{ wenn das Condition-Flag } "‘TRUE’" \text{ ist.} \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 1 & 0 & n & n & n & n & n & n & r & r & r & r \\
\hline
\text{ADDI_CT} & n6 (6 \text{ Bit}) & r1 (4 \text{ Bit})
\end{array}
\]

Beschreibung:
"‘ADDition Immediate when Condition True’" addiert eine 6 Bit Konstante zu Register r1 hinzu, wenn das Condition-Flag "‘TRUE’" ist. Es ist somit möglich eine Zahl zwischen -32 und +31 zu addieren. Das Carry-Flag wird nicht berücksichtigt.

70. ADDI_CFR1,n6

\[ r1 + n6 \rightarrow r1, \text{ wenn das Condition-Flag } "‘FALSE’" \text{ ist.} \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.
**A.2 Description**

**Befehlsformat:**

```
1 0 0 0 1 0 n n n n n n r r r r
ADDI, CF n6 (6 Bit) r1 (4 Bit)
```

**Beschreibung:**

"'ADDition Immediate when Condition False’‘ addiert eine 6 Bit Konstante zu Register r1 hinzu, wenn das Condition-Flag '‘False’‘ ist. Es ist somit möglich eine Zahl zwischen -32 und +31 zu addieren. Das Carry-Flag wird nicht berücksichtigt.

71. **ADD r1,r2**

\[
\mathrm{r1} + \mathrm{r2} \rightarrow \mathrm{r1}
\]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

**Befehlsformat:**

```
1 1 1 0 0 0 0 1 r r r r r r r
ADD r2 (4 Bit) r1 (4 Bit)
```

**Beschreibung:**

"'ADDition’‘ addiert die beiden Register r1 und r2. Das Carry-Flag wird nicht berücksichtigt (→ ADDC).

72. **ADD_CT r1,r2**

\[
\mathrm{r1} + \mathrm{r2} \rightarrow \mathrm{r1}, \text{ wenn das Condition-Flag '‘TRUE’‘ ist.}
\]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

**Befehlsformat:**

```
1 1 0 1 0 0 0 1 r r r r r r r
ADD_CT r2 (4 Bit) r1 (4 Bit)
```

**Beschreibung:**

"'ADDition when Condition True’‘ addiert die beiden Register r1 und r2, wenn das Condition-Flag '‘TRUE’‘ ist. Das Carry-Flag wird nicht berücksichtigt (→ ADDC_CT).
73. **ADD_CF r1,r2**

\[ r1 + r2 \rightarrow r1, \text{ wenn das Condition-Flag "FALSE" ist.} \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

**Befehlsformat:**

\[
\begin{array}{cccccccccc}
1 & 1 & 0 & 0 & 0 & 1 & r & r & r & r \\
ADD_CF & & & r2 \ (4 \ Bit) & & & r1 \ (4 \ Bit) \\
\end{array}
\]

**Beschreibung:**

"ADDition when Condition True" addiert die beiden Register \( r1 \) und \( r2 \), wenn das Condition-Flag "FALSE" ist. Das Carry-Flag wird nicht berücksichtigt (→ \( ADDC \_CT \)).

74. **ADDC r1,r2**

\[ r1 + r2 + \text{Carry} \rightarrow r1 \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

**Befehlsformat:**

\[
\begin{array}{cccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 0 & r & r & r \\
ADDC & & & r2 \ (4 \ Bit) & & & r1 \ (4 \ Bit) \\
\end{array}
\]

**Beschreibung:**

"ADDition with Carry" addiert die beiden Register \( r1 \) und \( r2 \), wobei das Carry-Flag berücksichtigt wird.

75. **ADDC\_CT r1,r2**

\[ r1 + r2 + \text{Carry} \rightarrow r1, \text{ wenn das Condition-Flag "TRUE" ist.} \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

**Befehlsformat:**

\[
\begin{array}{cccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 0 & r & r & r \\
ADDC\_CT & & & r2 \ (4 \ Bit) & & & r1 \ (4 \ Bit) \\
\end{array}
\]

Beschreibung:
"‘ADDition with Carry when Condition True’" addiert die beiden Register r1 und r2, wenn das Condition-Flag ”‘TRUE’" ist. Das Carry-Flag wird berücksichtigt.

76. ADDC\_CF r1,r2

\[ r1 + r2 + \text{Carry} \rightarrow r1, \text{wenn das Condition-Flag ”‘FALSE’" ist.} \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDC_CF</td>
<td>r2 (4 Bit)</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Beschreibung:
"‘ADDition with Carry when Condition True’" addiert die beiden Register r1 und r2, wenn das Condition-Flag ”‘FALSE’" ist. Das Carry-Flag wird berücksichtigt.

77. SUB r1,r2

\[ r1 - r2 \rightarrow r1 \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB</td>
<td>r2 (4 Bit)</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Beschreibung:
"‘SUBtraction’’ subtrahiert die beiden Register r1 und r2. Das Carry-Flag wird nicht berücksichtigt (→ SUBC).

78. SUB\_CT r1,r2

\[ r1 - r2 \rightarrow r1, \text{wenn das Condition-Flag ”‘TRUE’" ist.} \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.
Befehlsformat:

```
1 1 0 0 1 1 r r r r r r
```

SUB_CT \ r2 (4 Bit) \ r1 (4 Bit)

Beschreibung:

"'SUBtraction'" subtrahiert die beiden Register \( r_1 \) und \( r_2 \), wenn das Condition-Flag "'TRUE'" ist. Das Carry-Flag wird nicht berücksichtigt (\( \rightarrow \) SUBC).

79. **SUB_CF r1,r2**

\[ r_1 - r_2 \rightarrow r_1 \], wenn das Condition-Flag "'FALSE'" ist.

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

```
1 1 0 0 0 1 1 r r r r r r
```

SUB_CF \ r2 (4 Bit) \ r1 (4 Bit)

Beschreibung:

"'SUBtraction'" subtrahiert die beiden Register \( r_1 \) und \( r_2 \), wenn das Condition-Flag "'FALSE'" ist. Das Carry-Flag wird nicht berücksichtigt (\( \rightarrow \) SUBC).

80. **SUBC r1,r2**

\[ r_1 - r_2 - \text{Carry} \rightarrow r_1 \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

```
1 1 1 0 0 1 0 1 0 r r r r r r
```

SUBC \ r2 (4 Bit) \ r1 (4 Bit)

Beschreibung:

"'SUBtraction with Carry'" subtrahiert die beiden Register \( r_1 \) und \( r_2 \), wobei das Carry-Flag berücksichtigt wird.
81. **SUBC_CT r1,r2**

\[ r1 + r2 + \text{Carry} \rightarrow r1, \text{wenn das Condition-Flag } "\text{TRUE}" \]

ist.

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

```
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | r | r | r | r | r | r |
|----------------|----------------|----------------|----------------|----------------|----------------|
| SUBC_CT         | r2 (4 Bit)     | r1 (4 Bit)     |
```

Beschreibung:

"‘SUBtraction with Carry when Condition True’" subtrahiert die beiden Register r1 und r2, wenn das Condition-Flag "‘TRUE’" ist. Das Carry-Flag wird berücksichtigt.

82. **SUBC_CF r1,r2**

\[ r1 + r2 + \text{Carry} \rightarrow r1, \text{wenn das Condition-Flag } "\text{FALSE}" \]

ist.

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

```
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | r | r | r | r | r | r |
|----------------|----------------|----------------|----------------|----------------|----------------|
| SUBC_CF         | r2 (4 Bit)     | r1 (4 Bit)     |
```

Beschreibung:

"‘SUBtraction with Carry when Condition False’" subtrahiert die beiden Register r1 und r2, wenn das Condition-Flag "‘FALSE’" ist. Das Carry-Flag wird berücksichtigt.

83. **AND r1,r2**

\[ r1 \text{ and } r2 \rightarrow r1 \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:
Beschreibung: 
"AND" führt zu einer bitweisen Und-Verknüpfung der Register r1 und r2. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

84. **AND_CT r1,r2**

   r1 and r2 → r1, wenn das Condition-Flag "TRUE" ist. 
   Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Beschreibung: 
"AND when Condition True" führt zu einer bitweisen Und-Verknüpfung der Register r1 und r2, wenn das Condition-Flag "TRUE" ist. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

85. **AND_CF r1,r2**

   r1 and r2 → r1, wenn das Condition-Flag "FALSE" ist. 
   Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Beschreibung: 
"AND when Condition False" führt zu einer bitweisen Und-Verknüpfung der Register r1 und r2, wenn das Condition-Flag "FALSE" ist. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

86. **OR r1,r2**
r1 or r2 → r1
Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
r & r & r & r & r & r & r & r \\
\text{OR} & r2 (4 Bit) & r1 (4 Bit)
\end{array}
\]

Beschreibung:
"'OR'" führt zu einer bitweisen Oder-Verknüpfung der Register r1 und r2.
Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

87. **OR\_CT r1,r2**

r1 or r2 → r1, wenn das Condition-Flag "'TRUE'" ist.
Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
r & r & r & r & r & r & r & r \\
\text{OR\_CT} & r2 (4 Bit) & r1 (4 Bit)
\end{array}
\]

Beschreibung:
"'OR when Condition True'" führt zu einer bitweisen Oder-Verknüpfung
der Register r1 und r2, wenn das Condition-Flag "'TRUE'" ist. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

88. **OR\_CF r1,r2**

r1 or r2 → r1, wenn das Condition-Flag "'FALSE'" ist.
Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
r & r & r & r & r & r & r & r \\
\text{OR\_CF} & r2 (4 Bit) & r1 (4 Bit)
\end{array}
\]
Beschreibung:

"’OR when Condition False’" führt zu einer bitweisen Oder-Verknpfung der Register r1 und r2, wenn das Condition-Flag ”’FALSE’" ist. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

89. **EOR r1,r2**

\[ r1 \text{ eor } r2 \rightarrow r1 \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOR</td>
<td>r2 (4 Bit)</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Beschreibung:

"’Exclusive OR’" führt zu einer bitweisen XOR-Verknpfung der Register r1 und r2. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

90. **EOR_CT r1,r2**

\[ r1 \text{ eor } r2 \rightarrow r1, \text{ wenn das Condition-Flag } ”’TRUE’" \text{ ist.} \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOR_CT</td>
<td>r2 (4 Bit)</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Beschreibung:

"’Exclusive OR when Condition True’" führt zu einer bitweisen XOR-Verknüpfung der Register r1 und r2, wenn das Condition-Flag ”’TRUE’" ist. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

91. **EOR_CF r1,r2**

\[ r1 \text{ eor } r2 \rightarrow r1, \text{ wenn das Condition-Flag } ”’FALSE’" \text{ ist.} \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.
A.2 Description

Befehlsformat:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOR_CF</td>
<td>r2 (4 Bit)</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Beschreibung:
"‘Exclusive OR when Condition False’" führt zu einer bitweisen XOR-Verknüpfung der Register r1 und r2, wenn das Condition-Flag ”‘FALSE’” ist. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

92. NOT r1

\[ r1 \rightarrow \neg r1 \]

Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Beschreibung:
"‘NOT’” invertiert das Register r1 bitweise. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

93. NOT_CT r1

\[ r1 \rightarrow \neg r1 \]

, wenn Condition-Flag ”‘TRUE’” ist. Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

Befehlsformat:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT_CT</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Beschreibung:
"‘NOT when Cond-flag True’” invertiert das Register r1 bitweise, wenn das Condition-Flag ”‘TRUE’” ist. Ansonsten bleibt r1 unverändert. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.
94. **NOT CF r1**

\[ r1 \rightarrow \neg r1 \]

, wenn Condition-Flag ”‘FALSE’” ist. Carry-, Neg-, Overflow- und Zero-Flag werden manipuliert.

**Befehlsformat:**

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
\text{NOT CF} & r1 \text{ (4 Bit)}
\end{array}
\]

**Beschreibung:**

”‘NOT when Cond-flag False’” invertiert das Register r1 bitweise, wenn das Condition-Flag ”‘FALSE’” ist. Ansonsten bleibt r1 unverändert. Die Status-Bits werden wie bei arithmetischen Operationen gesetzt.

95. **TRAP n4**

\[ \text{ExceptionVector}(n4) \rightarrow \text{PC} \]

\[ \text{Status Register} \rightarrow \text{Save Status Register} \]

**Befehlsformat:**

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 0 & 1 & 1 & n & n \\
\text{TRAP} & n4 \text{ (4 Bit)}
\end{array}
\]

**Beschreibung:**

”‘TRAP’” ist einen Trap (= Software Interrupt) aus. Der Prozessor unterstützt 16 Traps, die Trap Vektoren liegen in der Exception Vector Table. Mit der Konstanten n4 wird bestimmt, welcher Trap ausgelöst wird. Wie bei allen Exceptions, wird auch bei einem Trap das Processor Status Register gerettet. Status Register und Save Status Register befinden sich in dem Processor Control Module.

96. **TRAP_CT n4**

wenn Condition-Flag ”‘TRUE’” ist
A.2 Description

ExceptionVector(n4) → PC
Status Register → Save Status Register

Befehlsformat:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAP_CT</td>
<td>n4 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Beschreibung:
"‘TRAP when Cond-flag True’" ist einen Trap (= Software Interrupt) aus. Der Prozessor unterstützt 16 Traps, die Trap Vektoren liegen in der Exception Vector Table. Mit der Konstanten n4 wird bestimmt, welcher Trap ausgelöst wird. Wie bei allen Exceptions, wird auch bei einem Trap das Processor Status Register gerettet. Status Register und Saved Status Register befinden sich in dem Processor Control Module. Ist das Condition-Flag "‘FALSE’", wird der Befehl ignoriert.

97. TRAP_CF n4

wenn Condition-Flag "‘FALSE’" ist

ExceptionVector(n4) → PC
Status Register → Save Status Register

Befehlsformat:

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAP_CF</td>
<td>n4 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Beschreibung:
"‘TRAP when Cond-flag False’" ist einen Trap (= Software Interrupt) aus. Der Prozessor unterstützt 16 Traps, die Trap Vektoren liegen in der Exception Vector Table. Mit der Konstanten n4 wird bestimmt, welcher Trap ausgelöst wird. Wie bei allen Exceptions, wird auch bei einem Trap das Processor Status Register gerettet. Status Register und Saved Status Register befinden sich in dem Processor Control Module. Ist das Condition-Flag "‘TRUE’", wird der Befehl ignoriert.
98. **JSR r1**

PC → rRTS(Register 14 im Registerfile)
r1 → PC

Befehlsformat:

```
1 1 1 0 1 1 1 0 0 0 0 0 r r r r
```

Beschreibung:

"'Jump to SubRoutine - save pc in register rts'" realisiert einen Unterprogrammaufruf. Der Wert des PC (Program Counter) wird automatisch in das Register RTS (Register 14 im Registerfile) abgespeichert. Anschliessend wird der Inhalt des Registers r1 in den Program Counter geladen.

99. **JSR_CT r1**

wenn Condition-Flag "'TRUE'" ist

PC → rRTS(Register 14 im Registerfile)
r1 → PC

wenn Condition-Flag "'FALSE'" ist

PC + 1 → PC

Befehlsformat:

```
1 1 0 1 1 1 1 0 0 0 0 r r r r
```

Beschreibung:

"'Jump to SubRoutine when Cond-flag True - save pc in register rts'" realisiert einen bedingten Unterprogrammaufruf. Wenn das Condition-Flag "'TRUE'" ist, wird der Wert des PC (Program Counter) automatisch in das Register RTS (Register 14 im Registerfile) abgespeichert. Anschließend wird der Inhalt des Registers r1 in den Program Counter geladen. Ist das Condition-Flag "'FALSE'", so erfolgt kein Sprung.
100. **JSR_CF r1**

wenn Condition-Flag ”’FALSE’” ist

\[
\text{PC} \rightarrow \text{rRTS (Register 14 im Registerfile)}
\]

\[
r1 \rightarrow \text{PC}
\]

wenn Condition-Flag ”’TRUE’” ist

\[
\text{PC + 1} \rightarrow \text{PC}
\]

Befehlsformat:

\[
\begin{array}{ccccccccc}
1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & r & r & r & r \\
\end{array}
\]

\[
\begin{array}{cc}
\text{JSR_CF} & \text{r1 (4 Bit)}
\end{array}
\]

Beschreibung:

”’Jump to SubRoutine when Cond-flag False - save pc in register rts’”
realisiert einen bedingten Unterprogrammaufruf. Wenn das Condition-
Flag ”’FALSE’” ist, wird der Wert des PC (Program Counter) automa-
tisch in das Register RTS (Register 14 im Registerfile) abgespeichert.
Anschließend wird der Inhalt des Registers r1 in den Program Counter
geladen. Ist das Condition-Flag ”’TRUE’”, so erfolgt kein Sprung.

101. **JMPI a10**

\[
\text{PC + n6} \rightarrow \text{PC}
\]

Befehlsformat:

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 0 & 1 & 1 & a & a & a & a & a & a & a \\
\end{array}
\]

\[
\begin{array}{cc}
\text{JMPI} & \text{a10 (10 Bit)}
\end{array}
\]

Beschreibung:

”’JuMP Immediate’” ist ein relativer Sprungbefehl. Die Zieladresse
setzt sich aus dem Offset und der Adresse des Befehls selbst zusammen.
Der Offset liegt zwischen -512 und +511. Die Rcksprungadresse wird
nicht gespeichert.

102. **JMPI_CT a10**
wenn Condition-Flag "'TRUE'" ist
PC + a10 → PC

wenn Condition-Flag "'FALSE'" ist
PC + 1 → PC

Befehlsformat:

```
   1 0 0 1 1 1 a a a a a a a a a a
```

Beschreibung:

103. **JMPI_CF a10**

wenn Condition-Flag "'FALSE'" ist
PC + a10 → PC

wenn Condition-Flag "'TRUE'" ist
PC + 1 → PC

Befehlsformat:

```
   1 0 0 0 1 1 a a a a a a a a a a a
```

Beschreibung:
104. **JMP r1**

r1 → PC

Befehlsformat:

```
  1 1 1 0 1 1 1 1 0 0 0 r  r  r  r  
  |   |   |   |   |   |   |   |   |   |   |
  | 1 1 1 0 1 1 1 1 0 0 0 r  r  r  r |
  |   |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |   |
JMP  r1 (4 Bit)
``` 

Beschreibung:

"JuMP" ist ein Sprungbefehl, die Zieladresse steht im Register r1. Die Rücksprungadresse wird nicht gespeichert.

105. **JMP_CT r1**

wenn Condition-Flag "TRUE" ist

r1 → PC

wenn Condition-Flag "FALSE" ist

PC + 1 → PC

Befehlsformat:

```
  1 1 0 1 1 1 1 1 0 0 0 r  r  r  r  
  |   |   |   |   |   |   |   |   |   |   |
  | 1 1 0 1 1 1 1 1 0 0 0 r  r  r  r |
  |   |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |   |
JMP_CT  r1 (4 Bit)
``` 

Beschreibung:

"JuMP when Cond-flag True" ist ein bedingter Sprungbefehl, die Zieladresse steht im Register r1. Die Rücksprungadresse wird nicht gespeichert. Ist das Condition-Flag "FALSE", so erfolgt kein Sprung.

106. **JMP_CF r1**

wenn Condition-Flag "FALSE" ist

r1 → PC

wenn Condition-Flag "TRUE" ist

PC + 1 → PC
Befehlsformat:

```
1 1 0 0 1 1 1 1 0 0 0 0  r  r  r  r
JMP_CF  r1 (4 Bit)
```

Beschreibung:

"'JuMP when Cond-flag False'" ist ein bedingter Sprungbefehl, die Zieladresse steht im Register r1. Die Rücksprungadresse wird nicht gespeichert. Ist das Condition-Flag "'TRUE'", so erfolgt kein Sprung.

107. **LDW r1,r2**

MEM(r2) → r1

Befehlsformat:

```
1 1 1 1 0 0 0 0 0 0  r  r  r  r  r  r
LDW  r2 (4 Bit)  r1 (4 Bit)
```

Beschreibung:

"'LoaD Word'" liet das Datenwort von der Adresse im Register r2, und kopiert es in Register r1. Dieser Befehl wird in der 32 Bit Konfiguration auch zum Ansprechen der Extension Module benutzt, da diese in den Datenspeicher gemappt sind und somit wie einfache Speicherstellen angesprochen werden können.

Dieser Befehl steht nur in einer 32 Bit Konfiguration zur Verfügung.

108. **LDH r1,r2**

MEM(r2) → r1

Befehlsformat:

```
1 1 1 1 0 0 0 1  r  r  r  r  r  r  r
LDH  r2 (4 Bit)  r1 (4 Bit)
```
Beschreibung:
”‘LoaD Half word’” liest ein 16 Bit breites Datenwort von der Adresse im Register r2, und kopiert es in Register r1. Sollte das Register r1 32 Bit breit sein, werden die oberen Bit vorzeichenrichtig erweitert. Dieser Befehl wird in der 16 Bit Konfiguration auch zum Ansprechen der Extension Module benutzt, da diese in den Datenspeicher gemappt sind und somit wie einfache Speicherstellen angesprochen werden können.

109. **LDHU r1,r2**

MEM(r2) → r1

Befehlsformat:

```
<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDHU</td>
<td>r2 (4 Bit)</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Beschreibung:
”‘LoaD Half word Unsigned’” liest ein 16 Bit breites Datenwort von der Adresse im Register r2, und kopiert es in Register r1. Sollte das Register r1 32 Bit breit sein, werden die oberen Bit mit 0 beschrieben. Dieser Befehl steht nur in einer 32 Bit Konfiguration zur Verfügung.

110. **LDB r1,r2**

MEM(r2) → r1

Befehlsformat:

```
<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDB</td>
<td>r2 (4 Bit)</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Beschreibung:
”‘LoaD Byte’” liest ein 8 Bit breites Datenwort von der Adresse im Register r2, und kopiert es in Register r1. Die oberen Bit werden vorzeichenrichtig erweitert.

111. **LDBU r1,r2**
MEM(r2) → r1

Befehlsformat:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDBU</td>
<td>r2 (4 Bit)</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Beschreibung:
"'LoaD Byte Unsigned'" liet ein 8 Bit breites Datenwort von der Adresse im Register r2, und kopiert es in Register r1. Die oberen Bit werden mit 0 beschrieben.

112. **STW r1,r2**

r1 → MEM(r2)

Befehlsformat:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>STW</td>
<td>r2 (4 Bit)</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Beschreibung:

113. **STH r1,r2**

r1 → MEM(r2)

Befehlsformat:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>STH</td>
<td>r2 (4 Bit)</td>
<td>r1 (4 Bit)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Beschreibung:
”‘STore Half word’” speichert den Inhalt vom Register r1 in den Daten-
speicher. In der 32 Bit Konfiguration werden nur die unteren 16 Bit in
den Datenspeicher abgelegt. Die dafür verwendet Zieladresse steht im
Register r2. Dieser Befehl wird in der 16 Bit Konfiguration auch zum
Ansprechen der Extension Module benutzt, da diese in den Daten-
speicher gemappt sind und somit wie einfache Speicherstellen ange-
sprochen werden können.

114. STB r1,r2

\[ r1 \rightarrow \text{MEM}(r2) \]

Befehlsformat:

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
\text{STB} & r2 (4 \text{ Bit}) & r1 (4 \text{ Bit}) \\
\end{array}
\]

Beschreibung:
”‘STore Byte’” speichert die unteren 8 Bit vom Register r1 in den
Datenspeicher. Die dafür verwendet Zieladresse steht im Register r2.

115. RTS

\[ r\text{RTS (Register 14)} \rightarrow \text{PC} \]

Befehlsformat:

\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
\text{RTS} & r14 (fix) \\
\end{array}
\]

Beschreibung:
”‘ReTurn from Subroutine’” ist einen Rücksprung aus einem Unterpro-
gramm aus. Die Rücksprunganadresse steht im Register RTS (Register
14 im Registerfile). Dieses Register wird durch den Unterprogram-
maufruf mit dem Befehl JSR automatisch beschrieben. Es kann aber
auch manuell ausgelesen und beschrieben werden.
116. **RTE**

\[ \text{rRTE (Register 15) → PC} \]

Save Status Register → Status Register

Befehlsformat:

\[
\begin{array}{cc}
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\hline
\text{RTS} & r15 \text{ (fix)}
\end{array}
\]

Beschreibung:

“ReTurn from Exception” ist einen Rückprung in den normalen Programmcode aus. Bei einer Exception (Interrupt oder Trap) wird die Rücksprungadresse automatisch im Register RTE (Register 15 im Registerfile) abgelegt. Dieser Befehl ldt den Inhalt dieses Registers in den Program Counter und stellt das Status Register wieder her. Status Register und Save Status Register befinden sich im Processor Control Module.

117. **LDVEC r1,n5**

\[ \text{VecTab(n5) → r1} \]

Befehlsformat:

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 1 & n & n & n & n & r & r & r & r \\
\hline
\text{LDVEC} & n5 \text{ (5 Bit)} & r1 \text{ (4 Bit)}
\end{array}
\]

Beschreibung:

“LoaD exception VECtor” liet die ”EXception Vector Table” aus. Die Exception Vector Table beinhaltet die Adressen, an denen im Falle eines Interrupts oder eines Traps gesprungen wird. Der Exception Vector mit den Index n5 wird ausgelesen, und in das Register r1 kopiert. n5 kann zwischen -16 und +15 liegen. Der Prozessor unterstützt 16 Interrupts und 16 Traps. Die Exception Vector Table ist so aufgebaut, dass die negativen Indizes (-16 bis -1) den Interrupt Vektoren zugeordnet sind und die positiven Indizes (0 bis 15) die Trap Vektoren beinhaltet.
118. **STVEC r1,n5**

    \[ r1 \rightarrow \text{VecTab}(n5) \]

Befehlsformat:

```
11111111110nnnnrrrr
STVEC   n5 (5 Bit)   r1 (4 Bit)
```

Beschreibung:

"STore exception VECtor" baut die "EXception Vector Table" auf. Die Exception Vector Table beinhaltet die Adressen, an denen im Falle eines Interrupts oder eines Traps gesprungen wird. An die Stelle n5 in der Exception Vector Table wird der Inhalt vom Register r1 kopiert. n5 kann zwischen -16 und +15 liegen. Der Prozessor unterstützt 16 Interrupts und 16 Traps. Die Exception Vector Table ist so aufgebaut, dass die negativen Indizes (-16 bis -1) den Interrupt Vektoren zugeordnet sind und die positiven Indizes (0 bis 15) die Trap Vektoren beinhaltet.

119. **NOP**

No Operation

Befehlsformat:

```
111111111100000000
NOP
```

Beschreibung:

"No OPeration" führt keine Operation aus.

120. **IllOp**

Illegal Opcode

Befehlsformat:

```
1111111111111111
IllOp
```

Beschreibung:
References


