FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

The SPEAR2 Hardware/Software Interface

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Martin Walter
Matrikelnummer 9925269

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Mitwirkung: Dipl.-Ing. Jakob Lechner

Wien, 24.02.2011

(Unterschrift Verfasser/in)

(Unterschrift Betreuer)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung

Martin Walter
Inzersdorferstrasse 99/1/17
1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24. Februar 2011

Kurzfassung

Das Hardware/Software Interface einer Rechnerarchitektur umfasst all jene Möglichkeiten mittels derer Informationen über die Grenze zwischen Hardware und Software hinweg übermittelt werden können. Die Bestandteile des Hardware/Software Interfaces finden sich daher in einer Vielzahl von Systemprogrammen, allen voran Compiler, Assembler und Linker wieder, in denen sie die Regeln für die Übersetzung von Quellcode einer Hochsprache in ein ausführbares Programm in der jeweiligen Maschinensprache vorgeben. Durch die kollektive Einhaltung dieser Regeln für eine bestimmte Architektur wird es ermöglicht, Anwendungen einmalig zu kompilieren und auf einem beliebigen Prozessor ausführen zu lassen, der eben diese Architektur implementiert. Im Rahmen dieser Diplomarbeit wird das Hardware/Software Interface des SPEAR2, einer am Institut für Technische Informatik der Technischen Universität Wien entwickelten soft-core Architektur, von Grund auf erarbeitet. Die Erkenntnisse werden dabei sowohl von einer bestehenden Sammlung mittlerweile nicht mehr zeitgemäßer Systemprogramme, als auch von den Quellcodes der Architektur selbst, gewonnen. Schlussendlich werden sich diese als unverzichtbar erweisen, wenn die Portierung und Integration einer Sammlung aktueller Entwicklungs-Tools, vorwiegend aus dem GNU Projekt[1], in eine funktionsfähige, wohl aufeinander abgestimmte, Toolchain vorgenommen werden soll.

[1] The GNU Project - Free Software Foundation (FSF), http://www.gnu.org/software/software.html

Abstract

The *hardware/software interface* of a computer architecture combines all means by which information can be exchanged across the hardware/software boundary, the place where the hardware and the software meet. As such, its components can be found implemented in a variety of systems software, such as compilers, assemblers and linkers, in which they lay out the rules by which high-level source code is translated into an executable machine language representation. At large, the collective adherence to the hardware/software interface of an architecture enables us to compile an application once and have it executed on virtually any processor which is compliant with this particular architecture. In this thesis we will elaborate on the hardware/software interface of the *SPEAR2 architecture*, a soft-core processor architecture developed at the Department of Computer Engineering at the Vienna University of Technology, from the very ground up. The information will be derived partly from a collection of legacy software development tools, as well as from the plain sources of the architecture itself and be presented in a high degree of accuracy. Finally, our elaborations will prove essential when we turn to the porting and integration of a collection of state-of-the-art software development tools, primarily chosen from the GNU Project[2], into a functional, well-matched, toolchain.

[2] The GNU Project - Free Software Foundation (FSF), http://www.gnu.org/software/software.html

Danksagung

Ich bedanke mich bei meinem Betreuerteam, Professor Andreas Steininger und Jakob Lechner, für die zahlreichen konstruktiven Diskussionen, Anregungen und Verbesserungsvorschläge und damit ausgezeichnete Betreuung dieser Diplomarbeit. Größter Dank für ihre Unterstützung gilt meiner geliebten Katharina, deren liebevolle Aufopferung nach der Geburt unseres Sohnes Adam eine Fertigstellung in meinem Sinne erst ermöglichte.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The *SPEAR2*[1] is a soft-core processor architecture written in VHDL, which is designed and actively maintained by the members of the Embedded Computing Systems Group of the Department of Computer Engineering at the Vienna University of Technology [Fle08]. A configurable and extendable processor core makes the SPEAR2 especially interesting for education: the moderate complexity allows the interested student to become familiar with practical concepts of modern pipelined processor design.

1.1 State of the Art

The *hardware/software interface* of a computer architecture combines all means by which information can be exchanged across the hardware/software boundary, the place where the hardware and the software meet. As such, the components of the hardware/software interface can be found implemented in a variety of systems software, such as compilers, assemblers and linkers, in which they lay out the rules by which high-level source code is translated into an executable machine language representation.

While some parts of the hardware/software interface are immediately derived from the *instruction set architecture*, i.e., the programmer-visible properties of the underlying hardware architecture, such as the order in which multi-byte data are stored in memory, or the dimension of the datapath limiting the size of a datum that can be loaded from or stored in memory within a single operation, some are defined by convention: systems software integrators agree on an *application binary interface*, a collection of rules, such as the rules of data representations, the register usage conventions or the procedure calling conventions, which define how procedure arguments and return values of procedure calls are passed between a calling and the called procedure.

[1] **S**calable **P**rocessor for **E**mbedded **A**pplications in **R**eal-Time Environments

It might very well be because of its scattered nature that the term hardware/software interface is not well-defined in literature. In fact, even the doyens of computer architecture, Hennessy and Patterson, avoided to present a concise definition in their highly acclaimed standard reference [DAP07]. Because we would not dare to give a definition of our own, we will instead keep relating to the hardware/software interface of an architecture in terms of the *instruction set architecture* and the *application binary interface* throughout this work. The relations between these two are well understood and are presented in Figure 1.1.

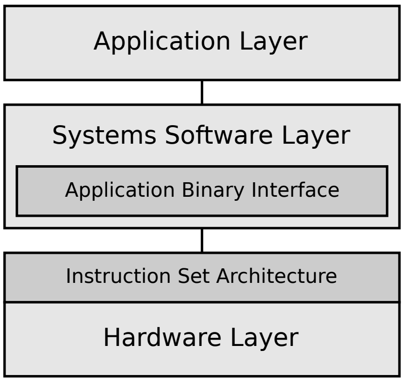


Figure 1.1: The Hardware/Software Interface

1.2 Motivation

The following example should make the criticality of the hardware/software interface for systems software integration obvious. In the C programming language, `int *ptr;` declares a pointer to an integer type. Nothing to get excited about you might think, but quite some important information is associated along with this seemingly trivial statement:

- The compiler must be aware of the dimension of the pointer type, reflecting the *memory addressing capacity*, and the size of the `int` data type, which, by convention, matches the *word size* of the underlying hardware architecture.

- The dereferencing of the `ptr` variable unveils the actual object pointed to in memory. Performing an operation on the object, such as through the assignment operation `*ptr = 0`, instructs the compiler to emit a predefined sequence of assembly language instructions, which manage the manipulation of an object in memory by the size of the associated data type, `int` in this case.

- Presuming that `ptr` will eventually get shared between functions, the compiler would have to know the rules by which arguments or return values should be handed over between the calling procedure and the called procedure. Depending on a number of influential factors, such as the number and size of other procedure arguments, and the data types they represent, arguments and return values can either be provided in dedicated registers, or in memory, or both.

Now, neither the sizes of the pointer type or the `int` data type, nor the exact location of an argument or a return value is currently of importance. What is important, however, is to understand that all systems software must collectively adhere to a variety of rules and conventions and must embody the characteristics of a particular hardware architecture in order to achieve interoperability and compatibility between binary programs. Generally speaking, the collective adherence to the hardware/software interface of an architecture enables us to compile an application once and have it executed on virtually any processor which is compliant with this particular architecture.

The objective of this thesis is to elaborate on the instruction set architecture and the application binary interface of the SPEAR2 architecture in a high degree of accuracy. Derived partly from a collection of legacy software development tools, as well as from the plain sources of the architecture itself, this first analytical part will later prove essential when we turn to the porting and integration of a collection of state-of-the-art software development tools, primarily chosen from the GNU Project[2], into a functional, well-matched, toolchain.

Doubtless, the GNU toolchain is a de facto standard on GNU/Linux operating systems and has become available for a large number of both mainstream and exotic architectures. Additionally, being open source software, its sources can be used, studied, copied, modified and redistributed freely under the terms of the GNU General Public License (GPL) [3].

1.3 Outline

This thesis is organized as follows: in Chapter 2 we will develop the *SPEAR2 Instruction Set Architecture* from the very ground up. Our explanations will be centered around the components of a basic computer system model, a hypothetical computer if you wish, which comprises a processor, memory, and input/output devices, interconnected by a system bus. After having become familiar with what the hardware architecture has to offer, we will turn our focus towards *systems software* in Chapter 3. Therein, we will explain in which ways the software development tools of our recent toolchain contribute to the translation of high-level source code into an executable machine language representation. This is followed by an elaboration on the *SPEAR2 Embedded Application Binary Interface*, an application binary interface in the absence of an operating system. The integration of the software development tools is outlined in Chapter 4, in which we will describe the realizations of selected aspects of the hardware/software interface within the GNU toolchain. This thesis is then concluded in Chapter 5.

[2] The GNU Project - Free Software Foundation (FSF), http://www.gnu.org/software/software.html

[3] The GNU General Public License v3.0, http://www.gnu.org/licenses/gpl.html

Chapter 2

Hardware Layer

The hardware layer of any computing system can be characterized by the following three basic components, which are interconnected by a system bus: a *processor*, *memory* for holding program instructions and data, as well as *input/output* devices, which allow for communication with the environment, i.e., the outside world. This very high-level basic computer system components model [Dan05, Section 2.1] is depicted in Figure 2.1.

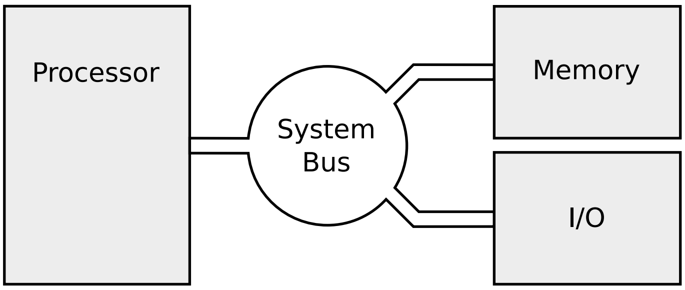


Figure 2.1: Basic Computer System Components Model

Throughout this chapter we will further refine this basic model at appropriate levels of detail to, in the end, come up with a definition of the *SPEAR2 Instruction Set Architecture* (ISA). The ISA is the portion of a computing system that is visible to the programmer or compiler writer [DAP07, Section 2.1], defining the features of the instruction set and the important characteristics of an architectural design [Noe05, Section 4.1].

2.1 Processor

In computer architecture, a processor is divided into two interacting parts: *datapath* and *control*. The control unit analyzes an instruction after it was fetched by the datapath to then control its processing through the datapath, i.e., its execution, by setting the control lines at the contained structures and the control bus respectively, as sketched in Figure 2.2.

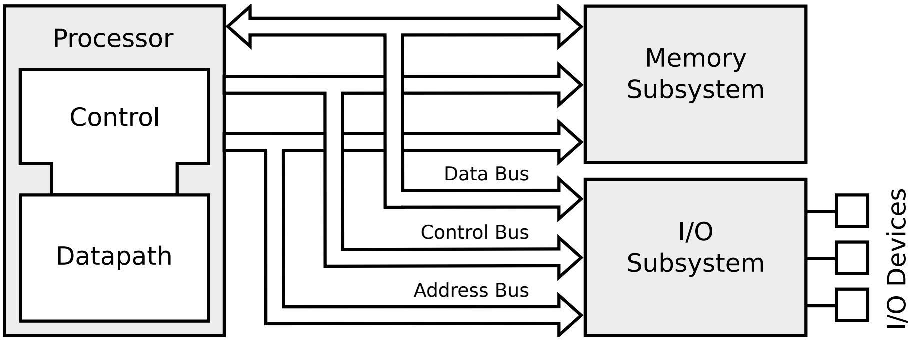


Figure 2.2: Extended Computer System Components Model

2.1.1 Datapath

A *datapath unit* contains structures whose dimensions are significant for the performance of an architecture: the *arithmetic/logical unit* (ALU) and the *register file*. Since the operands of an arithmetic/logical operation can either be located in a register or in memory, the size of the datapath for data transfers between these structures, as well as, of course, the size of the data bus, determines the maximum size of a datum that can be processed by the datapath in a single operation. This metric defines what the computer architect refers to as the *word size* of an architecture. An architecture with a word size of n bits is called an *n-bit architecture*.

Another metric that is related to the word size is the *memory addressing capacity* of an architecture and is determined by the size of the address bus. An n-bit address bus can address up to 2^n bytes of physical memory. Although it applies to the majority of modern architectures that the word size equals the memory addressing capacity, this does not necessarily need to be the case: the Intel 8086 processor, for example, has a 16-bit data bus and a 20-bit address bus [Dan05, Section 2.1].

SPEAR2 ISA The SPEAR2 features a *statically configurable datapath* which allows for 16-bit and 32-bit operation modes. Consequently, the SPEAR2 can appear either as a 16-bit or 32-bit architecture for the benefit of less complexity on the one hand and better performance on the other hand, but with slightly *differing instruction sets*. In both configurations the word size equals the memory addressing capacity.

2.1.1.1 Register File

The *register file*, or *register set*, of a processor is a comparatively small but particularly fast type of internal storage that is usually separated into a set of *general purpose registers* and a set of *special purpose registers*. While the general purpose registers hold instruction operands that are immediately or frequently used, the special purpose registers define what is commonly referred to as the *architectural state*, or *context* of a processor.

SPEAR2 ISA The SPEAR2 register file comprises 14 general purpose registers and 2 special purpose registers, all of which are accessible to the user programs, as shown in Table 2.1. The size of a single register depends on the configuration of the datapath unit, as described in Section 2.1.1.

Number	Assembly Mnemonic	Function
0	r0	-
1	r1	-
2	r2	-
3	r3	-
4	r4	-
5	r5	-
6	r6	-
7	r7	-
8	r8	-
9	r9	-
10	r10	-
11	r11	-
12	r12	-
13	r13	-
14	r14/rts	return from subroutine
15	r15/rte	return from exception

Table 2.1: SPEAR2 Register File

Please bear in mind that, even though any of the registers in the register file could be used as instruction operands whenever a register operand is expected, it is generally not advised to use any of the special purpose registers in a general purpose context since this could inadvertently alter, yet destroy, the architectural state of the processor.

Return-from-subroutine (rts) register The return-from-subroutine register holds the return address in case of a subroutine call: upon execution of a *jump-to-subroutine* (jsr) *instruction* the value of this register is set to the value of the program counter, which is described in Section 2.1.1.2. Subsequently, the program counter is set to the address of the requested subroutine, which is provided as an instruction operand, so that control will be transferred to the subroutine at the next instruction fetch cycle. Later on, upon execution of the *return-from-subroutine* (rts) *instruction*, which completes any subroutine, the program counter is restored from this register and control is returned to the calling program.

Return-from-exception (rte) register The return-from-exception register holds the return address in case of an exception: before control is transferred to an exception handler the value of this register is set to the value of the program counter. Subsequently, the program counter is set to the address of the exception handler routine. Later on, upon execution of the *return-from-exception* (rte) *instruction*, which completes any exception handler routine, the program counter is restored from this register and control is returned to the interrupted program.

Please note that any additional special purpose registers, such as the *processor status register* and *frame pointer registers* for the realization of a call stack are defined outside of the register file, in the *processor control module*, as described in Section 2.1.2.2.

2.1.1.2 Program Counter

The program counter is a special purpose register which holds the memory address of the next instruction to be fetched by the datapath unit at the next instruction fetch cycle.

Flow of control Observe that, because of the per default sequential *flow of control*, i.e., the order in which the particular instructions of a program are executed by the processor, the program counter is always incremented to point to the succeeding instruction at the end of the instruction fetch cycle. *Control transfer instructions*, such as jumps, procedure calls and returns are provided to alter the flow of control from within a program. *Exceptions*, as described in Section 2.1.2.1, are an additional means by which control is transferred to an exception handler as a response of the processor to some internal or external event. Because of these characteristics it is said that the program counter *manages* the flow of control [Dan05, Section 2.4].

SPEAR2 ISA The SPEAR2 program counter is neither part of the register file nor is it directly accessible to the user programs. *Control transfer instructions*, as described in Appendix A.2.4, must be used to alter the flow of control. The size of the program counter depends on the configuration of the datapath unit, as described in Section 2.1.1.

2.1.2 Control

Besides commanding the operation of the datapath, the *control unit* handles the flow of control, as described in Section 2.1.1.2. In advanced processor architectures the control unit may carry out a variety of highly complicated tasks, such as branch prediction, cache control, instruction prefetching, out-of-order execution, et cetera [Nur07, Chapter 2].

2.1.2.1 Exceptions

Besides jumps, procedure calls and returns, exceptions pose an additional means by which the flow of control of a program is altered. In distinction to jumps, which provide only a one-way transfer of control, a procedure call provides additional mechanisms to return control to the point of procedure invocation at the calling site when the called procedure has completed.

Exceptions are similar to procedure calls in that control is transferred to an *exception handler routine*, or *exception handler*, and control is returned to the interrupted program again once the exception handling has completed. Still, exceptions differ from procedure calls insofar as control transfers are accompanied by a *context switch*, a process in which the *context* of the processor, is stored and restored in order to allow the interrupted program to be resumed from the same state of execution once the exception handler has returned. In comparison to procedure calls, exceptions can originate both from software and from hardware, which leads to the following, widely adopted, taxonomy [DH07, Section 6.7.2]:

Traps An exception that is initiated by software, that is, caused by the execution of a dedicated instruction, is called a *trap*. Because of its origin, a trap can be considered as an anticipated or planned event and is therefore said to be of *synchronous* nature [Dan05, Chapter 14]. Traps are commonly used to handle events which are related to the instruction execution itself, such as *instruction faults*, as for example, *divide by zero*, *illegal instruction* and the likes. The trap mechanisms can further be used to implement the *system call interface* of an operating system through which control is transferred to the kernel for the execution of privileged code.

Interrupts An exception that is initiated by a hardware request is called an *interrupt*. The fact that both internal and external hardware may attain the attention of the processor independent from the course of the user program makes the interrupt mechanism a truly powerful concept. Because of its origin an interrupt can be considered as an unanticipated or unplanned event and is therefore said to be of *asynchronous* nature [Dan05, Chapter 14].

Prioritized Interrupts As a rule, the interrupts of a processor are categorized into *prioritization levels*, in order to be able to balance *interrupt latency* versus *throughput* in the potential scenario where multiple sources issue an interrupt request simultaneously. By applying this scheme, higher prioritized interrupts have precedence over lower prioritized interrupts and, as a consequence, a higher level interrupt may not be disrupted by another interrupt of the same level or a lower level. Typically, prioritization levels are naturally numbered starting at 1 with the highest priority in consecutive order with the associated priorities decreasing.

Maskable vs. Non-maskable Interrupts Maskable interrupts are interrupts that may be ignored by the system through the use of interrupt masking techniques, commonly by setting or clearing an associated flag in an *interrupt mask register*. Non-maskable interrupts (NMI), on the other hand, are interrupts to signal events which must not be missed and therefore cannot be ignored by standard interrupt masking techniques, as for example, non-recoverable hardware errors or system timer interrupts. In systems with prioritized interrupts the interrupt with the highest priority in the system is generally non-maskable [Noe05, Section 8.1.1].

Vectored Interrupts vs. Polled Interrupts The term *vectored interrupts* names an interrupt handling scheme by which the interrupt controller determines the origin of the interrupt request to then direct the processor towards an associated interrupt handler. As opposed to this, with *polled interrupts*, the processor is dispatched to a unique interrupt handler that is typically located at a fixed address in memory. It is then the task of the programmer to poll all peripherals to thereby identify the hardware device which caused the interrupt within the interrupt handler routine.

An *interrupt vector* is an entry in an *interrupt vector table* which denotes the address of an interrupt handler in memory. When a vectored interrupt is requested by a hardware device, and is accepted by the interrupt controller, a number that uniquely identifies the interrupt line on which the request was received is used to look up the interrupt vector in the interrupt vector table. Interrupt vector tables are maintained through dedicated instructions.

It needs to be stressed that for every interrupt which is not masked, both a dedicated interrupt vector and an interrupt handler must be provided. Otherwise, if the respective interrupt became active, the processor would most likely be directed to a random address in memory and never return back to the interrupted program. Therefore, it is generally within the responsibilities of the *C runtime* or of an *operating system* to populate the interrupt vector table upon system startup.

SPEAR2 ISA The SPEAR2 implements an exception vector table comprising a *trap vector table* for 16 trap vectors and an *interrupt vector table* to hold another 16 interrupt vectors, as depicted in Figure 2.3. The size of a vector depends on the configuration of the datapath unit, as described in Section 2.1.1.

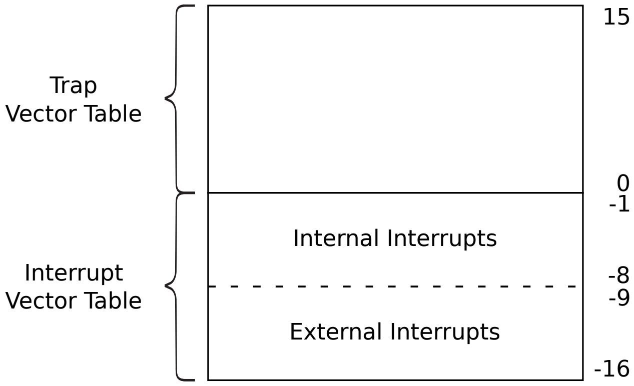


Figure 2.3: SPEAR2 Exception Vector Table

The exception vector table is populated through the ⟨stvec *reg, simm5*⟩ instruction, where *reg* denotes the name of a general purpose register, as shown in Table 2.1, to hold the exception vector, and *simm5* signifies a 5-bit signed immediate value which defines the index of the exception vector within the exception vector table. An example for the SPEAR2 32-bit architecture variant is presented in Table 2.2.

```
    .file "store_trap_handler.S"
#define ARG_VECTOR_INDEX 0

    .section .text
    .global store_trap_handler
    .type store_trap_handler, @function
store_trap_handler:
    ; load symbol of exception handler into r13
    ldhi  r13, 4th(trap_handler)
    ldliu r13, 3rd(trap_handler)
    sli   r13, 8
    ldliu r13, hi(trap_handler)
    sli   r13, 8
    ldliu r13, lo(trap_handler)
    stvec r13, ARG_VECTOR_INDEX ; store trap handler at vector table index ARG_VECTOR_INDEX
    rts ; return from subroutine
    .size store_trap_handler, .-store_trap_handler

    .type trap_handler, @function
trap_handler:
    rte ; return from exception
    .size trap_handler, .-trap_handler
```

Table 2.2: spear2_32-none-eabi-as Assembly Code Sample: Store Trap Handler

Traps Traps are triggered through the execution of the ⟨*trap uimm4*⟩ instruction, where *uimm4* signifies a 4-bit unsigned immediate value which defines the index of the trap vector within the trap vector table. The SPEAR2 trap vector table, after a reset, is shown in Table 2.3.

When a trap gets triggered, the control unit stores the *context*, of the processor and performs a jump to the trap handler. Each exception handler must be terminated by a *return-from-exception* (rte) *instruction* which restores the context again, thereby returning control to the interrupted program.

Trap Number	**Exception Vector Table Number**	**Address**
0	0	*undefined*
1	1	*undefined*
2	2	*undefined*
3	3	*undefined*
4	4	*undefined*
5	5	*undefined*
6	6	*undefined*
7	7	*undefined*
8	8	*undefined*
9	9	*undefined*
10	10	*undefined*
11	11	*undefined*
12	12	*undefined*
13	13	*undefined*
14	14	*undefined*
15	15	*undefined*

Table 2.3: SPEAR2 Trap Vector Table (After Reset)

Please bear in mind that, since the trap vectors are *undefined* after a reset, it is the responsibility of the programmer to provide dedicated trap vectors and trap handlers for every trap that gets eventually triggered by the user program.

Nested Traps The triggering of a trap from within a trap handler results in what is commonly referred to as a *nested trap*, and is something that requires careful handling by the programmer, as explained by the following scenario: assume that the handler of $trap_A$ triggers some other $trap_B$. Then, since $trap_A$ is being executed, the control unit must already have performed a context switch to $handler_A$, thereby storing the return address for $handler_A$ in the *return-from-exception* (rte) *register*. Consequently, when a context switch to $handler_B$ happens before $handler_A$ has returned, the return address for $handler_A$ will be set to the return address of $handler_B$. As a result, the processor will be caught within $handler_A$ and therefore be prevented from returning to its calling site.

A solution for the above problem, covering the SPEAR2 32-bit architecture variant, is presented in Table 2.4, where the contents of the *return registers* and the *saved custom status register* of the *processor control module*, as described in Section 2.1.2.2, are pushed onto the stack before triggering a nested trap and are popped off the stack and restored again once the trap handler has returned (lines 25f).

```
01:     .file "nesting_safe_trap_handler.S"
02: #define ARG_VECTOR_INDEX 0

03:     .section .text
04:     .global store_trap_handler
05:     .type store_trap_handler, @function
06: store_trap_handler:
07:     ; load symbol of trap handler into r13
08:     ldhi  r13, 4th(nesting_safe_trap_handler)
09:     ldliu r13, 3rd(nesting_safe_trap_handler)
10:     sli   r13, 8
11:     ldliu r13, hi(nesting_safe_trap_handler)
12:     sli   r13, 8
13:     ldliu r13, lo(nesting_safe_trap_handler)
14:     stvec r13, ARG_VECTOR_INDEX ; store trap handler at vector table index ARG_VECTOR_INDEX
15:     rts ; return from subroutine
16:     .size store_trap_handler, .-store_trap_handler

17:     .global trigger_trap_handler
18:     .type trigger_trap_handler, @function
19: trigger_trap_handler:
20:     trap ARG_VECTOR_INDEX ; trigger trap handler at vector table index ARG_VECTOR_INDEX
21:     rts ; return from subroutine
22:     .size trigger_trap_handler, .-trigger_trap_handler

23:     .type nesting_safe_trap_handler, @function
24: nesting_safe_trap_handler:
25:     stfpw_dec r14, -1          ; push contents of r14/rts onto stack
26:     stfpw_dec r15, -1          ; push contents of r15/rte onto stack
27:     jsr push_saved_status      ; push saved custom status register onto stack (defined elsewhere)
28:     trap (ARG_VECTOR_INDEX + 1) ; trigger another nested trap
29:     jsr pop_saved_status       ; pop saved custom status register from stack (defined elsewhere)
30:     ldfpw_inc r15, 0           ; pop r15/rte from stack
31:     ldfpw_inc r14, 0           ; pop r14/rts from stack
32:     rte ; return from exception
33:     .size nesting_safe_trap_handler, .-nesting_safe_trap_handler
```

Table 2.4: spear2_32-none-eabi-as Assembly Code Sample: Nesting-Safe Trap Handler

Interrupts Interrupts are triggered through an interrupt request on an associated interrupt line, which is assigned a unique interrupt number, by either internal or external hardware. The SPEAR2 interrupt vector table, after a reset, is shown in Table 2.5.

The *processor control module*, as described in Section 2.1.2.2, provides a *processor status register* and an *interrupt mask register*, which serve the global and selective enabling and disabling of interrupts:

- The *global interrupts enable* (GIE) *flag* of the *processor config register* globally enables all interrupts if the bit is set, and globally disables all interrupts otherwise.
- The *interrupt mask register* allows for the selective enabling and disabling of all maskable interrupts, where the bit numbers of the interrupt mask register correspond with the interrupt numbers, as provided in Table 2.5. An interrupt is masked if the corresponding bit number is set, and unmasked otherwise.

Whenever an interrupt request occurs on an interrupt line, independent of whether the associated interrupt is unmasked and interrupts are globally enabled, the control unit registers the event in the *interrupt protocol register* of the *processor control module.* An interrupt request is protocolled if the corresponding bit is set, which persists until the request is either handled or the respective bit gets cleared manually by the programmer. Please refer to Section 2.1.2.2 on how the interrupt protocol register is modified correctly. Again, the bit numbers of the interrupt protocol register correspond with the interrupt numbers, as provided in Table 2.5.

When an interrupt gets protocolled, provided that the interrupt is unmasked and that interrupts are globally enabled, the control unit looks up the *interrupt vector* according to the interrupt number in the interrupt vector table. Subsequently, the control unit stores the *context*, of the processor, clears the *GIE flag* in order to temporarily disable interrupts and jumps to the *interrupt handler.* Each exception handler must be terminated by a *return-from-exception* (rte) *instruction*, which sets the *GIE flag* and restores the *context* again, thereby returning control to the interrupted program. If the above precondition does not hold the programmer may still poll the interrupt protocol register, which becomes useful if interrupts shall be handled in a synchronous manner, as for example in real-time environments.

Every interrupt request that gets handled must be acknowledged by the programmer at the requesting peripheral by setting the *interrupt acknowledge* (INTA) *flag* in the *generic config register* of the *extension module interface*, as described in Section 2.3.1, early during the course of the interrupt handler. Interrupt lines may be shared by several interrupt sources, in which a polling scheme must be enforced to determine which of the peripherals issued the request: the *interrupt* (INT) *flag* of the *generic status byte* of the respective extension module is set if the module requested an interrupt, and clear otherwise.

	Interrupt Number	Exception Vector Table Number	Priority	Maskable	Masked	Source	Address
External Interrupts	0	-16	1 (highest)	N	N	-	*undefined*
	1	-15	16 (lowest)	Y	Y	-	*undefined*
	2	-14	15	Y	Y	-	*undefined*
	3	-13	14	Y	Y	-	*undefined*
	4	-12	13	Y	Y	-	*undefined*
	5	-11	12	Y	Y	-	*undefined*
	6	-10	11	Y	Y	-	*undefined*
	7	-9	10	Y	Y	-	*undefined*
Internal Interrupts	8	-8	9	Y	Y	AMBA shared memory	*undefined*
	9	-7	8	Y	Y	AMBA modules shared	*undefined*
	10	-6	7	Y	Y	AMBA communication error	*undefined*
	11	-5	6	Y	Y	miniUART/Breakpoint module	*undefined*
	12	-4	5	Y	Y	*reserved*	*undefined*
	13	-3	4	Y	Y	Programmer module	*undefined*
	14	-2	3	Y	Y	SysCtrl module	*undefined*
	15	-1	2	Y	Y	illop instruction	*undefined*

Table 2.5: SPEAR2 Interrupt Vector Table (After Reset)

Nested Interrupts The disruption of the execution of an interrupt handler for the processing of another, usually higher prioritized, interrupt results in what is commonly referred to as a *nested interrupt*. Besides the need to enable support for nested interrupts, it is something that requires careful handling by the programmer, as explained by the following scenario: assume that the handler of $interrupt_A$ is disrupted by some other $interrupt_B$. Then, since $interrupt_A$ is being handled the control unit must already have performed a context switch to $handler_A$, thereby storing the return address for $handler_A$ in the *return-from-exception register*. Consequently, when a context switch to $handler_B$ happens before $handler_A$ has returned, the return address for $handler_A$ will be set to the return address of $handler_B$. As a result, the processor will be caught within $handler_A$ and therefore be prevented from returning to the interrupted program.

A solution for the above problem, covering the SPEAR2 32-bit architecture variant, is presented in Table 2.6, where interrupts are globally re-enabled and the contents of the *return registers* and the *saved custom status register* of the *processor control module*, as described in Section 2.1.2.2, are pushed onto the stack at the beginning of the interrupt handler and are popped off the stack and restored again before returning (lines 20f).

```
01:     .file "nesting_safe_interrupt_handler.S"
02:     .section .text
03: #define ARG_VECTOR_INDEX 16

04:     .global store_interrupt_handler
05:     .type store_interrupt_handler, @function
06: store_interrupt_handler:
07:     ; load symbol of interrupt handler into r13
08:     ldhi  r13, 4th(nesting_safe_interrupt_handler)
09:     ldliu r13, 3rd(nesting_safe_interrupt_handler)
10:     sli   r13, 8
11:     ldliu r13, hi(nesting_safe_interrupt_handler)
12:     sli   r13, 8
13:     ldliu r13, lo(nesting_safe_interrupt_handler)
14:     stvec r13, ARG_VECTOR_INDEX ; store interrupt handler at vector table index ARG_VECTOR_INDEX
15:     rts ; return from subroutine
16:     .size store_interrupt_handler, .-store_interrupt_handler

17:     .type store_interrupt_handler, @function
18: nesting_safe_interrupt_handler:
19:     jsr set_INTA              ; acknowledge interrupt request (defined elsewhere)
20:     stfpw_dec r14, -1         ; push contents of r14/rts onto stack
21:     stfpw_dec r15, -1         ; push contents of r15/rte onto stack
22:     jsr push_saved_status     ; push saved custom status register onto stack (defined elsewhere)
23:     jsr set_GIE               ; globally re-enable interrupts (defined elsewhere)
24:     nop                       ; do something
25:     jsr pop_saved_status      ; pop saved custom status register from stack (defined elsewhere)
26:     ldfpw_inc r15, 0          ; pop r15/rte from stack
27:     ldfpw_inc r14, 0          ; pop r14/rte from stack
28:     rte ; return from exception
29:     .size nesting_safe_interrupt_handler, .-nesting_safe_interrupt_handler
```

Table 2.6: spear2_32-none-eabi-as Assembly Code Sample: Nesting-Safe Interrupt Handler

Finally, we conclude this section on interrupt handling with a review on the interrupt processing mechanism of the SPEAR2 architecture in the activity diagram in Figure 2.4.

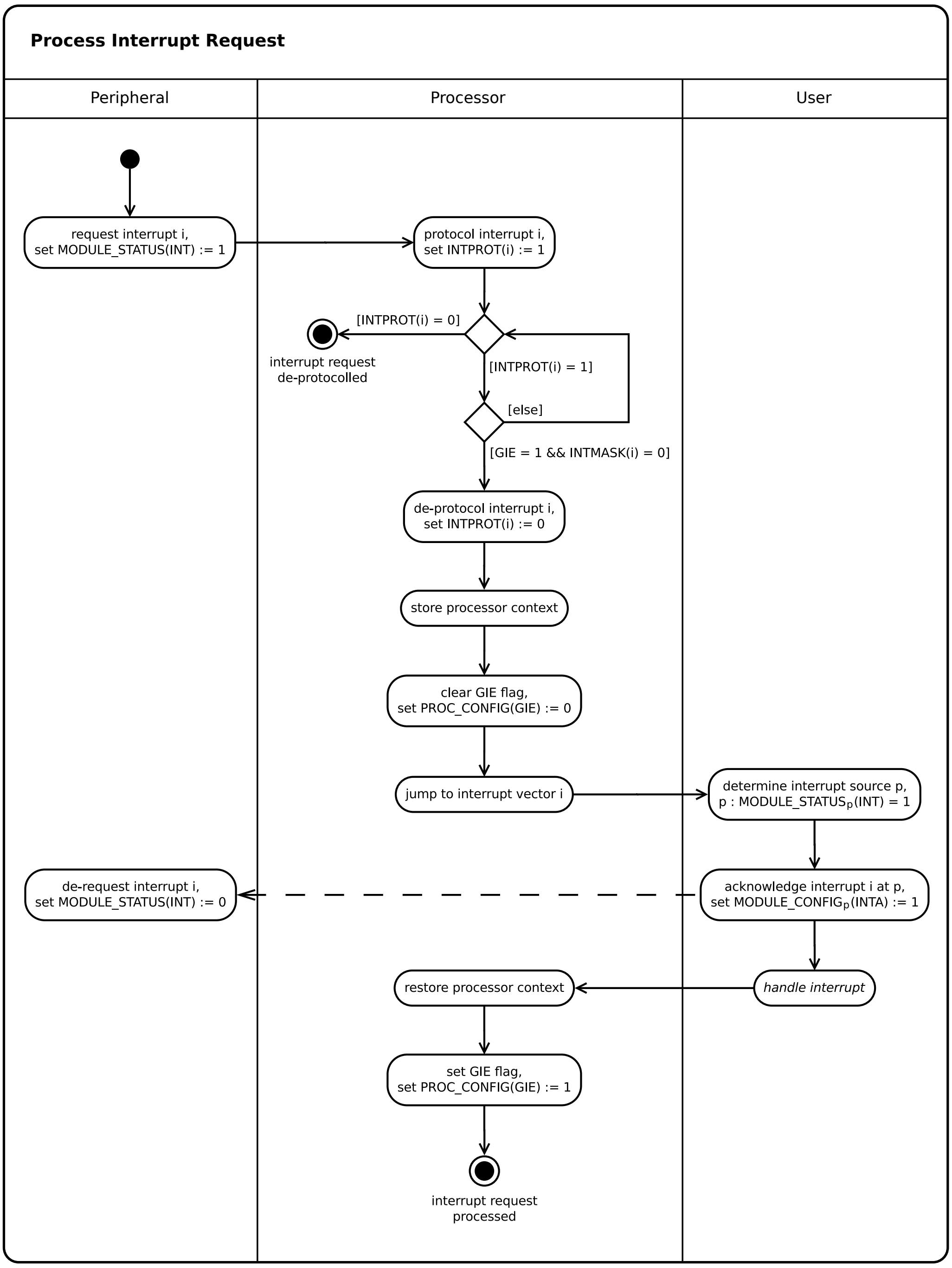

Process Interrupt Request
Peripheral
Processor
User
request interrupt i, set MODULE_STATUS(INT) := 1
protocol interrupt i, set INTPROT(i) := 1
[INTPROT(i) = 0]
interrupt request de-protocolled
[INTPROT(i) = 1]
[else]
[GIE = 1 && INTMASK(i) = 0]
de-protocol interrupt i, set INTPROT(i) := 0
store processor context
clear GIE flag, set PROC_CONFIG(GIE) := 0
jump to interrupt vector i
determine interrupt source p, p : MODULE_STATUS_p(INT) = 1
acknowledge interrupt i at p, set MODULE_CONFIG_p(INTA) := 1
de-request interrupt i, set MODULE_STATUS(INT) := 0
handle interrupt
restore processor context
set GIE flag, set PROC_CONFIG(GIE) := 1
interrupt request processed

Figure 2.4: Activity Diagram: SPEAR2 - Process Interrupt Request

2.1.2.2 System Control Modules

The term *system control modules* denotes a vital class of extension modules, as described in Section 2.3.1, which implement such basic functionality as *special purpose registers*, an *interrupt controller*, as well as a mechanism for *programming the instruction memory*, and are hence, despite being logically divided, an integral part of the SPEAR2 processor core that cannot be omitted.

As is the case with all extension modules, each system control module exports a set of special purpose registers which is mapped into a unique, statically assigned, segment in the data memory address space of the SPEAR2 architecture, so that its elements may be accessed conveniently via *load/store instructions*, as described in Appendix A.2.7. The base addresses of the system control modules for both SPEAR2 16-bit and 32-bit architecture variants are shown in Table 2.7.

Name	**(Base Address)$_{10}$**	**(Base Address)$_{16}$**		**Size** [bytes]
		16-bit	32-bit	
Processor Control Module	-32	0xFFE0	0xFFFFFFE0	32
Programmer Module	-64	0xFFC0	0xFFFFFFC0	32

Table 2.7: SPEAR2 System Control Modules - Memory Mappings

Processor Control Module The processor control module implements such essential special purpose register types as the *processor status register*, *processor config register*, *interrupt protocol register*, *interrupt mask register*, as well as *frame pointer registers* for the realization of a call stack. The interface of the processor control module for both SPEAR2 16-bit and 32-bit architecture variants is presented in Table 2.8.

Name	**(Base Offset)$_{16}$**	**Size** [bits]	
		16-bit	32-bit
Processor Status Register	+0x00	16	16
Processor Config Register	+0x02	16	16
Interrupt Protocol Register	+0x04	16	16
Interrupt Mask Register	+0x06	16	16
Frame Pointer W Register	+0x08	16	32
Frame Pointer X Register	+0x0C	16	32
Frame Pointer Y Register	+0x10	16	32
Frame Pointer Z Register	+0x14	16	32
Saved Custom Status Register	+0x18	8	8

Table 2.8: SPEAR2 Processor Control Module - Interface

Processor Status Register The processor status register is split into the *custom status byte* and the *generic status byte*, as depicted in Figure 2.5. The *custom status byte* provides the *condition code flags* of the arithmetic/logical operations of the ALU, and is described in Table 2.9. As for the *generic status byte*, refer to the definition of the *generic extension module interface* in Section 2.3.1.

Custom Status Byte								Generic Status Byte							
–	–	–	COND	ZERO	NEG	CARRY	OVER	LOOR	–	–	FSS	BUSY	ERR	RDY	INT
X	X	X	0	0	0	0	0	0	X	X	0	0	0	1	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 2.5: SPEAR2 Processor Status Register (After Reset)

Name	Function
OVER	The *overflow flag* indicates if the result of an operation has overflown.
CARRY	The *carry flag* indicates if an arithmetic carry has been generated.
NEG	The *negative flag* indicates if the result of an operation is negative.
ZERO	The *zero flag* indicates if the result of an operation equals zero.
COND	The *condition flag* indicates the result of a logical operation.

Table 2.9: SPEAR2 Processor Status Register - Custom Status Byte

Processor Config Register The processor config register is split into the *custom config byte* and the *generic config byte*, as depicted in Figure 2.6. The *custom status byte* provides the *global interrupts enable flag* and the *sleep flag*, as described in Table 2.10. As for the *generic config byte*, refer to the interface definition of the *generic extension module interface* in Section 2.3.1.

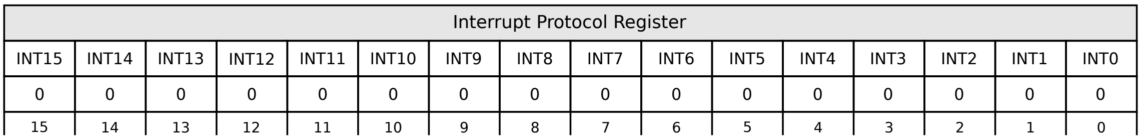
Custom Config Byte								Generic Config Byte							
GIE	SLEEP	–	–	–	–	–	–	LOOW	–	–	EFSS	OUTD	SRES	ID	INTA
0	0	X	X	X	X	X	X	0	X	X	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 2.6: SPEAR2 Processor Config Register (After Reset)

Name	Function
SLEEP	The *sleep flag* triggers the sleep mode of the processor, if set.
GIE	The *global interrupts enable flag* globally enables interrupts, if set.

Table 2.10: SPEAR2 Processor Config Register - Custom Config Byte

Interrupt Protocol Register The interrupt protocol register, as depicted in Figure 2.7, serves the registering of any incoming interrupt request by the interrupt controller, where the bit numbers of the interrupt protocol register correspond with the interrupt numbers, as provided in Table 2.5. An interrupt is protocolled if the corresponding bit number is set, which persists until the interrupt is either handled or the respective bit gets cleared by the programmer.

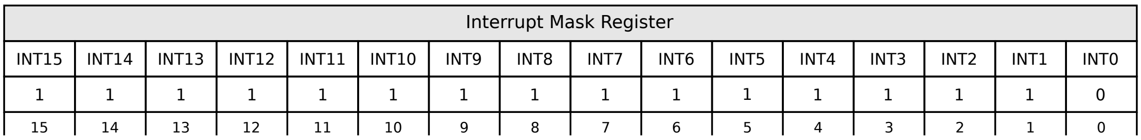


Interrupt Protocol Register															
INT15	INT14	INT13	INT12	INT11	INT10	INT9	INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 2.7: SPEAR2 Interrupt Protocol Register (After Reset)

Please note that any value that gets written to the interrupt protocol register by the user will be associated with the present value of that register in an *xor* operation. This allows for a more efficient implementation, compared to a read-modify-write operation by the user in which interrupts could get lost, as described in [Fle08, Section 5.5.10].

Interrupt Mask Register The interrupt mask register, as depicted in Figure 2.8, allows for the selective enabling and disabling of all maskable interrupts by the user, where the bit numbers of the interrupt mask register correspond with the interrupt numbers, as provided in Table 2.5. An interrupt is masked if the corresponding bit number is set, and unmasked otherwise. All maskable interrupts are masked per default.



Interrupt Mask Register															
INT15	INT14	INT13	INT12	INT11	INT10	INT9	INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 2.8: SPEAR2 Interrupt Mask Register (After Reset)

Frame Pointer Registers A frame pointer register, as depicted in Figure 2.9, is used to point to a location in the data memory address space of the SPEAR2 architecture. Together with an ensemble of *frame pointer instructions*, as described in Appendix A.2.7, the elementary push and pop operations of the stack memory model, which is essentially a first-in/last-out data structure, can effectively be emulated. The size of a single register depends on the configuration of the datapath unit, as described in Section 2.1.1.

Frame Pointer Register (High-order Half-Word)															
ADDR$_{31}$	ADDR$_{30}$	ADDR$_{29}$	ADDR$_{28}$	ADDR$_{27}$	ADDR$_{26}$	ADDR$_{25}$	ADDR$_{24}$	ADDR$_{23}$	ADDR$_{22}$	ADDR$_{21}$	ADDR$_{20}$	ADDR$_{19}$	ADDR$_{18}$	ADDR$_{17}$	ADDR$_{16}$
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

Frame Pointer Register (Low-order Half-Word)															
ADDR$_{15}$	ADDR$_{14}$	ADDR$_{13}$	ADDR$_{12}$	ADDR$_{11}$	ADDR$_{10}$	ADDR$_{9}$	ADDR$_{8}$	ADDR$_{7}$	ADDR$_{6}$	ADDR$_{5}$	ADDR$_{4}$	ADDR$_{3}$	ADDR$_{2}$	ADDR$_{1}$	ADDR$_{0}$
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 2.9: SPEAR2 Frame Pointer Register (After Reset)

The basic usage scenario of the stack, as depicted in Figure 2.10, is to store register contents before they are destroyed by subsequent processing tasks as well as to restore the original contents again once the processing tasks have finished. A designated pointer register, commonly referred to as the *stack pointer*, is automatically adjusted by the push and pop operations to always point to the top of the stack, thereby preventing the next stack operations from corrupting any previously stacked data [Yiu07, Chapter 3].

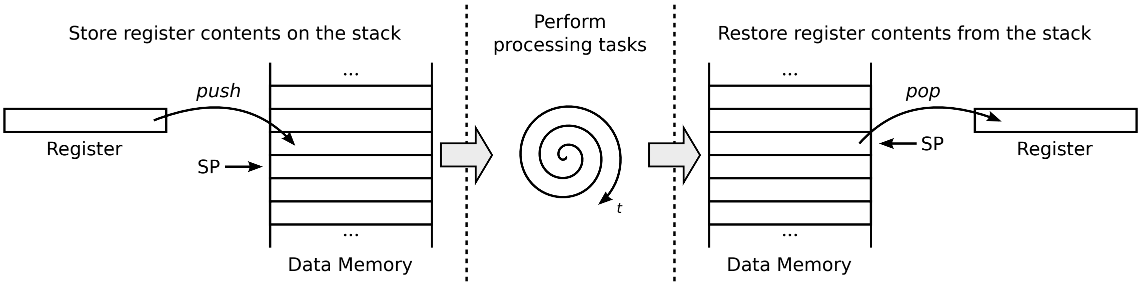


Figure 2.10: Basic Stack Usage Scenario

The emulations of the elementary stack operations for both falling and rising stacks, according to their direction of growth in memory, are shown in Table 2.11. The presence of a stack allows for the realization of a call stack, or run-time stack, a complicated topic of its own, which we will deal with extensively in conjunction with the procedure calling conventions of the SPEAR2 architecture in Section 3.2.3.

Stack		**Push**	**Pop**
Falling	↓	⟨stfpw_dec *reg, -1*⟩	⟨ldfpw_inc *reg, 0*⟩
Rising	↑	⟨stfpw_inc *reg, 1*⟩	⟨ldfpw_dec *reg, 0*⟩

Table 2.11: SPEAR2 Elementary Stack Operations

Saved Custom Status Register The saved custom status register is used by the processor to save and restore the custom status byte of the processor status register upon occurrence of an exception.

Programmer Module The programmer module provides a means for the programming of the instruction memory by the user and is used in combination with a communication module in order to download program code from a host platform to the executing target platform. The reason why there is no such module for programming the data memory lies in the built-in ability of the architecture in accessing the data memory through dedicated load/store instructions, as described in Appendix A.2.7. The interface of the programmer module for both SPEAR2 16-bit and 32-bit architecture variants is presented in Table 2.12.

Name	**(Base Offset)**$_{16}$	**Size** [bits]	
		16-bit	32-bit
Programmer Status Register	+0x00	16	16
Programmer Config Register	+0x02	16	16
Address Register	+0x04	16	32
Instruction Register	+0x08	16	16

Table 2.12: SPEAR2 Programmer Module - Interface

Programmer Status Register The programmer status register is split into the *custom status byte* and the *generic status byte*, as depicted in Figure 2.11. At present, the *custom status byte* is unused. As for the *generic status byte*, refer to the definition of the *generic extension module interface* in Section 2.3.1.

Custom Status Byte								Generic Status Byte							
–	–	–	–	–	–	–	–	LOOR	–	–	FSS	BUSY	ERR	RDY	INT
X	X	X	X	X	X	X	X	0	X	X	0	0	0	1	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 2.11: SPEAR2 Programmer Status Register (After Reset)

Programmer Config Register The programmer config register is split into the *custom config byte* and the *generic config byte*, as depicted in Figure 2.12. The *custom status byte* is described in Table 2.13. As for the *generic config byte*, refer to the definition of the *generic extension module interface* in Section 2.3.1.

Custom Config Byte								Generic Config Byte							
EXE	–	–	–	–	–	–	–	LOOW	–	–	EFSS	OUTD	SRES	ID	INTA
0	X	X	X	X	X	X	X	0	X	X	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 2.12: SPEAR2 Programmer Config Register (After Reset)

Name	Function
EXE	The *execute flag* triggers the programming of an instruction, if set.

Table 2.13: SPEAR2 Processor Config Register - Custom Config Byte

Address Register The address register, as depicted in Figure 2.13, is used to store the destination address of the instruction to be programmed into the instruction memory. The size of the address register depends on the configuration of the datapath unit, as described in Section 2.1.1.

Address Register (High-order Half-Word)															
ADDR$_{31}$	ADDR$_{30}$	ADDR$_{29}$	ADDR$_{28}$	ADDR$_{27}$	ADDR$_{26}$	ADDR$_{25}$	ADDR$_{24}$	ADDR$_{23}$	ADDR$_{22}$	ADDR$_{21}$	ADDR$_{20}$	ADDR$_{19}$	ADDR$_{18}$	ADDR$_{17}$	ADDR$_{16}$
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

Address Register (Low-order Half-Word)															
ADDR$_{15}$	ADDR$_{14}$	ADDR$_{13}$	ADDR$_{12}$	ADDR$_{11}$	ADDR$_{10}$	ADDR$_{9}$	ADDR$_{8}$	ADDR$_{7}$	ADDR$_{6}$	ADDR$_{5}$	ADDR$_{4}$	ADDR$_{3}$	ADDR$_{2}$	ADDR$_{1}$	ADDR$_{0}$
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 2.13: SPEAR2 Programmer Address Register (After Reset)

Please bear in mind that, since the instruction memory is conceptually laid out as an array of instructions and may therefore only be addressed instruction-wise, as described in Section 2.2.2, the value provided to the address register will be interpreted as an index to an element in the instruction array and must hence not be confused with a byte address.

Instruction Register The instruction register, as depicted in Figure 2.14, is used to store the instruction to be programmed into the instruction memory.

Instruction Register															
INS$_{15}$	INS$_{14}$	INS$_{13}$	INS$_{12}$	INS$_{11}$	INS$_{10}$	INS$_{9}$	INS$_{8}$	INS$_{7}$	INS$_{6}$	INS$_{5}$	INS$_{4}$	INS$_{3}$	INS$_{2}$	INS$_{1}$	INS$_{0}$
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 2.14: SPEAR2 Programmer Instruction Register (After Reset)

Finally, we conclude this section on instruction programming with the programmer module with an example, covering the SPEAR2 32-bit architecture variant, in Table 2.14.

```
    .file "program_instruction.S"
#define ADDR_PROG_MOD_BASE         -64
#define ADDR_PROG_MOD_ADDR_REG    ADDR_PROG_MOD_BASE + 0x4
#define ADDR_PROG_MOD_INSN_REG    ADDR_PROG_MOD_BASE + 0x8
#define ADDR_PROG_MOD_CONFIG_BYTE ADDR_PROG_MOD_BASE + 0x3
#define ARG_ADDR_VALUE 0
#define ARG_INSN_VALUE 0

.macro LOAD_ADDR_TO_REG addr reg
    ldhi  \reg, 4th(\addr)
    ldliu \reg, 3rd(\addr)
    sli   \reg, 8
    ldliu \reg, hi(\addr)
    sli   \reg, 8
    ldliu \reg, lo(\addr)
.endm

    .section .text
    .global program_instruction
    .type program_instruction, @function
program_instruction:
    LOAD_ADDR_TO_REG ADDR_PROG_MOD_ADDR_REG r0 ; load address of address register into r0
    LOAD_ADDR_TO_REG arg_address r1            ; load symbol of address argument into r1
    ldw r2, r1 ; load address argument into r2
    stw r2, r0 ; store address argument in r2 at address register address in r0

    LOAD_ADDR_TO_REG ADDR_PROG_MOD_INSN_REG r0 ; load address of instruction register into r0
    LOAD_ADDR_TO_REG arg_insn r1               ; load symbol of instruction argument into r1
    ldh r2, r1 ; load instruction argument into r2
    sth r1, r0 ; store instruction argument in r2 at instruction register address in r0

    LOAD_ADDR_TO_REG ADDR_PROG_MOD_CONFIG_BYTE r0 ; load address of custom config byte into r0
    ldliu r1, 0 ; load value 0 into r1
    bset r1, 7  ; set EXE bit at bit number 7 in r0
    stb r1, r0  ; store value in r1 at custom config byte address in r0
    rts ; return from subroutine
    .size program_instruction, .-program_instruction

    .section .data
    .p2align 2
arg_address:
    .int ARG_ADDR_VALUE

    .p2align 1
arg_insn:
    .short ARG_INSN_VALUE
```

Table 2.14: spear2_32-none-eabi-as Assembly Code Sample: Instruction Memory Programming

2.2 Memory

The memory of a computing system is a comparatively large but particularly slower type of internal or external type of storage for holding program instructions and data. For this reason, variables that are immediately or frequently used are held in the general purpose register file of the processor, a particularly fast type of internal storage, as described in Section 2.1.1.1.

2.2.1 Addressability

Memories are organized as two-dimensional arrays of *storage locations*, with each location being assigned a numeric address. Memory addresses typically start at 0 and are limited upwards by the *memory addressing capacity* of an architecture, as explained in Section 2.1.1. Recall that an architecture with a 32-bit address bus is able to address memory locations in the range of $[0 : 2^{32} - 1]$.

Although, in reality, memories are built from single bit cells, it would be a mere waste of memory addresses if each of these cells could be addressed individually. These days, a location in memory almost always refers to an 8-bit wide segment, a byte, while, in the past, locations ranged from as few as 1 bit up to 64 bits [Lev99, Section 2.2]. This significant observation leads to the definition of a *byte-addressable memory*, i.e., each byte in memory has a unique address [DH07, Section 6]. Therefore, the architecture from the above example could address up to 4 GB of data in memory.

Since many of the data that is handled by a modern processor, particularly program addresses, is significantly larger than a byte, modern architectures inevitably support the addressing of memory in quantities of integer multiples of a byte, so-called *multi-byte data*. These low-level data structures can give rise to surprising phenomena if not dealt with carefully, as explained below.

Alignment Accesses to memories are typically required to be aligned on a particular boundary: a memory access to an object of size s at address a is aligned iff $a \ mod \ s \equiv 0$, which is commonly referred to as *natural alignment* of an object. The reason why alignment matters is due to the fact that modern memories themselves are typically aligned on a word or multi-word boundary. Therefore, reading an object from an unaligned memory address would internally be converted into multiple aligned memory references and requires additional logic to assemble the requested data to present it to the user as a single object [Yiu07, Chapter 5]. Obviously, considering the slowness of memory accesses, unaligned memory transfers may cause severe performance penalties and are therefore inhibited by many modern computer architectures. This restriction is referred to as the *hard alignment constraint* [Dan05, Section 2.7].

Byte Order The representation of multi-byte data in a byte-addressable memory, i.e., the order in which the separate bytes are stored, is defined by the *endianness property* of an architecture that is best explained with the following example:

A 32-bit word is represented by a sequence of bits of distinct significance, with bit 0 being the *least significant bit (LSB)* and bit 31 being the *most significant bit (MSB)*, obviously as an analogy to the significances of the digits in a number. Assuming a byte-addressable memory, the given word is represented in memory by four bytes, where the *least significant byte (LSB$_8$)* is the byte that contains the LSB and *the most significant byte (MSB$_8$)* is the byte that contains the MSB. In the big-endian byte order scheme, the MSB$_8$ is located at the numerically lowest address with the less significant bytes following at higher addresses. Contrary, in the little-endian byte order scheme, the LSB$_8$ is located at the numerically lowest address with the more significant bytes following at higher addresses, as depicted in Figure 2.15. Please observe that the significances of the bits never change.

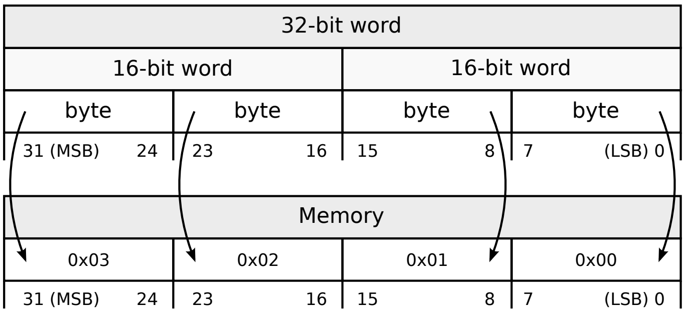


Figure 2.15: Little Endian Byte Order

The byte order employed by an architecture usually remains unnoticable. Nevertheless, byte-order related issues might arise when dealing with multi-byte data, as the following examples illustrate:

- *Accessing multi-byte data byte-wise*: iterating byte-wise over the character string *"Hello, World!"* in a little endian architecture yields *"!dlroW ,olleH"*, which is rather not what the programmer expects.
- *Exchanging data between computing systems*: while one might assume that byte-order related effects occur only if these systems employed different byte orders, it needs to be observed that modern computer networks employ a *network byte order*, which has to be converted to *host byte order* at the receiver [WRS03, Section 4.3].
- *Porting an application to another architecture*: applications which do not access the same multi-byte data differently and do not import binary data from elsewhere keep the byte order of the architecture invisible and can therefore be regarded portable [Swe06, Section 10.2].

2.2.2 Address Space

A memory address space is basically a set of memory addresses, as described above, whose dimensions are equally limited by the *memory addressing capacity* of an architecture. An architecture can have multiple address spaces, typically one for instructions and another one for data and provides independent address and data buses which allows instructions and data to be accessed simultaneously, as depicted in Figure 2.16, thereby improving considerably on the throughput of the system. This concept is commonly referred to as the *Harvard architecture*, named after Howard Aiken's and Grace Hopper's work on the Mark computers series at Harvard University [Shi07, Section 1.2.1]. Contrary, the *Von Neumann model* provides only a single address space for instructions and data [Noe05, Section 4.2].

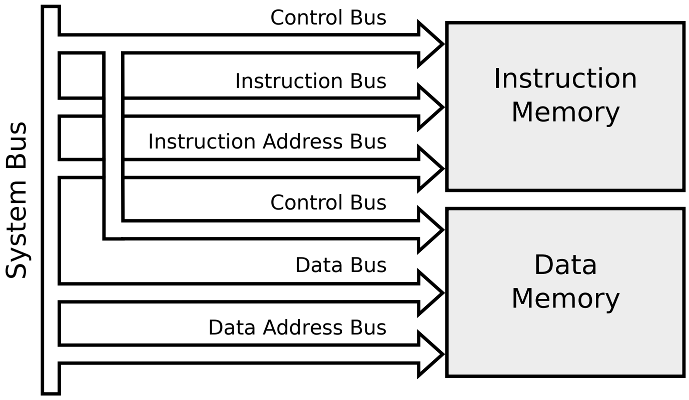


Figure 2.16: Harvard Architecture

Although memories are effectively organized as two-dimensional arrays of memory cells, to the programmer, the memory spaces of an architecture appear as large one-dimensional arrays of storage locations, typically bytes, which are referred to as *memory maps*. A memory map is divided into segments, where each segment defines a contiguous range of memory addresses, which starts at an associated *base address*, and that defines *offsets* for the elements which are located relative to that base address. Furthermore, a memory map clearly defines which segments are occupied by which components of the computing system.

Certainly, an architecture must provide means by which a multitude of address spaces and segments can be utilized. While the instruction memory address space is addressed via the *program counter* and dedicated *control transfer instructions* are provided to alter the, per default, sequential flow of control, the data memory address space is typically addressed via *load/store instructions*.

SPEAR2 ISA The SPEAR2 implements a Harvard architecture, so that there exists both an *instruction memory address space* and a *data memory address space*:

Instruction Memory Address Space The instruction memory address space of the SPEAR2 architecture comprises an *instruction memory* as well as an additional *boot memory* to hold the instructions of a *bootloader*, which is typically the first program to be executed after a hardware reset.

Since the SPEAR2 architecture employs a fixed length instruction coding of 16 bits, the instruction memory was deliberately designed to be *instruction-addressable* only, i.e., the smallest addressable unit in the instruction memory is an instruction. While this may seem cumbersome in the first place, it has the undeniable benefit of saving an extra bit in the immediate fields of the control transfer instructions as compared to a byte-addressable memory, which accounts for a gain in jump distance by a factor of 2. The instruction memory address space is solely addressed via the *program counter* and dedicated *control transfer instructions* are provided to alter the flow of control. Moreover, the instruction memory address space enforces a *little-endian* byte order.

The memory map of the instruction memory address space for both SPEAR2 16-bit and 32-bit architecture variants is depicted in Figure 2.17 and Table 2.15. The memory map represents the default settings of the current configurations of the target platforms in use. Please observe that the memory map is conceptually laid out using byte addresses.

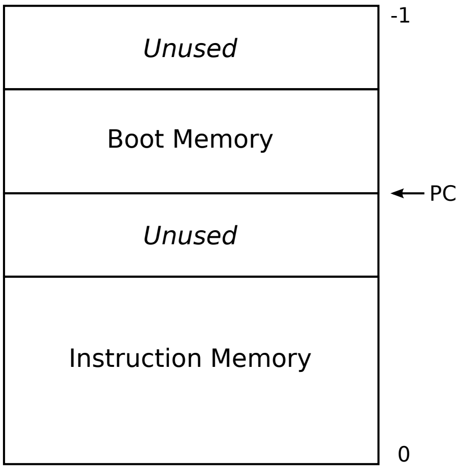


Figure 2.17: SPEAR2 Instruction Memory Address Space

Name	**Address Range** [bytes]		**Size** [bytes]	
	16-bit	32-bit	16-bit	32-bit
Instruction Memory	0x0000 : 0x7FFF	0x00000000 : 0x001FFFFF	0x8000	0x200000
Boot Memory	0x8000 : 0x8FFF	0x40000000 : 0x40000FFF	0x1000	0x1000

Table 2.15: SPEAR2 Instruction Memory Address Space Mappings (Defaults)

Data Memory Address Space The data memory address space of the SPEAR2 architecture comprises a *data memory*, as well as mappings to the *system control modules*, as described in Section 2.1.2.2, and the *I/O extension modules*, in Section 2.3.1.

The data memory address space is addressed via dedicated *load/store instructions*, as described in Appendix A.2.7, and requires all memory accesses to be *naturally aligned*. Moreover, the data memory address space enforces a *little-endian* byte order.

The memory map of the data memory address space for both SPEAR2 16-bit and 32-bit architecture variants is depicted in Figure 2.18 and Table 2.16. The memory map represents the default settings of the current configurations of the target platforms in use.

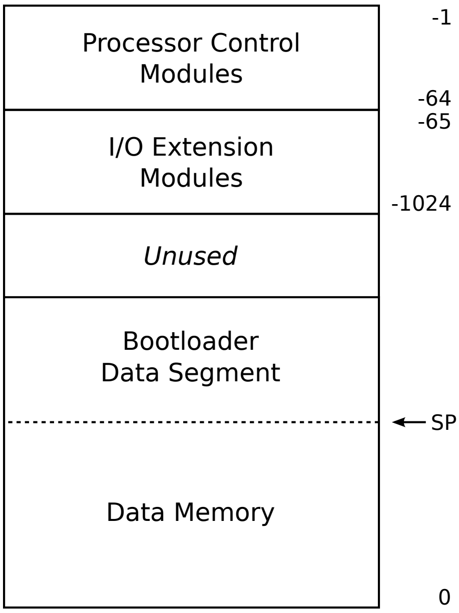


Figure 2.18: SPEAR2 Data Memory Address Space

Name	Address Range [bytes] 16-bit	Address Range [bytes] 32-bit	Size [bytes] 16-bit	Size [bytes] 32-bit
Data Memory	0x0000 : 0x7DFF	0x00000000 : 0x0007FDFF	0x7E00	0x7E00
Bootloader Data Segment	0x7E00 : 0x7FFF	0x0007FE00 : 0x0007FFFF	0x200	0x200
I/O Extension Modules	0xFC00 : 0xFFBF	0xFFFFFC00 : 0xFFFFFFBF	0x3C0	0x3C0
Processor Control Modules	0xFFC0 : 0xFFFF	0xFFFFFFC0 : 0xFFFFFFFF	0x40	0x40

Table 2.16: SPEAR2 Data Memory Address Space Mappings (Defaults)

2.3 I/O

Input/output (I/O) devices allow for the communication and, therefore, interaction of a computer system with the environment, i.e., the outside world. As a consequence, an I/O device requires access to the system bus, which is usually accomplished by an intermediate component, the I/O controller, that acts as an interface between the system and the I/O device, as depicted in Figure 2.19.

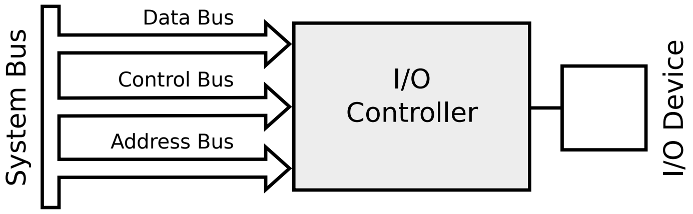


Figure 2.19: Basic I/O Controller Model

An I/O controller effectively abstracts away the characteristics of the associated device by providing the necessary commands and data which are required for the proper operation of the peripheral. On the system side, the I/O controller typically exports a set of registers: a read-only *status register* which represents the status, a *command register* which allows to command the operation, as well as additional *data registers* which serve the exchange of data between the system and the device under control. The registers exported by an I/O controller are commonly referred to as its *I/O ports*.

Memory-mapped vs. Isolated I/O With memory-mapped I/O, the ports of an I/O controller are mapped into a unique segment in the data memory address space of an architecture, so that its elements may be accessed conveniently via *load/store instructions*. A memory segment defines a contiguous range of memory addresses, which starts at an associated *base address*, and that defines *offsets* for the elements which are located relative to that base address. On the other hand, with isolated I/O, the ports of an I/O controller are mapped into a dedicated I/O address space that requires dedicated *I/O instructions* to be accessed. [Dan05, Section 2.6].

Interrupts vs. Polling Another matter that is closely related to I/O is the question of *input data acquisition*, i.e., the strategy by which data from a peripheral is gathered and provided to the system: I/O controllers typically occupy a physical interrupt line by which the device under control may attain the attention of the processor, as described in Section 2.1.2.1. Polling is a pull-based strategy by which the status of the I/O device is examined in a continuous or periodic fashion.

SPEAR2 ISA The SPEAR2 implements an extension module mechanism, as described in Section 2.3.1, as well as an extension for AMBA Modules, as described in Section 2.3.2, in order to provide I/O functionality.

2.3.1 Extension Modules

The extension modules mechanism is a proprietary means by which the built-in feature set of the SPEAR2 architecture can be extended. However, this mechanism is not necessarily restricted to input/output facilities, but may equally well be used to add to the processor core, or for the system to provide additional functionality efficiently in hardware which otherwise needed to be emulated in software.

Generic Extension Module Interface The extension modules mechanism provides a generic interface, which exports a set of special purpose registers that is mapped into a unique, statically assigned, segment in the data memory address space of the SPEAR2 architecture, so that its elements may be accessed conveniently via *load/store instructions*, as described in Appendix A.2.7. The generic extension module interface for both SPEAR2 16-bit and 32-bit architecture variants is depicted in Figure 2.20 and Table 2.17.

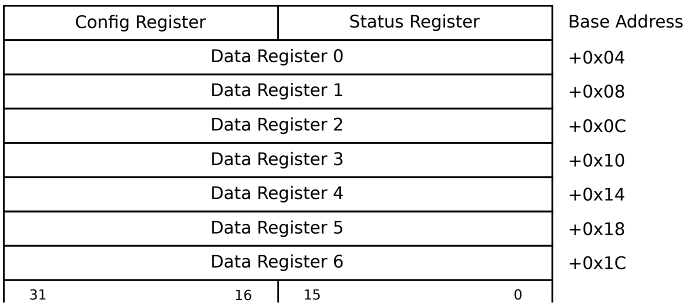
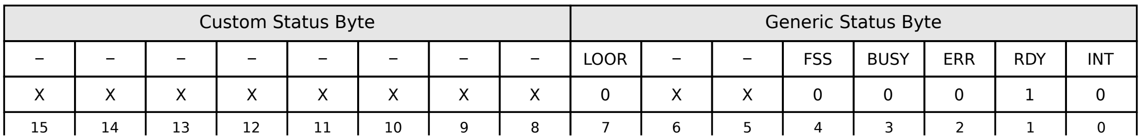


Figure 2.20: SPEAR2 Generic Extension Module Interface

Name	**(Base Offset)**$_{16}$	**Size** [bits]	
		16-bit	32-bit
Status Register	+0x00	16	16
Config Register	+0x02	16	16
Data Register 0	+0x04	16	32
Data Register 1	+0x08	16	32
Data Register 2	+0x0C	16	32
Data Register 3	+0x10	16	32
Data Register 4	+0x14	16	32
Data Register 5	+0x18	16	32
Data Register 6	+0x1C	16	32

Table 2.17: SPEAR2 Generic Extension Module Interface

Module Status Register The module status register is split into the *custom status byte* and the *generic status byte*, as depicted in Figure 2.21. The *generic status byte* defines a common extension module status interface, and is described in Table 2.18. The *custom status byte* is used to represent any additional, implementation-specific, module status.



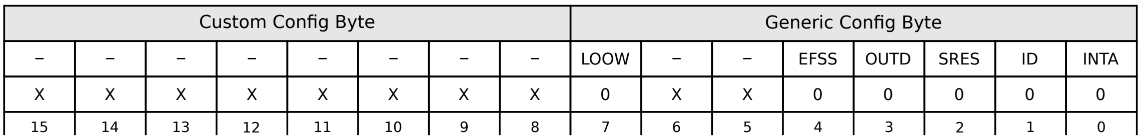
Custom Status Byte								Generic Status Byte							
–	–	–	–	–	–	–	–	LOOR	–	–	FSS	BUSY	ERR	RDY	INT
X	X	X	X	X	X	X	X	0	X	X	0	0	0	1	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 2.21: SPEAR2 Module Status Register (After Reset)

Name	Function
INT	The *interrupt flag* indicates if the module requested an interrupt.
RDY	The *ready flag* indicates if the module is ready for use.
ERR	The *error flag* indicates if the module raised an error.
BUSY	The *busy flag* indicates if the module is busy with work.
FSS	The *fail-safe-state flag* indicates if the module is in the fail-safe state.
LOOR	The *loop-read flag* indicates if the module is present by passing through the value written to the *LOOW flag* in the module config register.

Table 2.18: SPEAR2 Processor Status Register - Generic Status Byte

Module Config Register The module config register is split into the *custom config byte* and the *generic config byte*, as depicted in Figure 2.22. The *generic config byte* defines a common extension module command interface, and is described in Table 2.19. The *custom config byte* is used to configure any additional, module-specific, settings.



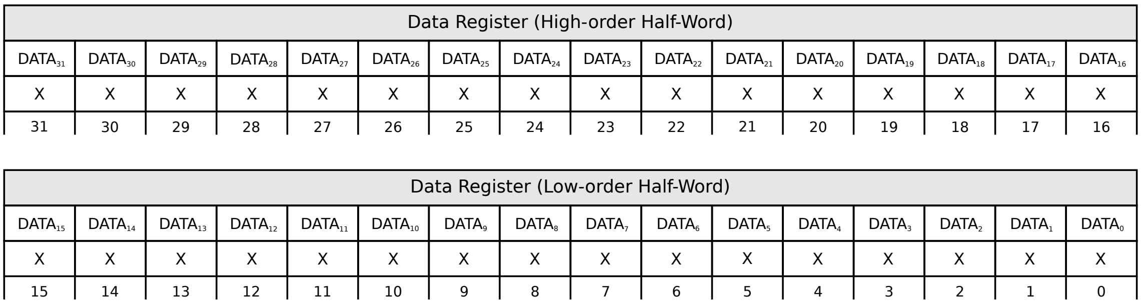
Custom Config Byte								Generic Config Byte							
–	–	–	–	–	–	–	–	LOOW	–	–	EFSS	OUTD	SRES	ID	INTA
X	X	X	X	X	X	X	X	0	X	X	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 2.22: SPEAR2 Module Config Register (After Reset)

Name	Function
INTA	The *interrupt acknowledge flag* acknowledges an interrupt request, if set.
ID	The *identify flag* places a manufacturer ID and version tag in the data register 0, if set.
SRES	The *soft-reset flag* triggers a soft reset at the module, if set.
OUTD	The *output-disable flag* decouples the module from its environment.
EFSS	The *enter-fail-safe-state flag* transists to the fail-safe state, if set.
LOOW	The *loop-write flag* is passed through to the *LOOR flag* in the module status register, if set and if the module is present.

Table 2.19: SPEAR2 Module Config Register - Generic Config Byte

Module Data Registers A data register, as depicted in Figure 2.23, is used to exchange data between the processor and the extension module.



Data Register (High-order Half-Word)															
$DATA_{31}$	$DATA_{30}$	$DATA_{29}$	$DATA_{28}$	$DATA_{27}$	$DATA_{26}$	$DATA_{25}$	$DATA_{24}$	$DATA_{23}$	$DATA_{22}$	$DATA_{21}$	$DATA_{20}$	$DATA_{19}$	$DATA_{18}$	$DATA_{17}$	$DATA_{16}$
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

Data Register (Low-order Half-Word)															
$DATA_{15}$	$DATA_{14}$	$DATA_{13}$	$DATA_{12}$	$DATA_{11}$	$DATA_{10}$	$DATA_{9}$	$DATA_{8}$	$DATA_{7}$	$DATA_{6}$	$DATA_{5}$	$DATA_{4}$	$DATA_{3}$	$DATA_{2}$	$DATA_{1}$	$DATA_{0}$
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 2.23: SPEAR2 Module Data Register (After Reset)

The base addresses of the I/O extension modules for both SPEAR2 16-bit and 32-bit architecture variants are shown in Table 2.20. Please refer to [Sei07] and [Del07] for further reference.

Name	**(Base Address)$_{10}$**	**(Base Address)$_{16}$**		**Size** [bytes]
		16-bit	32-bit	
miniUART Module	-128	0xFF80	0xFFFFFF80	32
Breakpoint Module	-160	0xFF60	0xFFFFFF60	32
Watchpoint Module	-192	0xFF40	0xFFFFFF40	32
Disp7seg Module	-288	0xFEE0	0xFFFFFEE0	32

Table 2.20: SPEAR2 I/O Extension Modules - Memory Mappings

2.3.2 AMBA Modules

AMBA Modules are an additional means to extend the built-in feature set of the SPEAR2 architecture. Please refer to [Mos08] for further information.

Chapter 3

Systems Software Layer

The systems software layer is the liaison between the hardware layer and the application layer, or what is commonly referred to as the "the software" of a computing system, in that it abstracts away from the complexity of the underlying hardware architecture, thereby providing a platform for running application software to the programmer.

In general, the systems software layer of a computing system comprises a high-level language compiler, assembler, linker, system libraries, debugger, as well as a bootloader, operating system and a variety of platform-dependent device drivers. Compiler, assembler, linker, system libraries and debugger are classified under the term *software development tools*, or *toolchain*, which, most notably, serve the translation of high-level source code into a debuggable and executable *binary program.* The term toolchain refers to the linked structure of the software development tools where the output of one tool serves as input to another. The basic systems software layer model is depicted in Figure 3.1.

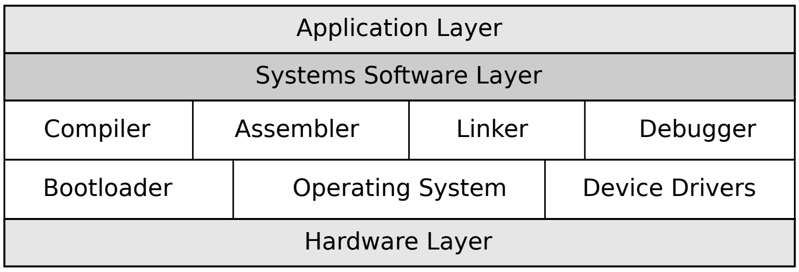


Figure 3.1: Basic Systems Software Layer Model

In this chapter we will shed a light on the functional principles of the most prominent members of the recent toolchain for the SPEAR2 architecture, followed by an elaboration on the *Embedded Application Binary Interface* (EABI), a collection of software standards to achieve interoperability and compatibility between binary programs in the absence of an operating system.

3.1 Software Development Tools

The collection of software development tools are an indispensable assortment of systems software for the integration of the software and the hardware of a computing system:

> *"Beyond the application code, there is a lot of software needed already in the development phase. The software tools are essential for the adoption of the processor, since nobody will use a processor without any tools, independently if it is a commercial processor or a proprietary special-purpose processor."* [Nur07, Chapter 4]

Although the composition of a toolchain may vary between architectures, the following software development tools are generally assumed to be provided:

- High-level language compiler, e.g., C, C++
- Assembler
- Linker
- System Libraries
- Debugger
- Simulator

Additionally, for a remaining class of *low-resource embedded computing systems* which do not allow for the use of an operating system, the following tools may equally be regarded part of a toolchain:

- Bootloader
- Device Drivers

The principal motivation behind the use of a toolchain is the achievement of a greater level of flexibility as well as a higher degree of independence from the underlying hardware architecture throughout the whole software development process: these days, software is almost invariably developed by programmers in *high-level programming languages*, such as C or C++. The term *high-level* refers to the essential characteristic of a programming language in that it provides language constructs which are closer to the human spoken language and thus semantically further away from the machine language that is native to the underlying hardware architecture. Therefore, high-level source code must further be translated to some low-level code of a *low-level programming language*, such as assembly language, whose instructions are typically merely symbolic representations of the *machine language instructions* which can effectively be executed by the processor.

The relationship between the various levels of software abstraction and hardware-dependency, as described above, are illustrated in Figure 3.2 [Dan05, Section 1.1].

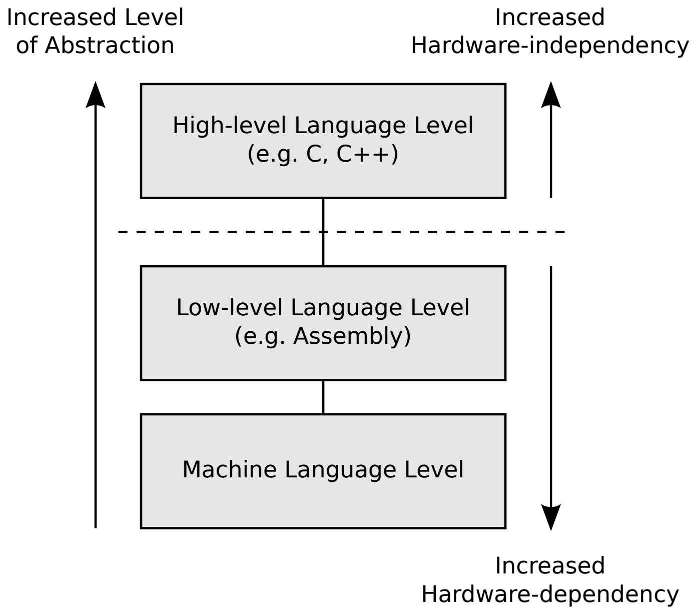

Increased Level of Abstraction
Increased Hardware-independency
High-level Language Level (e.g. C, C++)
Low-level Language Level (e.g. Assembly)
Machine Language Level
Increased Hardware-dependency

Figure 3.2: Levels of Software Abstraction

Another notable benefit of writing software in a high-level language is that, due to its hardware-independence, it can be run on virtually any architecture provided that there exists a toolchain by which it can be translated to a machine language representation that is native to the target architecture.

3.1.1 Embedded Software Development Process

The software development process for an embedded architecture differs from developing software for an average desktop system in that the development environment is typically made up of a *host computing system*, such as a PC, and the *target computing system*, i.e., the embedded system under development. The *host* communicates with the *target* via some transmission medium, such as a serial bus, or Ethernet, and often conveniently incorporates the software development tools for the target architecture under the hoods of an *integrated development environment* (IDE) in which the actual software is developed and from which it can be downloaded to the target, executed and be debugged with the simple click of a mouse-button. Because of these characteristics, the host is said to *manage* the development of the target [Noe05, Section 12.1].

The software development flow, i.e., the sequence by which a piece of software provided as high-level source code is translated into an executable machine language representation, exemplified by the toolchain for the SPEAR2 architecture, is illustrated in Figure 3.3.

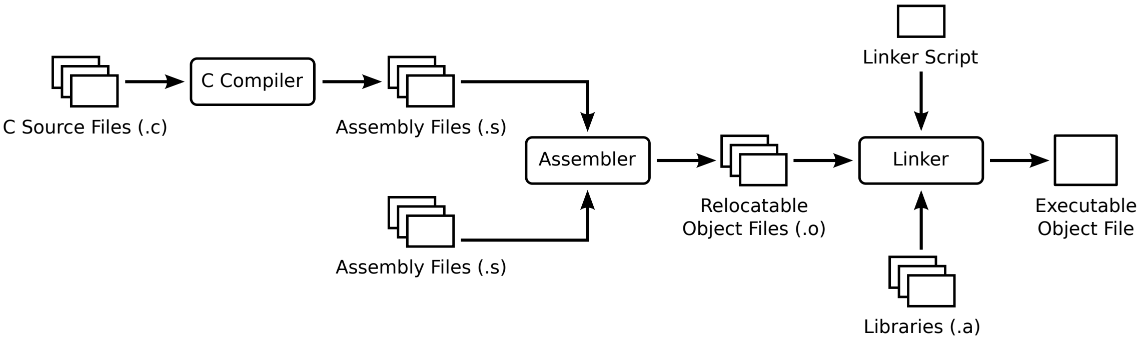


Figure 3.3: SPEAR2 Software Development Flow

After successful translation of the high-level source code, the executable object file may require further preparation, e.g., be brought into a simpler, more compact, format in order to be made recognizable for the bootloader, as illustrated in Figure 3.4.

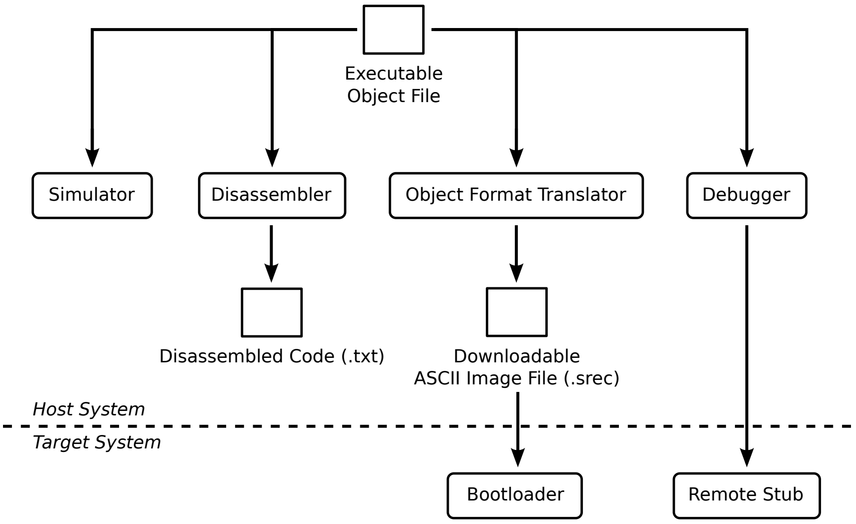


Figure 3.4: SPEAR2 Executable File Processing

Before we continue to describe the functional principles of the various stages of the software development flow in more detail, we shall give an overview of the most prominent members of the toolchain for the SPEAR2 architecture in Table 3.1.

Function	Artifact	
	16-bit	32-bit
C Compiler	spear2_16-none-eabi-gcc	spear2_32-none-eabi-gcc
Assembler	spear2_16-none-eabi-as	spear2_32-none-eabi-as
Linker	spear2_16-none-eabi-ld	spear2_32-none-eabi-ld
Debugger and Simulator	spear2_16-none-eabi-gdb	spear2_32-none-eabi-gdb
Disassembler	spear2_16-none-eabi-objdump	spear2_32-none-eabi-objdump
Object Format Translator	spear2_16-none-eabi-objcopy	spear2_32-none-eabi-objcopy
Bootloader	spear2_16-bootloader	spear2_32-bootloader
Remote Stub	spear2_16-gdb-stub	spear2_32-gdb-stub

Table 3.1: SPEAR2 Software Development Tools (Overview)

3.1.2 Compiler (gcc)

The GNU Compiler Collection (GCC)[1] is a compiler framework which supports a number of programming languages, such as C, C++, Java, Fortran and Ada, for a wide variety of architectures. The GCC processes one source file at a time, which has to run through a great number of processing stages. A simplified representation, which basically follows the model in [Mau08, Appendix C.1], is presented in Figure 3.5. At the time of writing, the *GCC* for the SPEAR2 architecture supports the C programming language only.

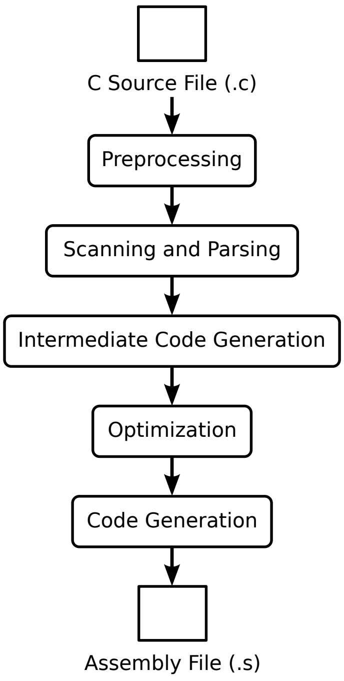


Figure 3.5: GCC Compilation Stages

[1] The GNU Compiler Collection, http://gcc.gnu.org

The *gcc* command is essentially a *compiler driver* in the sense that it actually invokes a sequence of other programs to take care of the *preprocessing*, *compiling*, *assembling* and *linking*, i.e., the translation, of high-level source code into an executable object file.

3.1.2.1 Preprocessing (cpp)

The *preprocessor* takes care of all preprocessing related actions, such as the inclusion of header files, the definition and expansion of macros and the replacement of code comments with white space characters, in order to generate a single, large, compilable input file from the source file (.c) and any included header files (.h).

3.1.2.2 Scanning and Parsing (cc1)

The *scanner* analyzes the input file on a character-per-character basis to look for *tokens*, i.e., atomic symbols, such as keywords, names, operators and punctuation characters, in order to generate a *token stream*. This token stream gets passed on to the *parser* which ensures syntactical correctness of the token sequence according to the grammatical rules of the respective programming language. The parser arranges the tokens in the form of an *abstract syntax tree* (AST), a parse tree which is still merely a syntactical representation of the input file [DAP07, Section 2.12].

3.1.2.3 Intermediate Code Generation (cc1)

During intermediate code generation, the *parse tree*, as provided by the parser, is converted into an intermediate representation known as the *register transfer language* (RTL). The RTL language actually looks similar to assembly language for a hypothetical processor, but with the essential property that it supports an infinite number of virtual processor registers. Later on, after optimization, these registers will be mapped to a finite set of real hardware registers or be spilled into memory otherwise.

3.1.2.4 Optimization (cc1)

The *optimizer* applies a variety of optimization schemes in order to make the intermediate code more efficient for execution on the target architecture. The optimization strategies include, amongst others, the simplification of arithmetic expressions, the elimination of code which cannot be reached by the program flow and the allocation of virtual registers into real hardware registers.

3.1.2.5 Code Generation (cc1)

Code generation concludes the compilation process during which the optimized code in the RTL language is translated to assembly code that includes assembly language instructions according to the instruction set specification of the target architecture. Each architecture supported by the GCC must provide a *machine description* which defines, amongst others, the rules by which assembly code is generated from intermediate code.

Please allow me to stress that the compiler itself does not generate an executable object file. Instead, the output of the compiler is a text file containing assembly language instructions and data, labels to name and identify particular quantities in memory, as well as assembler directives which instruct the *assembler* how the contents of the *assembly file* are translated into an *object file*. Additionally, the compiler may issue optional debugging information which allows for the analysis of a program within the scope of a debugger.

3.1.3 Assembler (as)

The GNU Assembler (gas), a member of the *GNU Binutils*[2], translates a single *assembly file* into a single *object file* containing, most notably, binary representations of assembly language instructions and data in an *object file format* that is supported by the target architecture. A simplified model of the assembler stages is illustrated in Figure 3.6.

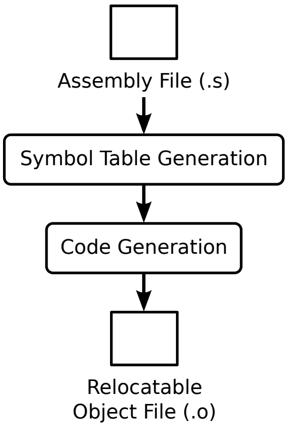


Figure 3.6: GAS Assembly Stages

3.1.3.1 Symbol Table Generation

During symbol table generation, the assembler scans the assembly file for labels, i.e., a symbol followed by a colon ':' which is used to name and identify a particular quantity in memory and which is suitable for use as an instruction operand, and stores the name of the symbol together with the size of the named quantity in the symbol table. Once the assembler has reached the end of the input file, the symbol table contains entries on all symbols which were defined within that file. The information contained within the symbol table aids during the subsequent code generation stage in which the symbols are mapped to concrete addresses within the output object file.

[2] The GNU Binutils, http://sourceware.org/binutils/

3.1.3.2 Code Generation

Once again, the assembler iterates the assembly file on a line-per-line basis and translates the individual statements, most notably assembly language instructions and assembler data storage directives, such as `.byte`, or `.int`, into binary machine code representations that are native to the target architecture. Assembly language instructions are assembled by concatenating the binary equivalents of their constituent parts, usually opcode and instruction operands, in the order specified by the computer architect in the instruction set specification. Binary data are organized within designated *sections* of an object file, such as `.text`, `.data`, `.rodata` or `.bss`, as instructed by dedicated assembler directives. We will refer to the concrete meanings of these sections in more detail when dealing with the *ELF Object File Format* in Section 3.1.6.

Programs almost invariably originate from a variety of source files, commonly referred to as *modules*, which serve the structuring and modularization of high-level source code into logically contained, isolated compilation units. Modules are allowed to interact with each others by declaring functions and data in a *global* scope which may thus be referenced from other modules. However, since modules are processed independently from each others by compilers and assemblers, references to functions and data being *external* to a module cannot be resolved as yet. When the assembler reads an assembly language instruction that references a symbol, the symbol is looked up in the symbol table and, provided it exists, the instruction operand in question is adapted to reflect the address of the symbol within the object file accordingly. Otherwise, an entry for the missing symbol will be made in the symbol table marking it as *undefined*. Such undefined references will be tried to be resolved later on in a global context amongst all object files by the linker.

Addresses of instructions and data within the object file are derived both from the entries in the symbol table and the dimensions associated with the data storage directives: by keeping track of the various symbol sizes the code generator can determine the address of the instruction or datum to be placed next within the object file. Addresses are assigned relative to the start address of the target section of a symbol with all sections starting at address `0x0`. Obviously, these start addresses can by no means represent the effective addresses of these sections when loaded into memory. In fact, since modules are processed in isolation, the assembler cannot know the effective addresses yet. Likewise, all absolute references to symbols are invalidated by the assembler by referring to address `0x0`. These references will be fixed in a process referred to as *relocation* once the effective addresses of the respective symbols were determined by the linker.

An object file cannot be executed as long as it contains any unresolved references to functions or data in another object file or as long as it contains any unrelocated symbol references and, until fixed by the linker, remains referred to as *relocatable*. The assembler greatly facilitates the job of the linker by providing important *bookkeeping information* in the form of the *symbol table* as well as additional *relocation information* within designated sections of the relocatable object file.

3.1.4 Linker (ld)

The GNU Linker (ld), a member of the *GNU Binutils*, translates the relocatable object files of a program into a single executable object file, as illustrated in Figure 3.7.

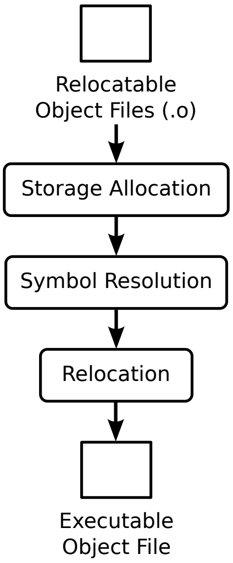


Figure 3.7: LD Linking Stages

The key benefit behind the concept of separate compilation and assembly of modules, as described in Section 3.1.3.2, is that changes to a single module do not affect the compilation and assembly status of any other modules and, as a consequence, do not require complete recompilation of an entire program. This demands for an additional processing step during which the sections of all relocatable object files, further referred to as the *input sections*, of a program are combined into *output sections* within a single executable object file, effective addresses are assigned and undefined symbol references are resolved in a global context.

3.1.4.1 Storage Allocation

The storage allocator collects the sizes of the input sections and, once it has determined their memory requirements, allocates a common storage in order to generate combined representations of the input sections: object files are processed in sequence, where each input section s of a module m_i is grouped with the corresponding section of module m_{i-1} in a corresponding output section s' within the allocated common storage, followed by any subsequent sections, as depicted in Figure 3.8.

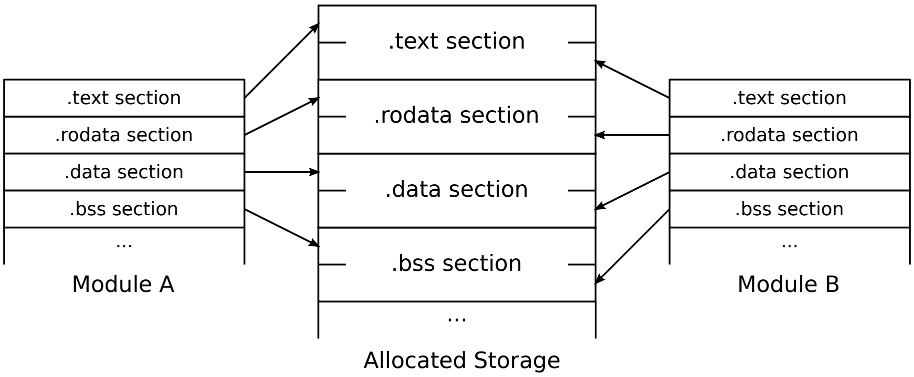


Figure 3.8: Linker Storage Allocation

The effective address of an input section within the common storage is derived from the sizes of the input sections allocated before that section. However, linkers typically also provide for more fine-grained control over the storage allocation process either through command line switches, or through the use of linker control scripts, or simply *linker scripts*.

Linker Scripts The *GNU ld* provides a scripting language for fine-grained control over the linking process by which the arrangement of output sections within the executable object file as well as their concrete base addresses can be specified with great flexibility. This is particularly useful for mapping the contents of binary programs to the usually non-contiguous process address spaces of embedded architectures, into which often a multitude of distinct physical memories is packed, as described in Section 3.2.4.

A simple example of a linker script is given in Table 3.2, in which the input sections on the right side are assigned to the output sections on the left side of the '`:`' operator. Output sections are allocated in the order specified in the linker script, where the `.rodata` section is explicitly instructed to start at address `0x0` with the `.data` and `.bss` sections following immediately afterwards. The `.text` section is mapped to address `0x80000000`.

```
SECTIONS
{
    . = 0x0;
    .rodata : { *(.rodata) }
    .data   : { *(.data) }
    .bss    : { *(.bss) }

    . = 0x80000000;
    .text : { *(.text) }
}
```

Table 3.2: GNU ld - Linker Script Sample

Virtual Memory Addresses vs. Load Memory Addresses The base addresses assigned to the output sections are commonly referred to as *virtual memory addresses*, since they refer to locations of the process image of a binary program in virtual memory. However, in systems with multiple virtual address spaces, as for example with separate address spaces for instructions and data, a memory address value alone apparently cannot bear enough information to be safely assignable to one of the available address spaces. This poses serious problems to those members of the system software layer which load the contents of a binary program into memory. For this very reason, the *GNU ld* allows for the organization of many address spaces of *virtual memory addresses* (VMAs) into a single address space of *load memory addresses* (LMAs), as illustrated in Figure 3.19, by dedicated linker script commands.

Please note that LMAs are nothing more than alternative base addresses of the output sections of an executable object file and will only be interpreted by those members of the system software layer which require this sort of information, such as *loaders* and *debuggers*. The instructions within a program will always refer to the VMAs.

3.1.4.2 Symbol Resolution

During symbol resolution the linker extracts the symbol tables from all relocatable object files in order to derive a global symbol table which contains entries for all symbols defined and referenced in any input file. Having built the global symbol table, the linker is finally able to identify any unresolved symbol references.

3.1.4.3 Relocation

The term relocation refers to a central aspect of the linking process by which any absolute references to symbols within machine code instructions are patched to reflect the final locations of the symbols within their respective output sections.

Relocatable object files contain designated sections which hold relocation information to aid the linker in identifying the instructions which require relocation. Each of these sections contains a collection of *relocation entries* where an entry contains: an index to the referenced symbol in the symbol table, an offset within the `.text` section to the relocatable instruction plus a relocation type. The *relocation type* refers to a data structure within the linker which defines how address operands are extracted from an instruction, how the effective symbol address is calculated and how the effective address is properly patched back into the instruction. Please refer to Section 3.41 for the relocation types of the SPEAR2 architecture in the absence of an operating system.

Finally, the linker sequentially processes all relocation entries by patching the address operands of all relocatable instructions with the effective symbol addresses. As soon as all relocation entries are processed, the linker disposes of all relocation information and produces an *executable object file*.

3.1.5 Libraries

Libraries are collections of relocatable object files, containing instructions and data, which are included into an executable object file during the linking process, as required.

The systems software layer of an architecture typically contains a set of *system libraries* which provide low-level services that hardly any program can live without. System libraries are automatically linked into a program when needed. With the SPEAR2 architecture, as with many other architectures alike, the system libraries comprise:

3.1.5.1 GCC Runtime Library (libgcc)

The *GCC Runtime Library*[3] handles arithmetic operations on integers, floating point and fixed point data types which the target processor cannot handle directly. As an example, integer multiplication and division as well as all operations on floating point and fixed point data types are emulated in software by the `libgcc.a` for the SPEAR2 architecture.

3.1.5.2 C Library (libc)

The *newlib C Library*[4] is a collection of functions which provide low-level services, such as string manipulation, dynamic memory allocation and input/output, in the C programming language in `libc.a` for the SPEAR2 architecture.

C Runtime (crt0) The *C Runtime* is part of the *C Standard Library* and contains initialization code of a binary program in assembly programming language. As an example, the *crt0* typically includes the initialization of the stack pointer and frame pointer registers, as well as the clearing of the *.bss section* in memory, as described in Section 3.1.6, in `crt0.o`. A configuration in the linker script forces the initialization code to always be placed first within the executable object file, followed by the actual program.

Board Support Package The *newlib* depends on the availableness of a variety of glue routines, referred to as *stubs*, which are used to connect the *libc* to the *system call interface* of an operating system in order to access functionality from the underlying hardware. As an example, the functions `printf`, `fprintf`, `sprintf` and `snprintf` of the *libc* depend on the stubs `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk` and `write`.

Besides these stubs, the *board support package* (BSP) for the SPEAR2 architecture in the absence of an operating system, encapsulated in the library `libnosys.a`, additionally contains a driver for the *miniUART extension module*, as described in [Sei07], to which any file operations on `stdin`, `stdout` and `stderr` are redirected. Furthermore, a number of machine-dependent header files are delivered together with `libnosys.a` in order to facilitate programming for the SPEAR2 architecture variants, as presented below.

[3] The GCC Runtime Library, http://gcc.gnu.org/onlinedocs/gccint/Libgcc.html

[4] The Red Hat newlib C Library Documentation, http://sourceware.org/newlib/libc.html

Machine Register Descriptions (modules.h) The machine register descriptions header file names the locations and bit numbers of the special purpose registers which are mapped into the data memory address space of the SPEAR2 architecture. At the time of writing, the *System Control Modules*, as presented in Section 2.1.2.2, as well as a variety of *Extension Modules*, as described in Section 2.3.1, are included in <`machine/modules.h`>.

An excerpt of a typical register description for the SPEAR2 32-bit architecture variant is given in Table 3.3, followed by a simple usage example in Table 3.4.

```
#ifndef __SPEAR2_32_MODULES_H__
#define __SPEAR2_32_MODULES_H__

#include <stdint.h>

/* Processor Control Module register definitions */
#define PROC_CTRL_SIZE    24
#define PROC_CTRL_BADDR   ((uint32_t)-32)
...
#define PROC_CTRL_STATUS_C_BOFF    1
#define PROC_CTRL_STATUS_C_BADDR   PROC_CTRL_BADDR + PROC_CTRL_STATUS_C_BOFF
#define PROC_CTRL_STATUS_C         (*(volatile uint8_t  *const) (PROC_CTRL_STATUS_C_BADDR))
#define PROC_CTRL_STATUS_C_COND    0x5
#define PROC_CTRL_STATUS_C_ZERO    0x4
#define PROC_CTRL_STATUS_C_NEG     0x3
#define PROC_CTRL_STATUS_C_CARRY   0x2
#define PROC_CTRL_STATUS_C_OVER    0x1
...
#define PROC_CTRL_CONFIG_C_BOFF    3
#define PROC_CTRL_CONFIG_C_BADDR   PROC_CTRL_BADDR + PROC_CTRL_CONFIG_C_BOFF
#define PROC_CTRL_CONFIG_C         (*(volatile uint8_t  *const) (PROC_CTRL_CONFIG_C_BADDR))
#define PROC_CTRL_CONFIG_C_GIE     0x7
#define PROC_CTRL_CONFIG_C_SLEEP   0x6
...
```

Table 3.3: SPEAR2 Register Descriptions in <`machine/modules.h`>

```
#include <machine/modules.h>

int main (int argc, char *argv[])
{
    /* Globally enable all interrupts */
    PROC_CTRL_CONFIG_C |= (1 << PROC_CTRL_CONFIG_C_GIE);
    ...
    /* Check the condition flag status */
    if (PROC_CTRL_STATUS_C & (1 << PROC_CTRL_STATUS_C_COND))
    {
        /* Send the processor to sleep mode */
        PROC_CTRL_CONFIG_C |= (1 << PROC_CTRL_CONFIG_C_SLEEP);
    }
    ...
    return 0;
}
```

Table 3.4: spear2_32-none-eabi-gcc C Code Sample - Register Description Usage

Exception Macros (interrupts.h) The exception macros header file provides a variety of preprocessor macros which are primarily related to the handling of interrupts for the SPEAR2 architecture. At the time of writing, the following macros are provided in <`machine/interrupts.h`>, as described in Table 3.5.

Name	**Description**
`sleep_mode()`	Sends the processor to sleep mode
`sei()`	Globally enables all interrupts
`cli()`	Globally disables all interrupts
`register_trap(func, num)`	Registers `func` as trap handler number `num` $\in [0; 15]$
`register_interrupt(func, num)`	Registers `func` as interrupt handler number `num` $\in [0; 15]$
`maski(num)`	Masks interrupt number `num` $\in [0; 15]$
`umaski(num)`	Unmasks interrupt number `num` $\in [0; 15]$
`maskl(nums)`	Masks all interrupt numbers whose bits are set in `nums`
`umaskl(nums)`	Unmasks all interrupt numbers whose bits are set in `nums`
`proti(num)`	Protocols interrupt number `num` $\in [0; 15]$
`uproti(num)`	Un-protocols interrupt number `num` $\in [0; 15]$
`protl(nums)`	Protocols all interrupt numbers whose bits are set in `nums`
`uprotl(nums)`	Un-protocols all interrupt numbers whose bits are set in `nums`

Table 3.5: SPEAR2 Exception Macros in <`machine/interrupts.h`>

The *GCC* for the SPEAR2 architecture allows for the definition of interrupt handler code through dedicated *function attributes*[5] provided as part of the function declaration of a designated interrupt handler routine. Interrupt handlers may be declared *disruptible* in order to allow for *nested interrupts*, a topic which we have dealt with in Section 2.1.2.1. The supported function attributes are described in Table 3.6, followed by a simple usage example in Table 3.7.

Name	**Description**
`interrupt`	Declares a disruptible interrupt handler (nesting-safe)
`signal`	Declares a non-disruptible interrupt handler

Table 3.6: spear2-gcc Function Attributes

```
void isr() __attribute__ ((interrupt));
void isr()
{
      ...
}
```

Table 3.7: spear2-gcc C Sample Code: Interrupt Handler Declaration

[5] GCC Function Attributes, http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html

Finally, we conclude this section with an overview on interrupt programming for the SPEAR2 architecture in Table 3.8. Observe how interrupt programming, as described in Section 2.1.2.1, has been facilitated considerably with the advent of a supporting toolchain, as opposed to the assembly code sample in Table 2.6.

```
#include <machine/modules.h>
#include <machine/interrupts.h>

#define INTERRUPT_NUM 0

void isr() __attribute__ ((interrupt));
void isr()
{
    /* Acknowledge interrupt at source */
    ...
    /* Handle the interrupt */
    ...
}

int main (int argc, char *argv[])
{
    /* Register interrupt */
    register_interrupt(isr, INTERRUPT_NUM);

    /* Unmask interrupt */
    umaski(INTERRUPT_NUM);
    ...
    /* Globally enable interrupts */
    sei();
    ...
    return 0;
}
```

Table 3.8: spear2-gcc C Code Sample: Interrupt Programming

3.1.5.3 C Math Library (libm)

The *newlib C Math Library*[6] is a collection of mathematical functions, such as `log`, `sin` and `sqrt`, in `libm.a` for the SPEAR2 architecture. Although math libraries are typically bundled with a C library distribution, they are not regarded part of the systems libraries and are therefore not automatically linked into a program. Consequently, when a program uses functionality from this library by including `<math.h>`, the GCC must be instructed to explicitly link against this library via `-lm`.

[6] The RedHat newlib C Math Library, http://sourceware.org/newlib/libm.html

3.1.6 ELF Object File Format

Object file formats serve the purpose of organizing the contents of binary programs in a standardized way by, at the same time, being flexible enough to accommodate any kind of information that needs to be referenced by assemblers, linkers, loaders and debuggers. The *Executable and Linking Format* (ELF)[7] is the standard object file format of the Linux and Unix operating systems. Because the ELF format is of considerable complexity, we will only give a simplified presentation in this place.

The ELF specification accommodates three different types of object files within a single file format: *relocatable*, *executable* and *shared object*: *relocatable object files* are generated by assemblers from an assembly file and are suitable for being processed by a linker, which translates them into an *executable object file* or *shared object file. Executable object files* are suitable to be loaded into memory and executed by a loader. *Shared object files*, which are used in dynamic linking, require extensive support by an operating system and, since not being available for the SPEAR2 architecture at the time of writing, are not further elaborated on in this document for the sake of simplicity.

ELF files have a rather unusual dual nature in that they are treated as collections of *sections* by assemblers and linkers, while loaders treat them as collections of *segments.* Sections are containers for a single type of information, such as instructions, data, symbol table, relocation information, or debugging information, and are described by a *section header table.* Contrary, segments typically comprise several sections and are described by a *program header table.* As an example, a loadable read-only segment typically contains instructions and read-only data. The *linking view* and the *execution view* of an ELF file are illustrated in Figure 3.9 [Lev99, Section 3.7].

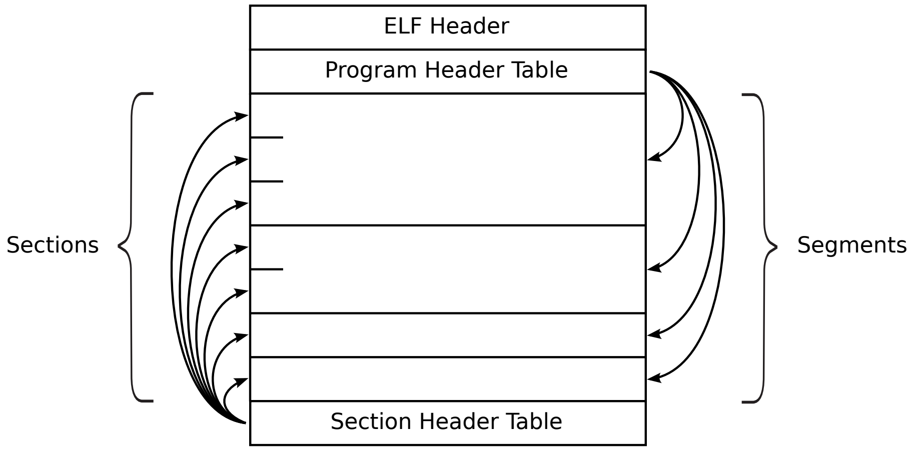


Figure 3.9: ELF File - Linking View vs. Execution View

[7] Executable and Linking Format (ELF) Specification, http://refspecs.freestandards.org/elf/elf.pdf

ELF Header The ELF header describes the organization of an ELF file and is located at the beginning. Within its initial bytes, the header contains an *identification pattern* which identifies the file as an ELF file and specifies how to the file contents shall be interpreted by an arbitrary architecture, as for example by indicating the byte order in which information is encoded within the file, in a machine-independent form. Additionally, the header specifies, amongst others, the *object file type*, the *required hardware architecture*, as well as the offsets and sizes of the *program header table* and the *section header table* within the object file.

Relocatable Object Files The contents of relocatable object files are described by the section header entries of the *section header table*. A section header defines various attributes, including a *name*, *type*, *flags*, as well as the *offset* and *size* of a section within the object file. In this section we will describe the attributes at an appropriate level of detail which is helpful in gaining a basic understanding of how the contents of relocatable object files are to be interpreted:

Section Name The section name, such as `.text`, `.data`, `.rodata`, or `.bss`, identifies a section within the section header table.

Section Type The section type defines the semantics of the information contained within a section. The most important types are described in Table 3.9.

Type	**Description**
`PROGBITS`	The section contains program contents, such as instructions, data and debugging information, for which storage is allocated within the object file.
`NOBITS`	The section contains program contents for which no storage is allocated within the object file. This type marks the *.bss section*, for which storage is allocated when the program is loaded.
`SYMTAB`	The section contains a symbol table.
`RELA`	The section contains relocation information.

Table 3.9: ELF Section Types

Section Flags The section flags define miscellaneous attributes of a section, of which the most prominent ones are described in Table 3.10.

Flag	**Description**
`ALLOC`	The section occupies space when the program is loaded into memory.
`WRITE`	The section contains data that should be writable during program execution.
`EXECINSTR`	The section contains executable machine code instructions.

Table 3.10: ELF Section Flags

An assortment of sections of great significance is presented in Table 3.11, followed by a typical layout of a relocatable object file in Figure 3.10.

Type	Type	Flags	Description
`.text`	`PROGBITS`	`ALLOC`, `EXECINST`	The section contains instructions.
`.data`	`PROGBITS`	`ALLOC`, `WRITE`	The section contains initialized data.
`.rodata`	`PROGBITS`	`ALLOC`	The section contains read-only data.
`.bss`	`NOBITS`	`ALLOC`, `WRITE`	The section contains uninitialized data. As a rule, the system initializes the data with zeros when the program is loaded.
`.symtab`	`SYMTAB`	-	The section contains a symbol table.
`.rela.text`	`RELA`	-	The section contains relocation information for the `.text` section.
`.rela.data`	`RELA`	-	The section contains relocation information for the `.data` section.
`.rela.rodata`	`RELA`	-	The section contains relocation information for the `.rodata` section.
`.rela.bss`	`RELA`	-	The section contains relocation information for the `.bss` section.

Table 3.11: ELF Sections

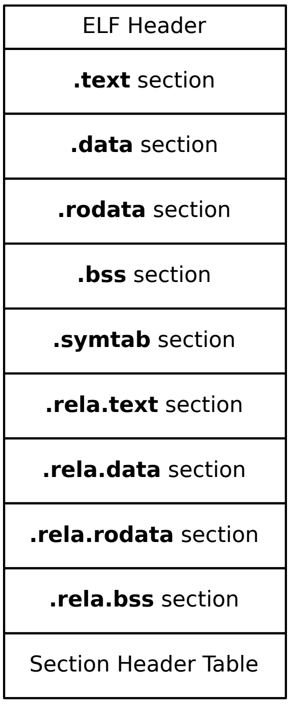


Figure 3.10: Sample ELF Relocatable Object File

Executable Object Files The contents of executable object files are described by the program header entries of the *program header table*. A program header defines various attributes, including a *type*, *flags*, the *offset* and *size* of a segment within the object file, as well as the *address* and *size* of a segment in memory. The memory address of a segment is specified both as *virtual memory address* and as *load memory address*, as described in Section 3.1.4.1. In this section we will describe the attributes at an appropriate level of detail which is helpful in gaining a basic understanding of how the contents of the loadable segments of executable object files are to be interpreted:

Segment Type The segment type defines the semantics of the information contained within a segment. The most important types are described in Table 3.9.

Type	Description
`NULL`	The segment is unused, the program header entry will be ignored.
`LOAD`	The segment is loadable into memory.

Table 3.12: ELF Segment Types

Segment Flags The segment flags define the permission attributes of a segment, of which the most important ones are described in Table 3.13.

Flag	Description
`R`	The loadable segment is readable during program execution.
`W`	The loadable segment is writable during program execution.
`X`	The loadable segment is executable during program execution.

Table 3.13: ELF Segment Flags

In order that a binary program can be executed it must at least have a single loadable and executable segment. Typically, an executable object file comprises a small number of segments only, such as a read-only segment for instructions and read-only data, as well as a read/write segment for read/write data [Lev99, Section 3.7].

3.1.7 DWARF Debugging Information Format

Debugging information establishes the relationship between the source code and the binary program in a manner that is suitable for inclusion within object files by at the same time being efficient enough to be processed with a debugger. The *Debugging With Attributed Records Formats* (DWARF)[8] is the standard debugging information format of the Linux and Unix operating systems and, although developed together with the ELF object file format, is independent of the object file format used to organize the contents of binary programs. Even more so, the DWARF debugging information format was designed to be flexible enough to be able to describe virtually any procedural programming language on virtually any architecture [Eag07]. Because the DWARF format is considerably complex, we will only give an overview on the most basic aspects in this place.

DWARF organizes debugging information in the form of *trees*, similar to those used by compilers as intermediate representations of source code, as described in Section 3.1.2. In fact, trees are a very natural means of representing the structure of programs written in block structured programming languages, in which entities, e.g., a variable, are contained within other entities, e.g., a function, and which are therefore predestined to be described in terms of parent-child relations.

Debugging Information Entries A *debugging information entry* (DIE) is the basic entity to describe an element of source code, where each entry is described by a classifying *tag* as well as a number of *attributes*. Organized within a *debugging information tree*, each entry may have an arbitrary number of children, and may as well hold references to other debugging information entries defined within the same *compilation unit*, e.g., the module, or within another. An oversimplified representation of a hypothetical program by means of debugging information entries is depicted in Figure 3.11 to illustrate these relationships.

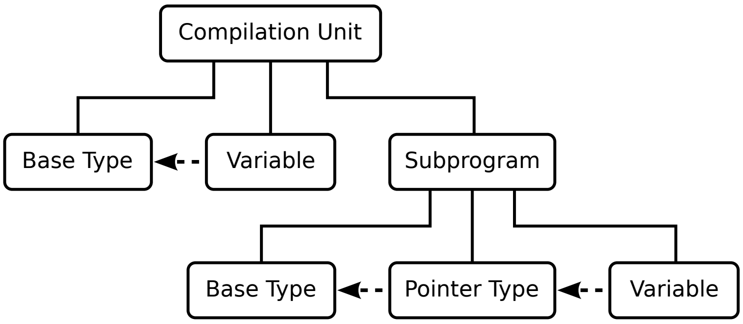


Figure 3.11: Sample DWARF Program Representation

[8] The DWARF Debugging Standard, http://www.dwarfstd.org/

The DWARF Debugging Standard specifies a plethora of debugging information entry types whose entire presentation would certainly go beyond the scope of this document. In this section, we will focus on the description of two simpler, yet significant, types of debugging information entries, `DW_TAG_base_type` and `DW_TAG_variable`, in a simplified way that is elaborate enough to explain a concluding usage example.

Base Types Debugging information entries tagged `DW_TAG_base_type` describe the *basic data types* of a particular programming language and serve as the building blocks for *complex data types* such as arrays, structures and unions. The attributes of this type are described in Table 3.14.

Type	**Description**
`DW_AT_name`	The name of the base type as recognized by the programming language, e.g., `char`, `int`, `float` or `double`.
`DW_AT_byte_size`	The size of the base type in bytes.
`DW_AT_encoding`	The encoding of the base type, as described in Table 3.15.

Table 3.14: DWARF Base Type Entry

Encoding	**Description**
`DW_ATE_boolean`	The base type refers to a *boolean.*
`DW_ATE_signed`	The base type refers to a *signed* quantity.
`DW_ATE_unsigned`	The base type refers to an *unsigned* quantity.
`DW_ATE_float`	The base type refers to a *floating point* number.
`DW_ATE_address`	The base type refers to a *memory address.*

Table 3.15: DWARF Base Type Entry - `DW_AT_encoding` Attributes

Variables Debugging information entries tagged `DW_TAG_variable` describe variables. The attributes of this type are described in Table 3.16.

Type	**Description**
`DW_AT_name`	The name of the variable as it appears in the source code.
`DW_AT_type`	The type of the variable, e.g., a reference to a base type.
`DW_AT_location`	The location of the variable during program execution.

Table 3.16: DWARF Variable Entry

Please note that the `DW_AT_location` attribute is not available for the representation of non-defining declarations of a variable, as for example '`int number`' in the C programming language.

We conclude this section on debugging information entries with the representation of the C programming language expression '`int number`' for the SPEAR2 32-bit architecture variant in a human readable form in Table 3.17.

```
<1>: DW_TAG_base_type
         DW_AT_name = int
         DW_AT_byte_size = 4
         DW_AT_encoding = DW_ATE_signed

<2>: DW_TAG_variable
         DW_AT_name = number
         DW_AT_type = <1>
```

Table 3.17: spear2_32-none-eabi-readelf Sample: DWARF Representation of '`int number`'

Besides the debugging information entries, the DWARF Debugging Standard provides for a variety of other debugging information, such as *line number information*, which maps the executable lines of source code to the corresponding binary contents in memory, *macro information*, which allows a programmer to inspect the "values" of preprocessor macros after compilation, as well as important *call frame information*, as described below.

Call Frame Information Call frame information (CFI) assists the debugger in finding a particular *call stack frame*, a storage segment which accommodates, most notably, the arguments, return value, local variables as well as saved register contents of a procedure in memory. Although the layout of a stack frame typically follows a predefined pattern, the exact dimensions of a stack frame depend on a variety of factors, such as the number of procedure arguments and local variables, but also on variety of other, not so obvious influences, such as whether the procedure eventually calls a sub-procedure, or whether the compiler manages to allocate space for a particular datum within an internal hardware register. The procedure calling conventions of an architecture are a complicated topic of their own which we will deal with extensively in Section 3.2.3.

DWARF identifies a stack frame by its *canonical frame address* (CFA), i.e., the memory address of the stack frame in the DWARF terminology. Register contents of a procedure saved on the stack frame are marked with their register numbers and address entries which are relative the the CFA address so that they can be found by the debugger. The debugger uses this information to *virtually unwind* the stack, a technique to virtually return to the state of a calling procedure by iteratively restoring saved register contents from the stack.

For more detailed information on the DWARF debugging information format we advise you to refer to [Eag07], which manages to provides an excellent overview on the capabilities of this widely used format.

3.1.8 Debugger and Simulator (gdb)

The GNU Debugger (GDB)[9] is a debugging framework for the analysis of binary programs written in programming languages such as C, C++, Java, Fortran and Ada, for a wide variety of architectures.

The purpose of a debugger is to analyze the processes going on inside a program during its execution. Besides, the GDB supports functionality which allows for the altering of the execution path of a program by manipulating registers and stack contents, as well as many more advanced features. In those embedded development environments which are made up of a *host computing system* and a *target computing system*, as described in Section 3.1.1, the debugger, executed on the host, communicates with the target through a *remote stub* which talks to the GDB instance through an implementation of the *GDB remote protocol*[10].

The *remote stub* is a software component which is executed from the boot memory of the SPEAR2 architecture, as described in Section 2.2.2, and that runs virtually in parallel to the actual program being remotely debugged. Because the remote stub also serves as a *loader* in that it places instructions and data of a binary program into their respective memory segments, it may well be regarded as a special type of bootloader, as described below. All communication between the debugger and the remote stub is realized via the *miniUART extension module*, as presented in [Sei07]. The remote stub introduces the *breakpoint* and *watchpoint* extension modules and has some stringent requirements on the hardware configuration of the SPEAR2 architecture, as documented in [Mei08].

As an alternative to *remote debugging*, the *GDB* port for the SPEAR2 architecture implements an *instruction set simulator* (ISS), which allows for binary programs to be executed within a simulated target environment on the host. Most importantly, the ISS simulates the *miniUART extension module* via a *pseudo terminal* master and slave pair located in `/dev/ptmx` and `/dev/pts/*`, respectively, where `*` represents an integer number that is dynamically assigned by the operating system. This certainly provides an effective means of inter-process communication between the simulated program and an arbitrary application, such as a test bench, which may send data to the simulated program via the *master device* and read from the simulated program via the *slave device*.

The debugging of a binary program requires debugging information to be included in the executable object file, which is provided by the compiler according to a debugging information format, such as DWARF. When debugging with the GDB, a program should be compiled with the `-ggdb` switch, which, additionally to `-g`, tells the GCC to generate any proprietary extensions, if possible. Although the generation of debugging information for optimized code is permitted, debugging will only produce the expected results if the program was compiled with `-O0`, the default.

[9] The GNU Debugger, http://www.gnu.org/software/gdb/

[10] GDB Remote Protocol, http://sourceware.org/gdb/current/onlinedocs/gdb/Remote-Protocol.html

3.1.9 Bootloader

A bootstrap loader, or simply *bootloader*, is typically the first program to be executed after a hardware reset. Bootloaders are stored in a dedicated, typically non-volatile, memory, the *boot-ROM*, and, upon execution, transfer control to some more sophisticated *operating system loader*, which extracts a compressed operating system kernel image into memory to then have control transferred to the kernel. Under the operating system, a *program loader*, most commonly referred to as "the loader", serves the loading of the loadable segments of an executable object file into a dedicated *process address space* in virtual memory, as outlined in Figure 3.12 and further described in Section 3.2.4.

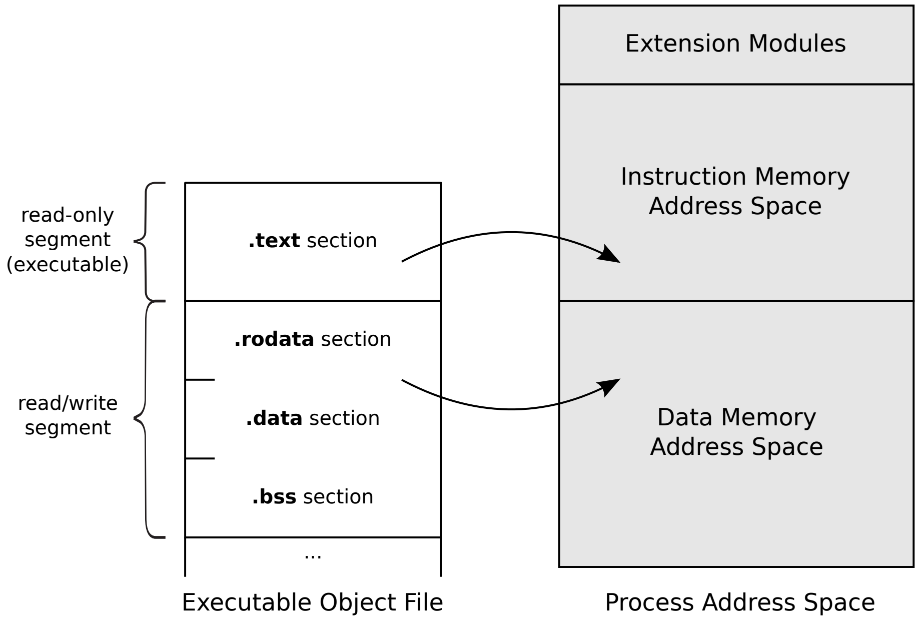


Figure 3.12: Segment-to-Memory Mapping

In the absence of an operating system, the boot sequence is greatly simplified: instead of loading an operating system the bootloader receives the loadable contents of a binary program, i.e., the application to be executed, in a packed format, such as *Intel HEX*[11], or *Motorola S-Record*[12], from the host via some transmission medium. The translation of the loadable segments of an executable object file into a packed representation ensures that only those kinds of information are retained for transmission which are suitable for being loaded into memory, plus the minimal overhead introduced by the chosen format which can be considered negligible. This certainly has a favorable impact on the complexity of the bootloader, which only needs to talk a simple protocol whose records can be processed line by line, thereby keeping the memory requirements of the bootloader application low. After having all loading done the bootloader jumps to the address of the first instruction in the instruction memory address space, thereby transferring control to the application.

[11] The Intel HEX Format, http://en.wikipedia.org/wiki/Intel_HEX

[12] The Motorola S-Record Format, http://en.wikipedia.org/wiki/SREC_(file_format)

The bootloader for the SPEAR2 architecture, in the absence of an operating system, implements the *Motorola S-Record* Format in the C programming language. The Motorola S-Record Format, which is closely related to the Intel HEX Format, in brief, has the virtue of encoding binary data in an easily parsable plain text, i.e., ASCII, representation: data is organized in so called records, structures which further comprise the number of data bytes a record accommodates and the address of the first data byte in memory as load memory address (LMA). The bootloader reads each record and tests whether the address field refers to an address in the *instruction memory address space* or in the *data memory address space*, as defined in Section 3.2.4. By having these address spaces be mapped into non-overlapping segments of a single load memory address space by linear address functions, the bootloader may safely assign each single datum to its designated address space within virtual memory by simple address arithmetics, as shown by a simplified excerpt of the SPEAR2 bootloader in Table 3.18.

```
struct srecord_t;
...

int main (int argc, char *argv[])
{
    struct srecord_t srec;
    ...

    while (1)
    {
        /* Read line from UART and parse srecord_t */
        ...

        /* Process the current srecord_t */
        switch (srec.type)
        {
            case SREC_TYPE_DATA4:
            {
                if (srec.address >= SPEAR2_CODEMEM_LMA)
                {
                    srec.address -= SPEAR2_CODEMEM_LMA;
                    program_codemem (&srec, buffer);
                }
                else
                    program_datamem (&srec, buffer);
                break;
            }
            case SREC_TYPE_DATA4_TERM:
            {
                /* Set the program counter to 0 */
                asm ("ldhi r13, 0");
                asm ("ldliu r13, 0");
                asm ("jmp r13");
            }
            default:
                continue;
        }
    }

    return 0;
}
```

Table 3.18: spear2-bootloader - Functional Principle

3.2 Application Binary Interface

An application binary interface (ABI) defines a variety of rules and conventions in order to achieve interoperability and compatibility between binary programs created for a given architecture. While many of these standards, such as the rules of *data representations*, the *register usage conventions*, the *standard procedure call sequence*, or the supported *object file formats* and *debugging information formats*, strongly depend on characteristics of the underlying hardware architecture, some of them are just arbitrarily defined. The specified standards must be implemented by all constituting parts of a toolchain in order to achieve tool-interoperability. Moreover, the collective conformance of a variety of toolchains for an architecture to the same ABI allows for code created with some $toolchain_A$ to be linked with code created with some other $toolchain_B$ without prior recompilation.

Furthermore, an ABI defines standards specific to the execution of binary programs within the context of an operating system, such as the *system call interface*, or the scheme for *dynamic linking* of incomplete programs with shared libraries at run-time. Obviously, both the operating system and the accompanying toolchain must follow the same software standards, since otherwise, a binary program may not even execute under the operating system at all. Moreover, a binary program may run on any operating system that adheres to the same ABI. Allow us to stress the importance of an ABI once more with the words of Levine [Lev99, Chapter 2]:

> *"From the point of view of an application, the ABI is as much a part of the system architecture as the underlying hardware architecture, since a program will fail equally badly if it violates the constraints of either."*

Although the conformance of a binary program created from high-level source code to an ABI is, for the most part, taken care of by the compiler and therefore usually remains unnoticed to the programmer, conformance must be taken into account in *mixed-mode programming*. Mixed-mode programming names a technique where high-level source code is interfaced by low-level assembly code and vice versa.

Since the availability of a full-fledged operating system cannot be taken for granted, particularly in the field of low-resource embedded systems, the collection of software standards for these *bare metal* systems is specified by the more lightweight *embedded application binary interface* (EABI). In the following we will describe the *SPEAR2 EABI*, supported by examples for the SPEAR2 32-bit architecture variant in the *C programming language*, although the same rules and conventions essentially apply to any other language as well.

3.2.1 Data Representations

The representation of data is strongly dependent on properties of the underlying hardware architecture: compilers take care that data objects are arranged in memory according to the respective dimensions and alignment restrictions of the various data types. Moreover, multi-byte data must be organized with respect to the byte-order requirements of the data memory address space.

3.2.1.1 Representation of Basic Data Types

The C programming language provides a variety of *basic data types* to the programmer, such as *char*, *short*, *int*, *long*, *float* and *double*, which essentially serve as the building blocks for *complex data types*. However, the bit sizes of the basic data types are not fixed by the C standard, but instead depend on the underlying hardware architecture and are defined as part of the architecture port of the C compiler.

The definitions of the *basic C data types* and how they are mapped to the *assembler data storage directives* as well as to the *memory addressing capabilities* of the SPEAR2 architecture, as described in Appendix A.1.3, are presented in Table 3.19.

C Type	**Assembler Directive**		**SPEAR2 Type**		**Size** [bits]	
	16-bit	32-bit	16-bit	32-bit	16-bit	32-bit
char	.byte	.byte	byte	byte	8	8
short	.short	.short	halfword	halfword	16	16
int	.short	.int	halfword	word	16	32
long	*emulated*	.int	*emulated*	word	32	32
long long	*emulated*	*emulated*	*emulated*	*emulated*	64	64
float	*emulated*	.int	*emulated*	word	32	32
double	*emulated*	*emulated*	*emulated*	*emulated*	64	64
long double	*emulated*	*emulated*	*emulated*	*emulated*	64	64

Table 3.19: SPEAR2 Basic C Data Types

Emulation of Higher-Order Quantities Please observe from Table 3.19 that the SPEAR2 architecture variants cannot access memory in quantities which are larger than their respective *word sizes*. Instead, references to *higher-order quantities*, i.e., those which go beyond the addressing capabilities of the architecture, must be emulated by means of software: they are assembled from a sequence of storage locations of word-size quantities of the respective architecture.

Alignment Restrictions and Byte-Order Requirements All accesses to the data memory address space must be *naturally aligned*. Furthermore, all multi-byte data must be represented in *little-endian* byte order, as described in Section 2.2.

The representations of the basic C data types in memory and the work involved by the C compiler are explained by the following example: the compiler translates the code sample in Table 3.20 into the assembly code sample in Table 3.21.

```
char        a = 1;
short       b = 2;
int         c = 3;
long        d = 4L;
long long   e = 5LL;
float       f = 6.0f;
double      g = 7.0;
long double h = 8.0;
```

Table 3.20: spear2_32-none-eabi-gcc C Code Sample: Basic C Data Types Representations

```
        .section .data
        .global a
    a:
        .byte 1
        .global b
        .p2align 1 ; align 'short b' on a 2-byte boundary
    b:
        .short 2
        .global c
        .p2align 2 ; align 'int c' on a 4-byte boundary
    c:
        .int 3
        .global d
        .p2align 2 ; align 'long d' on a 4-byte boundary
    d:
        .int 4
        .global e
        .p2align 3 ; align 'long long e' on an 8-byte boundary
    e:
        .int 5
        .int 0
        .global f
        .p2align 2 ; align 'float f' on a 4-byte boundary
    f:
        .int 1086324736
        .global g
        .p2align 3 ; align 'double g' on an 8-byte boundary
    g:
        .int 0
        .int 1086324736
        .global h
        .p2align 3 ; align 'long double h' on an 8-byte boundary
    h:
        .int 0
        .int 1075838976
```

Table 3.21: spear2_32-none-eabi-as Assembly Code Sample: Basic C Data Types Representations

Concluding, we give a summarizing overview on the final representations of the basic C data types in memory in Figure 3.13.

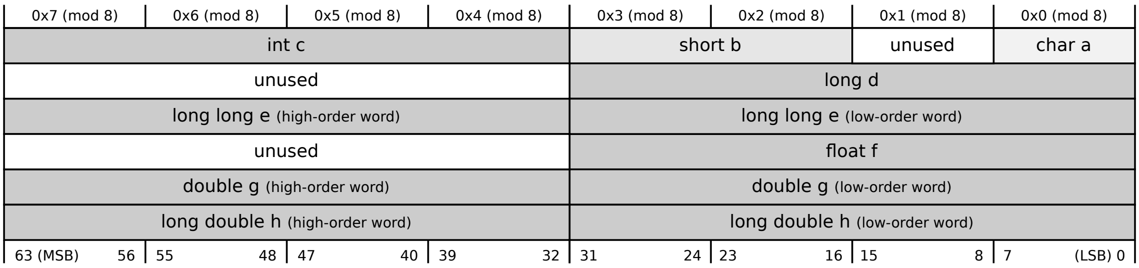


Figure 3.13: SPEAR2 (32-bit architecture): Basic C Data Types Representations

Signed and Unsigned Integer Data Types In the C programming language, each basic integer data type comes in *signed* and *unsigned* variants which are always the same size and, as a consequence, follow the same alignment restrictions. When the programmer does not explicitly specify which in the variable declaration, the compiler will implicitly generate a *signed* data type. However, the *char* data type plays an exceptional role: with many modern compilers a character defaults to an *unsigned char* [Swe06, Chapter 11].

Modern computing systems invariably represent their signed integer data types in the *two's complement* format, which comes with the particular advantage that the basic integer arithmetic operations, add, subtract, multiply and divide, have the same implementations for both signed and unsigned data types [Swe06, Chapter 11].

In the SPEAR2 architecture all basic integer C data types default to *signed* data types. Signed quantities are represented in the *two's complement* format. The ranges of the signed integer C data types for both SPEAR2 16-bit and 32-bit variants are shown in Table 3.22.

Signed C Type	**Range**	
	16-bit	32-bit
signed char	$[-2^7; 2^7-1]$	$[-2^7; 2^7-1]$
signed short	$[-2^{15}; 2^{15}-1]$	$[-2^{15}; 2^{15}-1]$
signed int	$[-2^{15}; 2^{15}-1]$	$[-2^{31}; 2^{31}-1]$
signed long	$[-2^{31}; 2^{31}-1]$	$[-2^{31}; 2^{31}-1]$
signed long long	$[-2^{63}; 2^{63}-1]$	$[-2^{63}; 2^{63}-1]$

Table 3.22: SPEAR2 Signed Integer C Data Types Ranges

Floating Point Data Types Since the SPEAR2 architecture has no hardware support for floating point operations, the basic floating point data types and the operations upon them are emulated in software on top of the basic integer data types: the C compiler turns a floating point operation into a function call within an internal software library, which then performs the required operation. The floating point library is implemented according to the *IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985).*

3.2.1.2 Representation of Complex Data Types

Generally speaking, a *complex data type*, or *compound data type*, is a collection of one or more basic data types which appear to the programmer as a single entity in a high-level programming language. The C programming language provides complex data types in the form of *struct*, *union* or *array* types.

Struct Types A struct type is a data structure whose members are arranged in memory in the order in which they are declared, where those which come first occupy the lower memory addresses. The alignment of a struct type is determined by the largest alignment boundary required by the contained elements. Unused bytes are inserted between the elements, a technique referred to as *padding*, in order to respect the individual alignment restrictions. Additionally, in order to allow for the concatenation in an *array type*, the C compiler automatically inserts unused bytes after the last element to respect the alignment restriction of the struct type [Swe06, Chapter 11].

The representation of a struct type in memory and the work involved by the C compiler are explained by the following example: the compiler translates the code sample in Table 3.23 into the assembly code sample in Table 3.24.

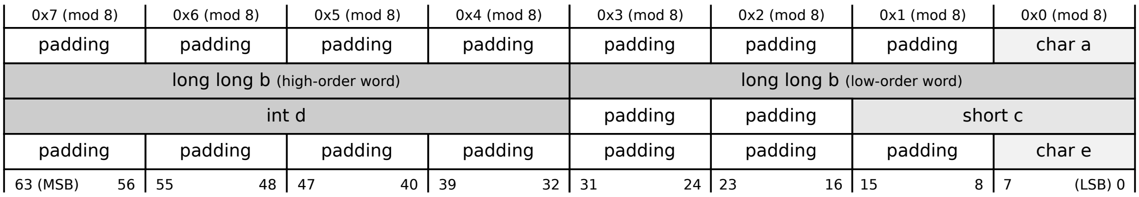
```
struct numbers
{
    char      a;
    long long b;
    short     c;
    int       d;
    char      e;
};

struct numbers n = {1, 2, 3, 4, 5};
```

Table 3.23: spear2_32-none-eabi-gcc C Code Sample: Struct Type Representation

```
     .section .data
     .global n
     .p2align 3 ; align 'struct numbers n' on an 8-byte boundary
n:
     .byte  1
     .skip  7 ; align 'long long b' on an 8-byte boundary by padding
     .int   2
     .int   0
     .short 3
     .skip  2 ; align 'int d' on a 4-byte boundary by padding
     .int   4
     .byte  5
     .skip  7 ; tail-pad 'struct numbers n' to an 8-byte boundary
```

Table 3.24: spear2_32-none-eabi-as Assembly Code Sample: Struct Type Representation

Furthermore, we give a summarizing overview on the final representation of the *struct numbers type* in memory in Figure 3.14.



0x7 (mod 8)	0x6 (mod 8)	0x5 (mod 8)	0x4 (mod 8)	0x3 (mod 8)	0x2 (mod 8)	0x1 (mod 8)	0x0 (mod 8)
padding	padding	padding	padding	padding	padding	padding	char a
long long b (high-order word)				long long b (low-order word)			
int d				padding	padding	short c	
padding	padding	padding	padding	padding	padding	padding	char e
63 (MSB) 56	55 48	47 40	39 32	31 24	23 16	15 8	7 (LSB) 0

Figure 3.14: SPEAR2 (32-bit architecture): Struct Type Representation

Observe from Figure 3.14 that padding may contribute considerably to the size of a struct type. An ordered arrangement of the constituting data types, e.g., from the smallest to the largest, would reduce the size of the *struct numbers* type from 32 bytes down to a mere 16 bytes.

Bit Fields The C programming language allows for the declaration of bit fields within a *struct type*, i.e., storage locations whose sizes may be smaller than what the compiler would usually allow: sizes may range from as few as 1 bit up to the bit size of the basic *int* data type and may either be *signed* or *unsigned* [Swe06, Section 11.6.1].

Although not officially supported by the standard, most compilers allow bit fields to run up to the size of any basic integer data type, provided that it was declared accordingly. Therefore, since each bit field is either implicitly or explicitly associated with a particular data type it might seem surprising that the organization of the data structure in memory is largely implementation-dependent. Consequently, a precise statement on the alignment requirements of bit fields cannot be made. Nevertheless, it is the common intent to *pack* bit fields as closely as possible into the common data structure. As a rule, bit fields are never padded, neither at the front nor at the end, to yield any particular alignment requirements.

Despite being dependent on the compiler implementation, bit fields also depend on the *endianness property* of the underlying hardware architecture: with *little-endian* byte order, the data structure is packed from the least significant bit to the most significant bit, whereas, with *big-endian* byte order, the data structure is packed the other way round. Therefore, the order in which bit fields are declared within the struct type must be taken into consideration when bit fields are used to exactly match a predefined data structure so that it can be referenced safely in the C programming language [SPHI02, Section 5.6.5].

Because of the above findings, bit fields are considered to be *non-portable* and hence must be used with caution.

The representation of bit fields in memory and the work involved by the C compiler are explained by the following example: the compiler translates the code sample in Table 3.25 into the assembly code sample in Table 3.26.

```
struct number
{
    unsigned int a:20;
    unsigned int b:3;
    unsigned int c:1;
};

struct number n = {0x10A0, 0x3, 1};
```

Table 3.25: spear2_32-none-eabi-gcc C Code Sample: Bit Fields Representation

```
     .section .data
     .global n
 n:
     .byte 160 ; = 0xA0
     .byte 16  ; = 0x10
     .byte 176 ; = 0xB0
```

Table 3.26: spear2_32-none-eabi-as Assembly Code Sample: Bit Fields Representation

Observe from Table 3.26 that bit fields are always assembled into a sequence of *.byte* storage specifiers, disregarding the dimensions of the associated data types. Consequently, bit fields do not require any particular alignment.

Concluding, we give another, more practical, example which represents what bit fields are predominantly used for: the *interfacing of hardware* which is mapped into the data memory address space of an architecture, as presented in Table 3.27.

```
#include <machine/modules.h> // defines PROC_CTRL_STATUS_C
typedef struct
{
    unsigned INTA:1;
    unsigned   ID:1;
    unsigned SRES:1;
    unsigned OUTD:1;
    unsigned EFSS:1;
    unsigned     :2;
    unsigned LOOW:1;
} ProcCtrlCustomConfigByte;

ProcCtrlCustomConfigByte *cfg = (*(volatile ProcCtrlCustomConfigByte *const) (PROC_CTRL_STATUS_C))
cfg->INTA = 1;
```

Table 3.27: spear2_32-none-eabi-gcc C Code Sample: Bit Fields for Hardware Interfacing

Union Types Union types are declared similar to struct types, yet are fundamentally different: while, in a struct type, members occupy different areas in memory, the members of a union share a common portion of memory whose size is determined by the size of its largest member. The alignment of a union type is determined by the largest alignment boundary required by the contained elements.

The representation of a union type in memory and the work involved by the C compiler are explained by the following example: the compiler translates the code sample in Table 3.28 into the assembly code sample in Table 3.29.

```
union number
{
    char      a;
    long long b;
    short     c;
    int       d;
    char      e;
};

union number n = {.b = 0x100000000LL};
```

Table 3.28: spear2_32-none-eabi-gcc C Code Sample: Union Type Representation

```
        .section .data
        .global n
        .p2align 3 ; align 'union number n' on an 8-byte boundary
    n:
        .int 0
        .int 1
```

Table 3.29: spear2_32-none-eabi-as Assembly Code Sample: Union Type Representation

Furthermore, we give a summarizing overview on the final representation of the *union number type* in memory in Figure 3.15.

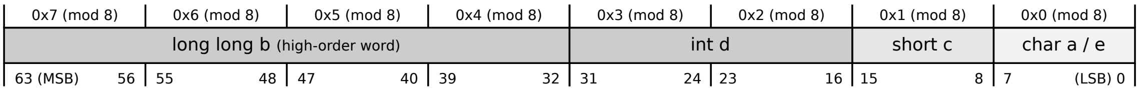


Figure 3.15: SPEAR2 (32-bit architecture): Union Type Representation

Observe from Figure 3.15 that the members of the *union number type* are effectively overlaid: the compiler allocates a common storage which can be accessed in different ways, according to the respective data types of the union members.

Array Types An array type is a data structure to accommodate a sequence of objects of a particular data type and number, whose elements are stored in a contiguous block of memory. Elements can be located via an array index, which denotes the position of the array element relative to the base address of the array in memory. The alignment of an array type is determined by the alignment boundary required by the declared data type which applies to all elements.

The representation of an array type in memory and the work involved by the C compiler are explained by the following example: the compiler translates the code sample in Table 3.30 into the assembly code sample in Table 3.31.

```
int n[] = {1, 2, 3, 4};
```

Table 3.30: spear2_32-none-eabi-gcc C Code Sample: Array Type Representation

```
        .section .data
        .global n
        .p2align 2 ; align 'int array n' on a 4-byte boundary
    n:
        .int 1
        .int 2
        .int 3
        .int 4
```

Table 3.31: spear2_32-none-eabi-as Assembly Code Sample: Array Type Representation

Concluding, we give a summarizing overview on the final representation of the *int type array n* in memory in Figure 3.16.

0x7 (mod 8)	0x6 (mod 8)	0x5 (mod 8)	0x4 (mod 8)	0x3 (mod 8)	0x2 (mod 8)	0x1 (mod 8)	0x0 (mod 8)
int n[1]				int n[0]			
int n[3]				int n[2]			
63 (MSB) 56	55 48	47 40	39 32	31 24	23 16	15 8	7 (LSB) 0

Figure 3.16: SPEAR2 (32-bit architecture): Array Type Representation

3.2.2 Register Usage Conventions

In addition to the hard-wired functionality of special purpose registers, compilers typically lay certain *usage conventions* upon general purpose registers in order to define, e.g., which registers hold the arguments and the return value of a procedure call and which register serves as the stack pointer register for building a call stack. The register usage conventions of the SPEAR2 architecture are shown in Table 3.32.

Name	Conventional Use	Caller Saved	Callee Saved
r0	Return value	Y	N
r1	Argument 1	Y	N
r2	Argument 2	Y	N
r3	Argument 3	Y	N
r4	Argument 4	Y	N
r5	Temporary (preserved across call)	Y	N
r6	Temporary (preserved across call)	Y	N
r7	Temporary (preserved across call)	Y	N
r8	Temporary (preserved across call)	Y	N
r9	Temporary (not preserved across call)	N	Y
r10	Temporary (not preserved across call)	N	Y
r11	Temporary (not preserved across call)	N	Y
r12	Temporary (not preserved across call)	N	Y
r13	Temporary	N	N
r14/rts	Procedure return address	N	Y[1]
r15/rte	Exception return address	N	Y[2]
FPW	Temporary (preserved across call)	Y	N
FPX	Temporary (preserved across call)	Y	N
FPY	Frame pointer	N	Y
FPZ	Stack pointer	-	-

[1] Only saved if procedure eventually calls a sub-procedure.

[2] Only saved if procedure implements an interrupt handler.

Table 3.32: SPEAR2 Register Usage Conventions

Please note that the *caller saved* and *callee saved* columns refer to a procedure call convention concerning whether the calling procedure or the called procedure will save a respective register, as described below.

3.2.3 Procedure Calling Conventions

Most high-level programming languages implement the concept of procedure calls to serve the modularization of high-level source code: a *procedure*, or function, receives a number of inputs, called *arguments*, and typically combines these in some particular way to create some output, the *return value.*

The *procedure calling conventions* define how control is transferred between the calling procedure, referred to as the *caller*, and the called procedure, the *callee*, as well as how inputs and outputs are passed along between them. Depending on the particular data types of the arguments and the return value, data can be exchanged within a combination of general purpose registers and a designated segment in memory, referred to as the *call stack*, or simply "the stack". The mutual commitment of the interacting procedures to the procedure calling conventions guarantees that both caller and callee cannot maliciously interfere with each others.

As an example, the *procedure calling conventions* permit a procedure to allocate its local variables on the stack, provided that it will deallocate any previously allocated space before it returns to the caller. Furthermore, the procedure calling conventions specify the layout of a storage segment on the stack, the *stack frame*, which is associated with the execution of a particular procedure and that defines where, e.g., the arguments, return value and the local variables have to be placed in memory.

3.2.3.1 Register Saving Conventions

Procedure calls invariably involve the saving of some processor state, which includes the saving of those registers which are used by the calling function, but are changed during the course of execution of the called function. As a consequence, caller and callee must adhere to the *register saving conventions*, which regulate which one of the interacting procedures must save these registers away upon the occurrence of a procedure call. The compliance of the caller and the callee to these conventions makes the register file appear to be mutually accessible by the currently executing procedure, an important prerequisite for safe procedure invocation [Dan05, Section 5.7].

In the *caller saving scheme*, the calling procedure saves those *caller saved registers* which it would like to have preserved across the call, whereas, in the *callee saving scheme*, the called procedure saves whichever of the *callee saved registers* it intends to use. In both schemes, the opposite procedure does not need to worry about register saving. Typically, a mixture of both schemes is applied.

Both caller saved and callee saved registers are stored within designated segments of the *stack frame*, as described in Section 3.2.3.2. The register saving conventions of the SPEAR2 architecture are shown in Table 3.32.

3.2.3.2 Stack Layout Conventions

The *stack memory model* essentially represents a first-in/last-out data structure located somewhere in the data memory address space, which is operated upon by the elementary *push* and *pop* operations, as described in Section 2.1.2.2. A designated pointer register, commonly referred to as the *stack pointer*, is automatically adjusted by the push and pop operations to always point to the top of the stack, i.e., to the most recently allocated space.

Modern architectures invariably implement a *falling stack*, which refers to its direction of growth in memory: the stack pointer of a falling stack is typically initialized to the top of the data memory from where it expands to lower addresses with each push operation, as illustrated in Figure 3.17.

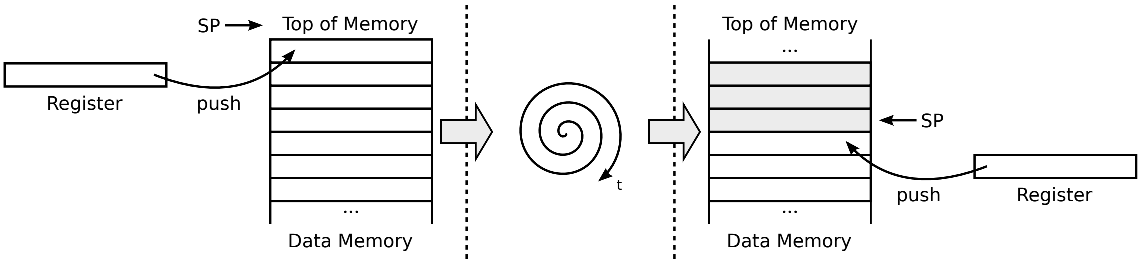


Figure 3.17: Basic Falling Stack Usage Scenario

The *call stack*, which is typically simply referred to as "the stack", is a high-level data structure that builds upon the stack memory model for the storing of statically allocated data of the currently executing process. The call stack is organized in *stack frames*, where each frame is associated with the execution of a particular procedure, for which it holds, e.g., the arguments, return value and the local variables.

The stack frame of a procedure is established in the *function prologue*, a small amount of maintenance code which is placed before the actual function body, and is destroyed again in the *function epilogue*, another small amount of code which is executed before the procedure returns to the caller, thereby deallocating any previously allocated space on the call stack. Therefore, at any time, the call stack represents the state of those procedures which have been called but have not yet finished executing.

Since the stack typically holds statically allocated data only, the dimensions of the stack frames can be determined successfully at compile-time. However, because the size of a stack frame depends on the memory requirements of the associated procedure, the dimensions of any two stack frames may vary almost arbitrarily. Therefore, the *call stack layout conventions* specify the basic structure of a stack frame, so that both caller and callee know where exactly to exchange their data. The compliance of the caller and the callee to these conventions makes the stack appear to be mutually accessible by the currently executing procedure, an important prerequisite for safe procedure invocation [Dan05, Section 5.7].

In the following we will describe the call frame structure of the SPEAR2 architecture, as depicted in Figure 3.18. The conventions of use of the various segments of a stack frame as well as how arguments and return values are exchanged between caller and callee in the SPEAR2 architecture are described in Table 3.33 and Table 3.34.

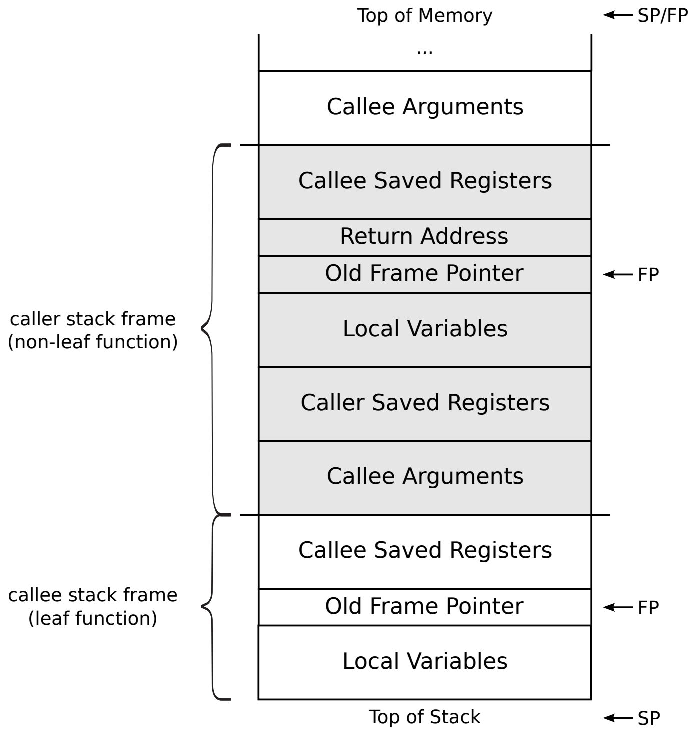


Figure 3.18: SPEAR2 Call Stack Frame Layout

Frame Pointer Concept In many modern architectures the addressing of data within a stack frame is facilitated by the existence of a dedicated *frame pointer*. The frame pointer is a special purpose register which points to about the beginning of the active stack frame, i.e., the frame on top of the stack that is associated with the currently executing procedure. The power behind the frame pointer concept is, as can be observed from Figure 3.18, that the local variables are located at known small negative offsets, whereas the arguments of the procedure are located at known small positive offsets from the frame pointer.

Leaf vs. Non-Leaf Functions We will stick to the terminology of the C compiler, in which the term *leaf function* refers to a procedure which constitutes a leaf in the *call tree representation* of a program, i.e., it does not make calls to sub-procedures. Consequently, leaf functions neither save the return address nor do they allocate space for any caller saved registers or any callee arguments on the stack.

Segment	**Conventions**
Callee Saved Registers	This segment holds whichever of the *callee saved registers* the associated procedure intends to use, as described in Section 3.2.3.1.
Return Address	This segment holds the return address of the associated procedure, i.e., the address of the instruction that follows the *jump-to-subroutine* (jsr) *instruction* which led to the execution of this procedure. This segment exists for non-leaf functions only.
Old Frame Pointer	This segment holds the frame pointer that points to the *previous stack frame*, i.e., the stack frame that is associated with the caller of the associated procedure.
Local Variables	This segment holds the local variables of the associated procedure. The variables are allocated in the *reverse order of declaration*, i.e., variables declared first appear last on the stack.
Caller Saved Registers	This segment holds those *caller saved registers* which the procedure would like to have preserved across a subsequent procedure call, as described in Section 3.2.3.1. This segment exists for non-leaf functions only.
Callee Arguments	This segment holds the arguments for a subsequent procedure call. The arguments are allocated in *right-to-left order*, i.e., arguments declared first appear last on the stack. This segment is established before a procedure call, and will be destroyed again after the callee has returned, by the caller.

Table 3.33: SPEAR2 Call Frame Segments - Usage Conventions

Data	**Conventions**
Arguments	The C compiler processes the arguments of a function from left to right and tries to accommodate the first four arguments in the *argument registers r1-r4*, as described in Section 3.2.2. Once an argument cannot be accommodated as a whole within a sequence of the available argument registers, all unprocessed arguments will be accommodated within the *callee arguments segment* of the caller stack frame, as described in Table 3.33. The arguments of a *variadic function*, i.e., a function which accepts a variable number of arguments, will always be accommodated on the stack.
Return Value	The C compiler tries to allocate space for the return value of a function in the *return value register r0*, as described in Section 3.2.2. When the return value cannot be accommodated as a whole within that register, the caller will pass a pointer to the space allocated for the return type on the caller stack frame as a "hidden" first argument within the *argument register r1.*

Table 3.34: SPEAR2 Call Stack - Data Exchange Conventions

3.2.3.3 Standard Procedure Call Sequence

The standard procedure call sequence defines the sequence of actions involved around the execution of a procedure that has been defined in a high-level programming language. Basically, the standard procedure call sequence concerns the transfers of control and data between the caller and the callee as well as stack management. The standard procedure call sequence according to the stack frame layout of the SPEAR2 architecture is presented in Table 3.35, in which the function under consideration takes on the role of a caller.

Sequence		**Action**
Function Prologue	1.	The *caller* saves any *callee saved registers* within the *callee saved registers segment.*
	2.	The *caller* saves the return address from the *return-from-subroutine* (rts) *register* in the *return address segment* of the caller stack frame (if the caller is not a leaf).
	3.	The *caller* saves the *frame pointer* in the *old frame pointer segment* of the caller stack frame.
	4.	The *caller* sets the *frame pointer* to the value of the *stack pointer.* From now on, the frame pointer always points to about the beginning, whereas the stack pointer always points to the end, of the caller stack frame.
	5.	The *caller* adjusts the stack pointer down to allocate space for the *local variables segment* and the *caller saved registers segment* of the caller stack frame.
Function Body	6.	The *caller* saves any *caller saved registers* within the *caller saved registers segment.*
	7.	The *caller* adjusts the stack pointer down to allocate space for the *callee arguments segment* of the caller stack frame (if the caller is not a leaf).
	8.	The *caller* accommodates the arguments of the callee in a combination of the *argument registers* and the *callee arguments segment* of the caller stack frame.
	9.	The *caller* accommodates the return value of the callee in the *return value register* or in some space within the caller stack frame.
	10.	The *caller* loads the address of the callee into a general purpose register and jumps to the procedure by executing the *jump-to-subroutine* (jsr) *instruction* upon that register. The processor saves the *return address* of the procedure, i.e., the address of the instruction that follows the *jsr*-instruction, in the *rts*-register.
	11.	**The callee is executed.**
	12.	The *caller* adjusts the stack pointer up again to deallocate the previously allocated *callee arguments segment* of the caller stack frame.
	13.	The *caller* assigns the return value of the callee to the variable receiving the return value within the *local variables segment* of the caller stack frame.
	14.	The *caller* restores any caller saved registers from the *caller saved registers segment* of the caller stack frame.
	15.	The *caller* accommodates its return value in the *return value register* or in some space within the stack frame of the calling procedure.
Function Epilogue	16.	The *caller* adjusts the stack up again to deallocate the previously allocated *local variables segment* and the *caller saved registers segment* of the caller stack frame.
	17.	The *caller* restores the frame pointer from the *old frame pointer segment* of the caller stack frame.
	18.	The *caller* restores the *rts*-register from the *return address segment* of the caller stack frame.
	19.	The *caller* restores any *callee saved registers* from the *callee saved registers segment* of the caller stack frame.
	20.	The *caller* executes the *rts*-instruction, thereby returning control to its calling procedure again.

Table 3.35: SPEAR2 Standard Procedure Call Sequence

3.2.4 Process Address Space

In a computing system, each process, i.e., the manifestation of a program's execution in memory, is assigned its own, private, address space: a *process address space* is a set of *virtual memory addresses*, which are mapped to the *physical memory addresses* through collaboration of an operating system and memory management facilities of the underlying hardware architecture. However, in the absence of memory management, processes have the full, albeit unprotected, physical memory at their own disposal. Therefore, in these kind of systems systems, the virtual memory addresses of the process address space equal the physical memory addresses.

The process address space for both SPEAR2 16-bit and 32-bit architecture variants, according to the *load memory address scheme*, as described in Section 3.1.4, is depicted in Figure 3.19 and Table 3.36.

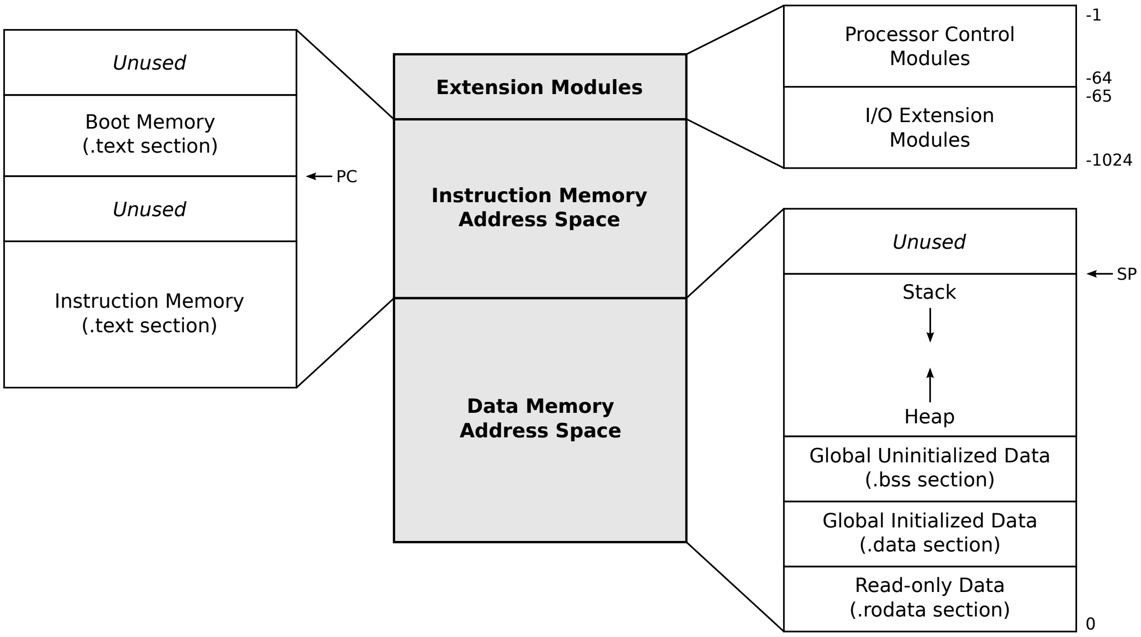

Unused
Boot Memory (.text section)
Unused
Instruction Memory (.text section)
PC
Extension Modules
Instruction Memory Address Space
Data Memory Address Space
Processor Control Modules
I/O Extension Modules
-1
-64
-65
-1024
Unused
SP
Stack
Heap
Global Uninitialized Data (.bss section)
Global Initialized Data (.data section)
Read-only Data (.rodata section)
0

Figure 3.19: SPEAR2 Process Address Space

Name	Address Range (LMAs) [bytes]	
	16-bit	32-bit
Data Memory Address Space	0x0000 : 0x7FFF	0x00000000 : 0x7FFFFFFF
Instruction Memory Address Space	0x8000 : 0xFBFF	0x80000000 : 0xFFFFFBFF
Extension Modules	0xFC00 : 0xFFFF	0xFFFFFC00 : 0xFFFFFFFF

Table 3.36: SPEAR2 Process Address Space Mappings

3.2.5 Object File and Debugging Information Format

The toolchain for the SPEAR2 architecture makes use of the ELF object file format and the DWARF debugging information format, as described in Section 3.1.6 and Section 3.1.7, respectively. In this section we refer to any additions made to these standards in order to enable support for the SPEAR2 architecture.

3.2.5.1 ELF Header Conventions

The supported values of the *e_machine* member of the ELF header, which specifies the required architecture of an individual ELF object file, for both SPEAR2 16-bit and 32-bit architecture variants are shown in Table 3.37.

Name	Value	Description
EM_SPEAR2_16	0x4F0C	SPEAR2 (16-bit variant), Vienna University of Technology, Austria
EM_SPEAR2_32	0x4F1C	SPEAR2 (32-bit variant), Vienna University of Technology, Austria

Table 3.37: SPEAR2 ELF Header Architecture Identification

The *e_flags* member of the ELF header allows for the accommodation of architecture-specific flags. The supported flags for both SPEAR2 16-bit and 32-bit architecture variants are shown in Table 3.38.

Name	Position	Description
EF_SPEAR2_16_MACH	0xF	Identifies the default SPEAR2 16-bit machine (optional)
EF_SPEAR2_32_MACH	0xF	Identifies the default SPEAR2 32-bit machine (optional)

Table 3.38: SPEAR2 ELF Header Flags

3.2.5.2 ELF Relocation Types

The term *relocation* refers to a central aspect during the linking process by which program addresses and references within multiple *relocatable object files* are adjusted to reflect their ultimate locations in virtual memory, as described in Section 3.1.4.3.

The relocatable instruction fields, i.e., those portions of an instruction set which contain relocatable operands, are depicted in Figure 3.20, whereas an overview on the relocatable instructions for both SPEAR2 16-bit and 32-bit architecture variants is given in Table 3.39.

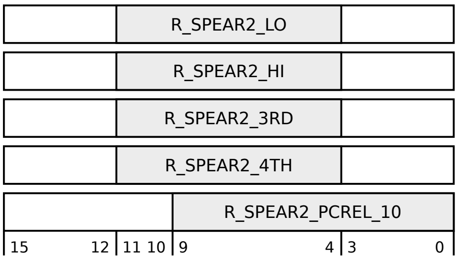


Figure 3.20: SPEAR2 Relocatable Instruction Fields

	Name	Relocation Type
32-bit	jmpi	R_SPEAR2_32_PCREL_10
	jmpi_cf	R_SPEAR2_32_PCREL_10
	jmpi_ct	R_SPEAR2_32_PCREL_10
	lo()[1]	R_SPEAR2_32_LO
	hi()[1]	R_SPEAR2_32_HI
	3rd()[1]	R_SPEAR2_32_3RD
	4th()[1]	R_SPEAR2_32_4TH
16-bit	jmpi	R_SPEAR2_16_PCREL_10
	jmpi_cf	R_SPEAR2_16_PCREL_10
	jmpi_ct	R_SPEAR2_16_PCREL_10
	lo()[2]	R_SPEAR2_16_LO
	hi()[2]	R_SPEAR2_16_HI

[1] spear2_32-none-eabi-as assembly instruction to extract the n-th byte from an n-byte argument
[2] spear2_16-none-eabi-as assembly instruction to extract the n-th byte from an n-byte argument

Table 3.39: SPEAR2 Relocatable Instructions

Please bear in mind that any arguments given to a relocatable instruction, as shown in Table 3.39, will only be subject to relocation if the address operand to be relocated explicitly refers to a symbol, i.e., a mnemonic identifying an object in memory. Constant operands are never relocated, since they are not structurally related to the entries in the symbol table somehow.

In the following, the relocation types for both SPEAR2 16-bit and 32-bit architecture variants and the calculations applied to any relocatable data items and instruction fields are presented. Observe that the operations required to extract the relocatable field from any neighboring fields of an instruction were left out for better readability. The symbols used to explain the address calculations of the various relocation types are described in Table 3.40.

Symbol	Description
S	The value of the symbol to be relocated.
A	The relocation addend, i.e., the value used to compute the absolute position of the symbol within the object file.
P	The relocation position offset, i.e., the offset of the symbol, relative to its segment within the executable object.

Table 3.40: SPEAR2 Relocation Types - Explanation of Symbols

	Name	Value	Size [bits]	Address Calculation
32-bit	R_SPEAR2_32_NONE	0	0	-
	R_SPEAR2_32_16	1	16	S + A
	R_SPEAR2_32_32	2	32	S + A
	R_SPEAR2_32_LO	3	8	(S + A) & 0x000000FF
	R_SPEAR2_32_HI	4	8	((S + A) & 0x0000FF00) $\gg$ 8
	R_SPEAR2_32_3RD	5	8	((S + A) & 0x00FF0000) $\gg$ 16
	R_SPEAR2_32_4TH	6	8	((S + A) & 0xFF000000) $\gg$ 24
	R_SPEAR2_32_PCREL_10	7	10	S + A - P
	R_SPEAR2_32_GNU_VTINHERIT	8	*n/a*	*n/a*
	R_SPEAR2_32_GNU_VTENTRY	9	*n/a*	*n/a*
16-bit	R_SPEAR2_16_NONE	0	0	-
	R_SPEAR2_16_16	1	16	S + A
	R_SPEAR2_16_32	2	32	S + A
	R_SPEAR2_16_LO	3	8	(S + A) & 0x00FF
	R_SPEAR2_16_HI	4	8	((S + A) & 0xFF00) $\gg$ 8
	R_SPEAR2_16_PCREL_10	5	10	S + A - P
	R_SPEAR2_16_GNU_VTINHERIT	6	*n/a*	*n/a*
	R_SPEAR2_16_GNU_VTENTRY	7	*n/a*	*n/a*

Table 3.41: SPEAR2 Relocation Types

Please note that the members of the GNU toolchain are laid out for *byte-addressable* memories only, a property which naturally applies to any modern architecture. However, since the instruction memory of the SPEAR2 architecture is *instruction-addressable* only, as described in Section 2.2.2, the linker adjusts all references to symbols which belong to the instruction memory through an additional *division by factor 2* at the end of relocation. This constant division factor reflects the fixed length instruction coding enforced by the SPEAR2 architecture, as described in Section A.1.4.

Chapter 4

Integrating the GNU Toolchain

Modern toolchains typically provide far more than the mere translation of high-level source code into a debuggable and executable binary program: the GNU toolchain, for example, which is predominantly used under GNU/Linux, and, for being free software, can be used, studied, copied, modified and redistributed freely under the terms of the GNU General Public License (GPL) [1], provides a wide variety of additional utility programs, such as for object code analysis or code profiling.

Essentially, all toolchains, independent of the actual services they provide, invariably have the same fundamental, yet unsurprising, characteristic in common: they implement the *instruction set architecture* and some *application binary interface* which supports the chosen architecture to achieve interoperability and compatibility between binary programs. This is why considerable efforts have been directed towards the elaboration on the *SPEAR2 Instruction Set Architecture* and the *SPEAR2 Embedded Application Binary Interface*, in Chapter 2 and Chapter 3, respectively, as a first essential step in the integration process.

The term "GNU toolchain" refers to a collection of software development tools which are themselves part of the GNU Project[2]. Besides the freedoms granted by free and open source software (FOSS), the GNU toolchain has the remarkable benefit of being available for a wide range of mainstream architectures, such as *ARM*, *Atmel AVR*, *Analog Devices' Blackfin*, *Intel i386*, *Motorola 68k*, *MIPS* and *Sun Microsystem SPARC*, as well as another large number of rather exotic computer architectures, such as *Axis Communications' CRIS*, *Fujitsu FR30*, *Tensilica Xtensa* and *OpenRISC*, to name just a few.

In this chapter we will present selected parts of our work involved in porting the members of the GNU toolchain for the SPEAR2 architecture, which happened during the practical part of this thesis, and share references to further readings where applicable.

[1] The GNU General Public License v3.0, http://www.gnu.org/licenses/gpl.html

[2] The GNU Project - Free Software Foundation (FSF), http://www.gnu.org/software/software.html

The goal of the practical part was the achievement of a functional toolchain which should be based on the tools of the GNU Project, albeit with a certain degree of freedom in the choice of its members, especially if particular properties of the underlying architecture were better supported by alternative solutions. In this sense, the term “functional” refers to the ability of translating source code in the C programming language into a debuggable and executable binary program which can either be simulated on the host or executed and debugged on the target platform in a correct manner.

Before we turn to the actual porting and integration work done in a bit more detail, let us first define where we started at and which supporting tools were available back then.

4.1 Analysis of Available Tools

Since the SPEAR2 processor has been in use for several years during Hardware/Software Codesign classes held at the Embedded Computing Systems Group of the Department of Computer Engineering at the Vienna University of Technology, a functional ensemble of supporting tools was already available for our evaluation, as outlined in [Puf07]:

4.1.1 GCC 4.1.1

The GNU Compiler Collection (GCC) was available as a functional port supporting the C programming language for both SPEAR2 16-bit and 32-bit architecture variants. Due to faults in the machine description, the compiler is unable to create valid code at a code optimization level greater than `-O0` (no optimization). Besides, a variety of other issues needed to be taken into consideration:

- Basic data types were represented by proprietary assembler data storage directives.
- `long long` and `double` data type representations were broken on 64-bit host systems.
- Call stack frame handling was flawed for both signal- and interrupt handler calls.
- The `libgcc` had moved out of the GCC into a minimal C Library implementation provided by the binutils package, as described below, thereby requiring reintegration.
- The compiler utilized the *stabs* debugging information format[3] which, with the wide dissemination of DWARF[4] on Linux and Unix operating systems alike, is considered deprecated nowadays.

Additionally, a multitude of smaller issues had to be considered in order to decouple the compiler from the proprietary interfaces of the other members of the toolchain.

[3] The stabs Debugging Information Format, http://sourceware.org/gdb/onlinedocs/stabs.html

[4] The DWARF Debugging Standard, http://www.dwarfstd.org/

4.1.2 CAS 0.5.1

The C-style Assembler (CAS) is a macro assembler which basically allows for assembly code to be mixed with sophisticated high-level language elements. Developed during a student's practical course, as described in [Puf04], the CAS project was originally designed to target the initial SPEAR architecture in the absence of a compiler. In this environment, however, CAS' contribution to the build process got reduced to simply act as an additional preprocessor for assembly files invoked by the assembler, as described below.

4.1.3 Binutils 0.3.6

The binutils, although not to be confused with the GNU Binutils, is a self-made project which contains an assembler (as) and a linker (ld). Instead of translating an assembly file into an object file containing binary representations of assembly language instructions and data, the assembler is merely an alias for the following sequence of operations: the input files are processed by the preprocessor portion of the CAS project, next syntax validated, to then be passed along to the linker which finally performs the actual translation into one of the following supported file formats:

- A binary format, also referred to as the *Spear2bin* format [Mei08, Section 5.2], which is suitable for download. The format, albeit binary, can be described as a minimal subset of the *Intel HEX*[5] format and may only contain instructions to be loaded into the instruction memory by the bootloader. This enforces an unusual dealing with program data in that data items are translated into sequences of instructions, where each datum is loaded into a register prior to being stored in memory at its destined address at program startup.

- The *Memory Initialization File* (MIF) format which, besides Intel HEX, is supported by the *Altera Quartus Memory Editor* to initialize the contents of an Altera device memory block. Likewise, as with the binary format above, program data is translated into sequences of store instructions which are executed before the actual program.

- The *Executable and Linking Object File* (ELF)[6] format, which is described in Section 3.1.6, was the last feature which made it into the binutils project in order to provide a qualified input file format for the recently ported GNU Debugger, as described below. However, this implementation of ELF enforced an instruction-wise addressing scheme for code, which indeed reflects the capabilities of the instruction memory address space of the SPEAR2 architecture, as described in Section 2.2.2, but which is incompatible with the strict byte-wise addressing scheme of the tools provided by the GNU Project.

Due to the lack of dedicated command line switches or even support for a linker script language, the mappings of instructions and data into memory are inherently immutable. Additionally, a number of bugs reportedly occurred in the anyway limited implementation of the C library, which was delivered as part of the binutils project.

[5] The Intel HEX Format, http://en.wikipedia.org/wiki/Intel_HEX

[6] Executable and Linking Format (ELF) Specification, http://refspecs.freestandards.org/elf/elf.pdf

4.1.4 GNU Debugger 6.6

The GNU Debugger (GDB) was available as a functional port for the SPEAR2 32-bit architecture variant only. As such, it implemented a description of the target architecture, provided a stub for remote debugging, as well as integrated the instruction set simulator which initially came as a standalone command-line component with the binutils project.

However, the port relied on the deprecated *stabs* debugging information format, which, in contrary to DWARF, does not come with any support for labeling and identification of call stack frames. This, of course, made the locating of run-time information, such as the state of local variables and procedure argument values, on the stack a difficult and error-prone task, since it involved heuristic analyses of the instruction stream. In the end, the approach did not do its job very well. Besides, a variety of other issues needed to be addressed:

- The simulator was integrated as a monolithic module. A modular, well-structured, design would considerably improve maintainability and extendability and therefore better support functional equivalence between the processor and the simulator.
- The simulator did not make a clean separation between LMA and VMA addresses, i.e., a clear and understandable memory address decoding scheme was not available.
- The simulator did not support the simulation of boot-ROM resident code, which did not allow for the simulation of bootloader software components.
- The simulator plug-ins representing the available extension modules were incomplete or required code improvements.

4.1.5 Bootloader

The bootloader for the SPEAR2 architecture was implemented in assembly language and supported the *Spear2bin* binary format, as described above. However, we considered this format not only proprietary, but also too limited in its design to adapt to the needs of a mature toolchain by at the same time maintaining backwards compatibility. Therefore, we decided to implement a new bootloader family in the C programming language on top of the more flexible *Motorola S-Record*[7] format, and was already described in Section 3.1.9.

4.1.6 Memory Maps

From an overall perspective, the members of the toolchain for the SPEAR2 architecture lacked another important aspect: a common view of the available memory address spaces. Almost each member of the toolchain had, according to its needs, its own notion of the memory hard-coded into its internals. This important observation created the need for a memory-mapping tool which should serve the central definition of platform-dependent, toolchain-wide, memory-maps through the creation of dedicated dependency files, including Makefiles and linker scripts for rapid application development.

[7] The Motorola S-Record Format, http://en.wikipedia.org/wiki/SREC_(file_format)

4.2 Integration of Tools

A thorough analysis of the given requirements and the available tools eventually led to the following tools become part of the GNU toolchain for the SPEAR2 architecture, targeting both 16-bit and 32-bit architecture variants. Please note that the members of the toolchain have been presented extensively at a behavioral level in Section 3.1.

- GCC[8] 4.1.1
- GNU Binutils[9] 2.19
- Red Hat newlib C Library[10] 1.17
- GNU Debugger[11] 6.8

4.2.1 Approach

The porting and integration of a toolchain is, generally speaking, a non-trivial task. An in-depth knowledge of the *instruction set architecture* is an essential prerequisite, so that peculiarities as well as potential weaknesses of the underlying hardware architecture can be identified and taken care of in some way during the porting procedure. Then, usually beginning with the compiler port, an *application binary interface* has to be defined and implemented by all members of the toolchain, albeit with varying focus. As regards the SPEAR2 architecture, however, the documentation on the instruction set architecture was often inaccurate and the application binary interface implemented by the legacy software development tools, as described above, was almost not documented at all.

Porting the GNU toolchain first of all involves poring over the official documentation that is delivered with the source codes, describing both internal and external programming interfaces of the respective members. Additional help can then be acquired from dedicated newsgroups on the Internet. Since the official documentation is quite comprehensive, the *GNU Compiler Collections Internals*[12] manual alone has more than 600 pages, developers seldomly start with their implementations from scratch. Instead, the implementations for a similar architecture are studied and refined until they reflect the characteristics of the own architecture, taking the manuals as a supporting reference. While this approach alone may not seem elaborate enough to, as for example, produce a highly optimized compiler port, it is commonly regarded as an effective approach to getting a functional port done with reasonable efforts.

[8] The GNU Compiler Collection, http://gcc.gnu.org

[9] The GNU Binutils, http://sourceware.org/binutils/

[10] The Red Hat newlib C Library, http://sourceware.org/newlib/

[11] The GNU Debugger, http://www.gnu.org/software/gdb/

[12] The GNU Compiler Collection Internals, http://gcc.gnu.org/onlinedocs/gccint/

4.2.2 GCC 4.1.1

The GCC was available as a functional port supporting the C programming language for both SPEAR2 16-bit and 32-bit architecture variants, as described in Section 4.1.1. While version 4.1.1 might seem obsolete at a first glance, analyses revealed that the migration of the compiler port to a newer build would have required considerable efforts due to the vast number of changes the compiler generation framework had undergone in the meantime. Since the migration was neither required nor particularly advantageous for the overall functionality of the toolchain, plus the compiler was known for its overall good stability, we decided to go with this version and make changes in whichever places necessary.

4.2.2.1 Machine Descriptions

Each supported architecture provides a *machine description* which allows the compiler generation framework to obtain information on the capabilities of the respective target architecture. A machine description is basically separated into the following two parts:

- A ⟨`machine`⟩`.md` file containing *instruction patterns* which mainly specify the rules by which code in the intermediate RTL language has to be translated into assembly language instructions specific to the target architecture.
- Supplementary ⟨`machine`⟩`.h` and ⟨`machine`⟩`.c` files which implement processor-specific details that cannot be accommodated by the ⟨`machine`⟩`.md` file, such as word size and endianness properties, sizes and alignments of basic data types, naming, description and classification of hardware registers, addressing modes, stack frame layout, and many more.

Constituting the centerpiece of any architecture port, machine descriptions may run up to enormous sizes. An overview on the approximate sizes of various machine descriptions in terms of line numbers of the GCC 4.1.1 source tree is given in Table 4.1.

Architecture	**Instruction Patterns** **Size** [lines]	**Macros and Functions** **Size** [lines]
ARM	11k	18k
Atmel AVR	3k	7k
Analog Devices' Blackfin	2k	4k
Intel i386	29k	33k
Motorola 68k	7k	7k
MIPS	10k	15k
SPEAR2	2k	3k
Sun Microsystems SPARC	10k	14k
Tensilica Xtensa	2k	4k

Table 4.1: Sizes of Machine Descriptions in the GCC 4.1.1

In the following we will summarize the adaptations made to the machine descriptions for both SPEAR2 16-bit and 32-bit architecture variants, according to the outcome of the analysis in Section 4.1.1, in the directories `gcc/config/spear16` and `gcc/config/spear32` of the GCC 4.1.1 source tree. Despite not being mentioned explicitly, all adjustments described below apply equally to both architecture variants.

Basic Data Types Basic data types get allocated by the standard `.byte`, `.short` and `.int` assembler data storage directives, as recognized by the GNU Assembler, in `spear32.c`, lines 125f.

Call Stack Frame The basic procedure by which the compiler establishes disruptible and non-disruptible interrupt handlers, as described in Section 3.1.5.2, is shown in Table 4.2. As we recall from Section 2.1.2.1, nested interrupt handlers must, most notably, save the contents of the *return registers* (lines 16, 28) and the *saved custom status register* on the stack (lines 18-20) at the beginning and restore these registers before returning. Additionally, interrupts need to be globally re-enabled (lines 22-25).

```
01: static void spear32_output_function_prologue (FILE *file, HOST_WIDE_INT size)
02: {
03:     int interrupt_func_p, signal_func_p;
04:     interrupt_func_p = interrupt_function_p (current_function_decl);
05:     signal_func_p = signal_function_p (current_function_decl);

06:     if (interrupt_func_p || signal_func_p)
07:     {
08:         /* Push contents of r7 onto stack */
09:         fprintf (file, "stfpz_dec r7, -1\n”);
            ...
10:         /* Push contents of r13 onto stack */
11:         fprintf (file, "stfpz_dec r13, -1\n”);
            ...
12:     }

13:     if (interrupt_func_p)
14:     {
15:         /* Push return address (r15/rte) onto stack */
16:         fprintf(file, "stfpz_dec r15, -1\n”);
            ...
17:         /* Push saved custom status register onto stack */
18:         fprintf (file, "ldli r7, -8\n”);
19:         fprintf (file, "ldbu r13, r7\n”);
20:         fprintf (file, "stfpz_dec r13, -1\n”);

21:         /* Set GIE bit at bit number 7 in the custom config byte */
22:         fprintf (file, "ldli r7, -29\n”);
23:         fprintf (file, "ldbu r13, r7\n”);
24:         fprintf (file, "bset r13, 7\n”);
25:         fprintf (file, "stb r13, r7\n”);
26:     }

27:     /* Push callee saved registers onto stack */
28:     /* Push return address (r14/rts) onto the stack */
        ...
```

Table 4.2: `gcc-4.1.1/gcc/config/spear32/spear32.c` - Interrupt Handler Description

GCC Runtime Library (libgcc) Those functions of the C Library implementation provided by the binutils package, which were related to operations on integer data types, were extracted and integrated into the files div.c, fixfloat.c and libgcc.S. This included, for example, multiplication *, signed and unsigned division /, signed and unsigned modulo operation %, shift left $\ll$ and arithmetic and logical shift right $\gg$. The makefile fragment t-spear32 was configured to generate proper emulations of floating point data types and the operations upon them, as well as to include the above files into libgcc.a.

DWARF Debugging Information Format DWARF replaces the obsolete *stabs* as the default debugging information format in spear32.h (lines 609f). In order to support DWARF, the basic stack layout was provided (lines 206f). Moreover, the generation of the function prologues needed to be extended to establish the cannonical frame address (CFA) for stack frame identification as well as to generate supplementary call frame information (CFI) to assist in identifying data relative to the CFA in spear32.c (lines 520f).

Proprietary Command Line Options With the SPEAR2 32-bit architecture variant, the proprietary command line options -mcodesize and -mdatasize were used to reduce the number of instructions issued when loading a symbol into a register, as described in Appendix A.2.3. The basic idea was that, if both codesize and datasize were $< 2^{15}$, then the number of required instructions could be reduced from 6 down to 2 with each occurrence. However, this optimization scheme silently assumed that for all applications instructions and data started at address 0x0. With the advent of the GNU Binutils for the SPEAR2 architecture and their support for linker scripts, these options involved a high risk of corrupting memory references and therefore got removed from the implementation. Additionally, the proprietary command line options -mmif, -mhex, -melf, -msep, -mlib and -minit-stack became obsolete and dropped out for good as well.

Miscellaneous

- Added __SPEAR2_32__ as a built-in macro definition to the preprocessor in spear32.h (line 28). This definition becomes a useful feature once an application is required to support both SPEAR2 16-bit and 32-bit architecture variants by, at the same time, avoiding code duplication. The programmer may then query the existence of this definition, e.g., via #ifdef __SPEAR2_32__ and act upon accordingly.

- A proper .file assembler directive gets issued at the start of each assembly file, as required by the GNU Assembler, in spear32.c (lines 1356f). Required support for named sections was added in spear32.c (lines 157f and 2552f). The issuing of preprocessor macros in generated assembly files, which were resolved by the binutils package, e.g., __tmp_reg__, was either replaced with meaningful values or removed throughout the implementation.

- The compiler driver passes the strings “-lc -lnosys”, “-lgcc” and “crt0.o” to the linker, which makes the application be automatically linked against the libraries libgcc.a, libc.a, libnosys.a and crt0.o, if required, in spear32.h (lines 575f).

4.2.3 GNU Binutils 2.19

Before we turn to selected tidbits from the porting of the GNU Binutils for the SPEAR2 architecture, let us quickly summarize how they are basically organized. After unpacking the source archive, the user is presented the directories described in Table 4.3.

Directory	**Description**
`bfd`	This directory contains the *Binary File Descriptor library* (libbfd), an object file library which allows applications to operate on a variety of object file formats through a common API. The libbfd implements the *application binary interface* in terms of the supported object file formats and relocation types of a particular architecture, and is used by the assembler, linker, binary utilities and the GNU Debugger project alike.
`binutils`	This directory contains the binary utilities, such as `objcopy`, `objdump` and `readelf`.
`cgen`	This directory contains CGEN, a framework for generating a *libopcodes* port from a central machine description, as further described below. This directory is not part of the GNU Binutils 2.19 source distribution, however, support for using CGEN is provided in the build environment of the GNU Binutils.
`cpu`	This directory contains the machine description source files for CGEN.
`elfcpp`	This directory contains elfcpp, a C++ ELF library supporting the gold linker.
`gas`	This directory contains the GNU Assembler. Architecture-specific source files are located in the `gas/config/` subdirectory.
`gold`	This directory contains gold, an ELF-only linker written in C++. Still under development, gold is planned to become a replacement for the older GNU Linker.
`gprof`	This directory contains the GNU Profiler, a code profiling tool.
`include`	This directory contains common header files. Architecture-specific header files are located in the subdirectories.
`intl`	This directory contains libintl, an internationalization library.
`ld`	This directory contains the GNU Linker. Architecture-specific source files are located in the subdirectories.
`libiberty`	This directory contains libiberty, a collection of functions used by several GNU programs, such as `getopt`, `strerror`, `strtol` and `strtoul`.
`opcodes`	This directory contains the *opcodes library* (libopcodes), containing information on how to assemble and disassemble instructions. The libopcodes implements a subset of the *instruction set architecture* of a particular architecture.

Table 4.3: spear2-binutils-2.19 - Directory Contents

In the following we will take a closer look at how selected aspects of the instruction set architecture and the application binary interface of the SPEAR2 architecture are realized within the *bfd* and the *opcodes* directories of the GNU Binutils 2.19. All examples given below apply to the SPEAR2 32-bit architecture variant and are presented in a simplified form in order to capture the essence of our work.

4.2.3.1 opcodes

The *libopcodes* port for the SPEAR2 architecture is generated by *CGEN*[13], a framework which aims at providing a central, application-independent, description of an architecture and its various concrete machine implementations from which software development tools of all kinds could be generated. At the time of writing, CGEN supports the generation of the *libopcodes* and the *sim* instruction set simulator for the GNU Debugger only.

With CGEN, a machine description consists of a description file `⟨arch⟩.cpu` in which the capabilities of the target architecture in terms of the functional hardware units and the instruction set architecture, are defined in a Scheme[14]-like description language. Besides, additional support for libopcodes is provided through `⟨arch⟩.opc`, in which supporting routines, such as a disassembler hash function and instruction operand validation routines, can be accommodated by the programmer in the C programming language.

When the GNU Binutils are configured with the `--enable-cgen-maint` switch, the CGEN framework will be instructed to generate the libopcodes for the given `--target`, providing the files described in Table 4.4.

File	**Description**
`⟨arch⟩-desc.h/.c`	These files define a variety of preprocessor macros and types to describe the functional hardware units and the instruction set architecture of the target architecture plus supporting routines.
`⟨arch⟩-ibld.c`	This file contains routines for instruction building and information retrieval.
`⟨arch⟩-opc.h/.c`	The files contains additional preprocessor macros and types which are of special interest to the assembler and the disassembler and are not already contained in `⟨arch⟩-desc.h`, such as assembler syntax tables for instructions and macro instructions plus supporting routines.
`⟨arch⟩-asm.c`	This file contains supporting routines for the assembler.
`⟨arch⟩-dis.c`	This file contains supporting routines for the disassembler.

Table 4.4: libopcodes - Generated by CGEN

[13] CGEN, the Cpu tools GENerator, http://sourceware.org/cgen/

[14] The Scheme Programming Language, http://en.wikipedia.org/wiki/Scheme_(programming_language)

The representation of an architecture and its concrete machine implementations in the machine description file ⟨arch⟩.cpu is shown in Figure 4.1, followed by a concrete example in Table 4.5.

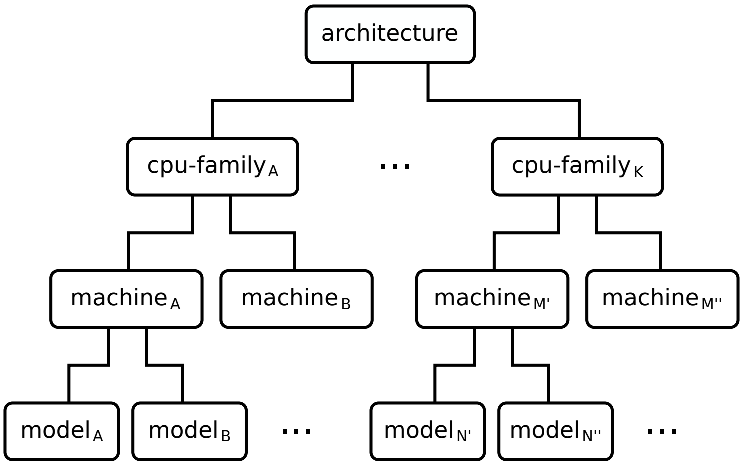


Figure 4.1: CGEN Architecture/Machine Hierarchy

```
; define-arch defines the overall architecture.
(define-arch
    (name spear2_32)
    (comment "SPEAR2 (32-bit architecture variant) architecture")
    (default-alignment aligned)   ; default alignment for accessing instructions and data in memory
    (insn-lsb0? #t)               ; define if the LSB in a word is bit number 0
    (machs spear2_32)             ; list of machine names in this architecture
    (isas spear2_32)              ; list of ISA names in this architecture
)

; define-cpu describes a CPU family (a class of machines specified by the programmer).
(define-cpu
    (name spear2_32bf)
    (comment "SPEAR2 (32-bit architecture variant) CPU base family")
    (endian little)
    (word-bitsize 32)
)

; define-mach defines a distinct CPU variant (machine).
(define-mach
    (name spear2_32)
    (comment "SPEAR2 (32-bit architecture variant) machine")
    (cpu spear2_32bf)             ; the name of the CPU family
    (isas spear2_32)              ; list of ISA names this machine supports
)

; define-model defines a concrete implementation of a CPU variant.
(define-model
    (name spear2_32gm)
    (comment "SPEAR2 (32-bit architecture variant) generic model")
    (mach spear2_32)              ; the name of the CPU variant
)
```

Table 4.5: binutils-2.19/cpu/spear2_32.cpu - Architecture/Machine Hierarchy

Since a single machine may support multiple instruction set architectures, as shown in Table 4.5, line 22, an instruction set architecture is represented by its own hierarchy in the machine description file ⟨arch⟩.cpu, as shown in Figure 4.2.

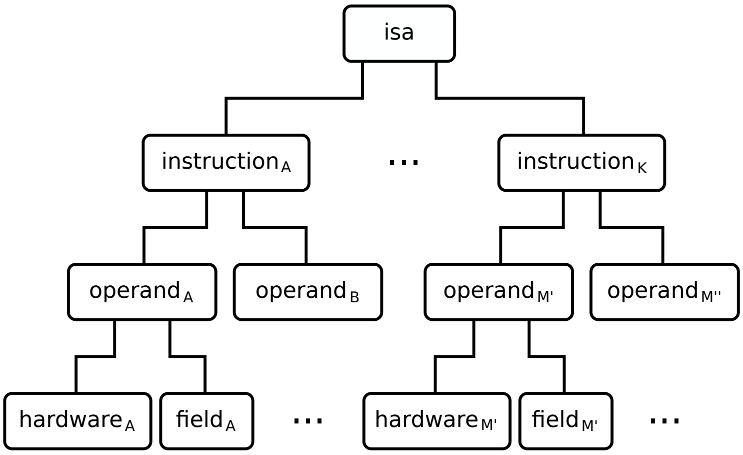


Figure 4.2: CGEN Instruction Set Architecture Hierarchy

As an example, the definitions of the jmpi, jmpi_cf and jmpi_ct instructions of the SPEAR2 architecture, as described in Appendix A.2.4, are provided in Table 4.6.

```
; define-isa describes the overall instruction set architecture.
(define-isa
    (name spear2_32)
    (comment "SPEAR2 (32-bit architecture variant) instruction set architecture")
    ; instructions are always 16 bits wide
    (default-insn-word-bitsize 16)
    (default-insn-bitsize 16)
    (base-insn-bitsize 16)
)

; dnf (define-normal-field) is a simplification macro for instruction field definitions.
; This defines the instruction field ’f-opc-6’ ranging from bit 15 down to 10.
(dnf
    f-opc-6                             ; name
    "6-bit opcode field"                ; comment
    ()                                  ; list of attributes
    15                                  ; start bit position
    6                                   ; length (bits)
)

; df (define-field) is another simplification macro for instruction field definitions.
; This defines the instruction field ’f-simm-10’ ranging from bit 9 down to 0 to hold
; a PC-relative integer address value with an optional sign.
(df
    f-simm-10                           ; name
    "10-bit signed immediate field"     ; comment
    (SIGN-OPT PCREL-ADDR)               ; list of attributes
    9                                   ; start bit position
    10                                  ; length (bits)
    INT                                 ; mode
)
```

```
; define-operand defines an instruction operand (from CGEN's perspective, an operand serves as a
; layer between the assembler and a hardware element (see the documentation for more information).
; This defines an instruction operand 'saddr10-pcrel' mapped to the 'f-simm-10' instruction field.
; The parser produces a call to parse_saddr10_pcrel(), defined in spear2_32.opc, for the assembler.
(define-operand
    (name saddr10-pcrel)
    (comment "10-bit signed pc-relative address operand")
    (type h-sint)                         ; the name of the hardware element
    (index f-simm-10)                     ; the index of the hardware element
    (handlers (parse "saddr10_pcrel"))
)

; define-normal-insn-enum defines names for the opcodes of an instruction field.
; This defines the names and values of the opcodes held in the instruction field 'f-opc-6'.
(define-normal-insn-enum
    insn-opc-6                            ; name
    "6-bit opcode enum"                   ; comment
    ()                                    ; list of attributes
    OPC6_                                 ; prefix
    f-opc-6                               ; the name of the instruction field
    (                                     ; list of name/value pairs
        (LDFPW #x18) (LDFPX #x19) (LDFPY #x1A) (LDFPZ #x1B)
        (STFPW #x1C) (STFPX #x1D) (STFPY #x1E) (STFPZ #x1F)
        (ADDI #x2A) (ADDI_CT #x26) (ADDI_CF #x22)
        (JMPI #x2B) (JMPI_CT #x27) (JMPI_CF #x23)
    )
)

; dni (define-normal-instruction) is a simplification macro for instruction definitions.
; This defines the instruction 'jmpi' as an unconditional control transfer instruction.
; The opcode is assigned the expression OPC6_JMPI, where JMPI is the index into the OPC6_ enum table.
(dni
    jmpi                                  ; name
    "Jump immediate"                      ; comment
    (UNCOND-CTI)                          ; list of attributes
    "jmpi $saddr10-pcrel"                 ; assembly syntax
    (+ OPC6_JMPI saddr10-pcrel)           ; format specification
)

; This defines the instruction 'jmpi_cf' as a conditional control transfer instruction.
(dni
    jmpi_cf                               ; name
    "Jump immediate if cond-flag false"   ; comment
    (COND-CTI)                            ; list of attributes
    "jmpi_cf $saddr10-pcrel"              ; assembly syntax
    (+ OPC6_JMPI_CF saddr10-pcrel)        ; format specification
)

; This defines the instruction 'jmpi_ct' as a conditional control transfer instruction.
(dni
    jmpi_ct                               ; name
    "Jump immediate if cond-flag true"    ; comment
    (COND-CTI)                            ; list of attributes
    "jmpi_ct $saddr10-pcrel"              ; assembly syntax
    (+ OPC6_JMPI_CT saddr10-pcrel)        ; format specification
)
```

Table 4.6: binutils-2.19/cpu/spear2_32.cpu - jmpi/jmpi_cf/jmpi_ct Instructions

4.2.3.2 bfd

The *libbfd* provides a type `bfd_arch_info_type` to contain architectural information that is associated with each binary file descriptor in `bfd/cpu-⟨arch⟩.c`. The `bfd_arch_info_type` contains a unique identifier of type `enum bfd_architecture`, which is defined for each supported architecture and which is used to access a variety of data structures that are specific to that particular architecture. A concrete example of the `bfd_arch_info_type` is presented in Table 4.7.

```
const bfd_arch_info_type bfd_spear2_32_arch =
{
    32,                   /* 32 bits per word */
    32,                   /* 32 bits per address */
    8,                    /* 8 bits per byte */
    bfd_arch_spear2_32    /* enum bfd_architecture architecture */
    bfd_mach_spear2_32,   /* Machine identifier */
    "spear2_32",          /* Architecture name (short version) */
    "spear2_32",          /* Architecture name (long version) */
    2,                    /* Section alignment power (log2) */
    TRUE,                 /* TRUE if this is the default machine for the architecture */
    spear2_32_compatible, /* Function for testing for compatibility between machines */
    bfd_default_scan,     /* Function to find a bfd_arch_info_type from a name string */
    NULL                  /* Pointer to the next spear2_32 machine architecture */
};
```

Table 4.7: `binutils-2.19/bfd/cpu-spear2_32.c` - Architectural Information

The porting of the libbfd is predominantly concerned with the implementation of the relocation types for the object file formats supported by the target architecture. Within the libbfd, support for a particular architecture and a given object file format is provided in `bfd/⟨fileformat⟩-⟨arch⟩.c`. Therein, relocation types are represented as an array of type `reloc_howto_type`, which provides a variety of parameters defining how a relocation is actually performed. The `reloc_howto_type` for the `R_SPEAR2_32_PCREL_10` relocation type, as defined in Section 3.2.5.2, is described in Table 4.8. The application of the parameters of the `reloc_howto_type` is explained in Table 4.11.

```
/* HOWTO is a simplification macro to define a reloc_howto_type */
HOWTO (
    R_SPEAR2_32_PCREL_10,     /* type */
    0,                        /* rightshift */
    1,                        /* size (16 bit) */
    10,                       /* bitsize */
    TRUE,                     /* pc_relative */
    0,                        /* bitpos */
    complain_overflow_dont,   /* complain_on_overflow */
    "R_SPEAR2_32_PCREL_10",   /* name */
    0,                        /* src_mask */
    0x3FF                     /* dst_mask */
)
```

Table 4.8: `binutils-2.19/bfd/elf32-spear2_32.c` - Relocation Type Representation

Clearly, a `reloc_howto_type` must somehow be associated with an instruction operand. Recall from Section 3.1.4.3 that a relocatable object file contains a collection of relocation entries which are provided by the assembler, as described in Section 3.1.3.2. Essentially, the assembler iterates the instruction operands of each relocatable instruction and determines and installs the appropriate relocation types. An example for looking up the relocation type for the address operand of the `jmpi`, `jmpi_cf` and `jmpi_ct` instructions in `md_cgen_lookup_reloc`, defined in `gas/config/tc-spear2_32.c`, is shown in Table 4.9.

```
bfd_reloc_code_real_type md_cgen_lookup_reloc (
    const CGEN_INSN *insn ATTRIBUTE_UNUSED,
    const CGEN_OPERAND *operand,
    fixS *fixP)
{
    ...
    switch (operand->type)
    {
        ...
        /* The operand type SPEAR2_32_OPERAND_SADDR10_PCREL is generated by CGEN. */
        case SPEAR2_32_OPERAND_SADDR10_PCREL:
            /* The relocation type BFD_RELOC_SPEAR2_32_PCREL_10 is mapped to
               the R_SPEAR2_32_PCREL_10 relocation type in elf32-spear2_32.c. */
            type = BFD_RELOC_SPEAR2_32_PCREL_10;
            fixP->fx_pcrel = 1;
            break;
        ...
```

Table 4.9: `binutils-2.19/gas/config/tc-spear2_32.c` - Relocation Type Lookup

Recall from Section 3.2.5.2 that the instruction-addressable nature of the instruction memory of the SPEAR2 architecture is handled by the linker by means of relocation: before a relocation is finally performed in `spear2_32_final_link_relocate`, defined in `bfd/elf32-spear2_32.c`, the symbol value to be relocated and the relocation addend are adjusted accordingly, if applicable, as shown in Table 4.10.

```
bfd_reloc_status_type spear2_32_final_link_relocate (
    reloc_howto_type *howto,    /* The relocation HOWTO information. */
    bfd *input_bfd,             /* The binary file descriptor the relocation applies to. */
    asection *input_section,    /* The input section the relocation applies to. */
    bfd_byte *contents,         /* The contents of the input section. */
    Elf_Internal_Rela *rel,     /* The relocation data. */
    bfd_vma relocation,         /* The value of the symbol to be relocated. */
    asection *output_section)   /* The output section the relocation applies to. */
{
    /* Check if the symbol belongs to an output section containing instructions. */
    if (output_section != NULL && (output_section->flags & SEC_CODE))
    {
        /* Divide the value of the symbol to be relocated and the relocation addend by factor 2. */
        relocation >>= 1;
        rel->r_addend >>= 1;
    }

    return _bfd_final_link_relocate (howto, input_bfd, input_section, contents, \
                rel->r_offset, relocation, rel->r_addend);
}
```

Table 4.10: `binutils-2.19/bfd/elf32-spear2_32.c` - Final Link Relocation

Concluding, we will discuss how the parameters of the `reloc_howto_type`, as described in Table 4.8, are applied to a relocatable instruction by the relocation handlers provided by the libbfd. As an illustrative example, a simplified form of the `_bfd_final_link_relocate` handler, defined in `bfd/reloc.c`, is shown in Table 4.11.

```
bfd_reloc_status_type _bfd_final_link_relocate (
    reloc_howto_type *howto,   /* The relocation HOWTO information. */
    bfd *input_bfd,            /* The binary file descriptor the relocation applies to. */
    asection *input_section,   /* The input section the relocation applies to. */
    bfd_byte *contents,        /* The contents of the input section. */
    bfd_vma address,           /* The address of the data to be relocated within ’input_section’. */
    bfd_vma value,             /* The value of the symbol to be relocated. */
    bfd_vma addend)            /* The relocation added. */
{
    bfd_byte *location = contents + address;

    /* Compute the relocated symbol value from the value of the symbol plus an addend. */
    bfd_vma relocation = value + addend;

    if (howto->pc_relative)
        /* Subtract the final location of the symbol in the output section. */
        relocation -= (input_section->output_section->vma + input_section->output_offset);

    /* Get the data we are going to relocate from the object file. */
    int size = bfd_get_reloc_size (howto);
    bfd_vma x = bfd_get (input_bfd, location, size);

    /* Put the relocated symbol value into the right bit position. */
    relocation >>= (bfd_vma) howto->rightshift;
    relocation <<= (bfd_vma) howto->bitpos;

    /* Add the relocated symbol value to the right bits in ’x’. */
    x = ((x & ~howto->dst_mask) | (((x & howto->src_mask) + relocation) & howto->dst_mask));

    /* Put the relocated symbol value back into the object file. */
    bfd_put (input_bfd, x, location, size);

    return bfd_reloc_ok;
}
```

Table 4.11: `binutils-2.19/bfd/reloc.c` - Relocation Handler

Further Readings

We recommend the advanced readings in [Red09] and [wik].

4.2.4 Red Hat newlib C Library 1.17

Before we continue to selected parts around the porting of the Red Hat newlib C Library for the SPEAR2 architecture, let us again quickly summarize how it is basically organized. After unpacking the source archive, the user is presented the directories described in Table 4.12.

Directory	Description
`libgloss`	This directory contains code for the creation of *board support packages*, a software library introduced to expose the functionality of the hardware to the C Library, as described in Section 3.1.5. Consequently, any target architecture may maintain multiple board support packages, one for each hardware platform it runs on.
`newlib`	This directory contains the code of the C Library (libc) and the C Math Library (libm), as described in Section 3.1.5, as well as any architecture-specific code in assembly language that these libraries require.

Table 4.12: spear2-newlib-1.17 - Directory Contents

Red Hat newlib C Library vs. GNU C Library

The Red Hat newlib C Library was chosen over the GNU C Library (glibc) for its size: while the glibc is well-suited for PC platforms which do not need to care much about memory utilization, the newlib is designed to be used on small embedded architectures where memory, for being expensive, is still the scarcest resource. Furthermore, the newlib is highly configurable and allows, for example, to disable the support for *long long* and *double* data types, which results in dramatic reductions of code size, at least on systems where floating point operations are emulated in software. We also appreciate the fact that the newlib integrates almost seamlessly into the GNU toolchain and that it can be ported to a bare metal system with little effort only.

In the following we will take a closer look at how certain aspects specific to the SPEAR2 architecture are realized within the *libgloss* and the *newlib* directories of the Red Hat newlib C Library 1.17. Please observe that we have already provided documentation on those parts of the newlib which are of direct interest to the programmer in Section 3.1.5.

4.2.4.1 newlib

Above all, the *libc* requires a new architecture to provide implementations of the `setjmp` and `longjmp` functions in the `newlib/libc/machine/⟨arch⟩/` directory. These functions essentially allow for control transfers beyond the typical procedure call and return sequence and are typically used for the purpose of error handling: a data structure storing a subset of the processor context is used together with `longjmp` to return to a known point of execution where it was created by a call to `setjmp`. Additional architecture-specific code for the libc and the libm can be provided in the respective `machine/` directories.

4.2.4.2 libgloss

The porting of the *libgloss* is centered around the creation of board support packages within the `libgloss/⟨arch⟩/` directory. This directory must contain an implementation of the *C Runtime (crt0)*, as well as the system call implementations for each board support package, as described in Section 3.1.5. The implementation of the crt0 for the SPEAR2 32-bit architecture, in `libgloss/spear2_32/crt0.S`, is described in Table 4.13.

```
    .file "crt0.S"
    .section .text
    .global _start
    .type _start, @function
_start: ; the linker script defines _start as the application entry point
    ; clear the bss section (’__bss_start’ and ’_end’ are provided through the linker script)
    LOAD_SYM_TO_REG r5 __bss_start    ; store symbol ’__bss_start’ in r5
    LOAD_SYM_TO_REG r6 _end           ; store symbol ’_end’ in r6
    cmp_eq  r5, r6                    ; check if address in r5 equals address in r6 (.bss empty)
    jmpi_ct .bss_clear

    ldli    r7, 0                     ; load value 0 into r7
.bss_clear_loop:
    stw r7, r5                        ; store value in r7 at address in r5
    addi    r5, 4                     ; increase address in r5 by the word size in bytes
    cmp_lt  r5, r6                    ; check if address in r5 is less than address in r6
    jmpi_ct .bss_clear_loop

.bss_clear:
    ; clear hardware registers
    CLEAR_REG r0
   ...
    CLEAR_REG r15
    ; clear memory-mapped extension module registers
    CLEAR_MEM_REG -24                 ; frame pointer W register
    CLEAR_MEM_REG -20                 ; frame pointer X register

    ; initialize the FP and SP register to the beginning of the stack
    ; (’__stack’ is provided through the linker script)
    LOAD_SYM_TO_REG r13 __stack       ; store symbol ’__stack’ in r13
    STORE_MEM_REG r13 -16             ; frame pointer register Y (FP)
    STORE_MEM_REG r13 -12             ; frame pointer register Z (SP)

    ; call main
    LOAD_SYM_TO_REG r13 main          ; store symbol ’main’ in r13
    jsr r13

    ; call _exit (’_exit’ is provided by the libc and loops indefinitely)
    mov r1, r0                        ; move return value of ’main’ in r0 to argument register r1
    LOAD_SYM_TO_REG r13 _exit         ; store symbol ’_exit’ in r13
    jsr r13
    .size _start, .-_start
```

Table 4.13: `newlib-1.17/libgloss/spear2_32/crt0.S` - C Runtime (crt0)

Further Readings

We recommend the advanced readings in [Ben10], [Gat], [Und] and [Lea].

4.2.5 GNU Debugger 6.8

The GDB was available as a functional port for the SPEAR2 32-bit architecture variant only, as described in Section 4.1.4. However, its reliance on the deprecated *stabs* debugging information format together with the poor documentation that came with the port made it impossible for us to make the GDB available for the SPEAR2 16-bit architecture with manageable effort. After considering the remaining deficiencies in the *sim* instruction set simulator, we decided on a re-implementation of the GDB at version 6.8 for both SPEAR2 16-bit and 32-bit architecture variants.

Before we turn to some details on the porting of the GNU Debugger for the SPEAR2 architecture, let us again quickly summarize how it is basically organized. After unpacking the source archive, the user is presented the directories described in Table 4.14.

Directory	**Description**
`bfd`	This directory contains the *Binary File Descriptor library* (libbfd), an object file library which allows applications to operate on a variety of object file formats through a common API. The libbfd implements the *application binary interface* in terms of the supported object file formats and relocation types of a particular architecture, and is used by the assembler, linker, binary utilities and the GNU Debugger project alike.
`gdb`	This directory contains the code of the GNU Debugger. The GDB implements the *application binary interface* in terms of data representations, the register usage conventions, procedure calling conventions and the supported debugging information format of a particular target architecture.
`include`	This directory contains common header files. Architecture-specific header files are located in the subdirectories.
`intl`	This directory contains libintl, an internationalization library.
`libdecnumber`	This directory contains libdecnumber, a library for decimal number arithmetics.
`libiberty`	This directory contains libiberty, a collection of functions used by several GNU programs, such as `getopt`, `strerror`, `strtol` and `strtoul`.
`opcodes`	This directory contains the *opcodes library* (libopcodes), containing information on how to assemble and disassemble instructions. The libopcodes implements a subset of the *instruction set architecture* of a particular architecture.
`readline`	This directory contains the GNU Readline library.
`sim`	This directory contains an instruction set simulator. The simulator implements the *instruction set architecture* of a particular target architecture. Architecture-specific source files are located in the subdirectories.

Table 4.14: spear2-gdb-6.8 - Directory Contents

In the following we will take a look at where specifics of the SPEAR2 32-bit architecture variant are realized within the *gdb* and the *sim* directories of the GNU Debugger 6.8. Please note that the GDB uses the *libbfd* and *libopcodes* implementations provided by the GNU Binutils for the SPEAR2 architecture, as described in Section 4.2.3.

4.2.5.1 gdb

The GNU Debugger provides a type `struct gdbarch` to contain architectural information, such as the bit sizes of the basic data types and the general purpose register file, as well as information related to the call stack, in `gdb/⟨arch⟩-tdep.c`. Additionally, a variety of functions has to be implemented to, for example, integrate disassembler functionality from libopcodes, or to enable the GDB to wind itself through the call stack - a key issue in porting the GDB. However, the latter was greatly simplified through the use of the DWARF debugging information format for which support is readily available.

4.2.5.2 sim

The instruction set simulator for a particular architecture is provided in the `sim/⟨arch⟩/` directory and must implement the interface declared in `include/gdb/remote-sim.h`. The *sim* implementation for the SPEAR2 architecture was overall refurbished and modularized to improve both on maintainability and extendability of the code. In the following we will summarize the most evident adaptions made to the instruction set simulator, according to the outcome of the analysis in Section 4.1.4.

- The instruction decoder makes use of the opcode enumeration types of *libopcodes.*
- The simulator makes use of the central register definitions in <`machine/modules.h`>, as described in Section 3.1.5.
- The simulator core implements the memory map in `include/gdb/sim-spear2_32.h`. The map is created by the memory-mapping tool, as outlined in Section 4.1.6.
- An address decoder maps both LMA and VMA addresses to the simulated memory instances, according to the memory map defined in `include/gdb/sim-spear2_32.h`.
- A simulation of the boot memory was implemented as an independent memory block. Since the program counter initially points to this memory, a jump to the instruction memory is automatically performed if no bootloader application is loaded.
- The loading of and communication with simulator plug-ins is encapsulated within a dedicated module.

An overview on the modules of the instruction set simulator for the SPEAR2 32-bit architecture is given in Table 4.15.

File	Description
`spear2_32-bootmem.c/.h`	These files simulate the boot memory.
`spear2_32-codemem.c/.h`	These files simulate the instruction memory.
`spear2_32-datamem.c/.h`	These files simulate the data memory.
`spear2_32-iss.c/.h`	These files implement the simulator core with its instruction fetch, instruction decode, instruction execute and write back stages and provides an interface to the simulated memory instances. The core includes a memory map provided in `include/gdb/sim-spear2_32.h`.
`spear2_32-mad.c/.h`	These files implement a uniform address decoder for LMA and VMA addresses by binding an address to the accessor functions and an offset of a simulated memory instance. The address decoder includes a memory map provided in `include/gdb/sim-spear2_32.h`.
`spear2-op.h`	This file defines the `spear2_op_t` type which is used for instruction decoding.
`spear2_32-plugins.c/.h`	These files implement a generic plug-in type `spear2_plugin_t` and manage the integration of simulator plug-ins.
`spear2_32-sim.c/.h`	These files implement the interface between the simulator core and the GDB.

Table 4.15: `gdb-6.8/sim/spear2_32/` - Instruction Set Simulator Contents

Last but not least, the remote stub implementation was adapted to make use of the central register definitions in <`machine/modules.h`>, as described in Section 3.1.5, but else remains unchanged.

Further Readings

We recommend the advanced readings in [Ben08b] and [Ben08a].

Chapter 5

Conclusion

Porting the GNU toolchain to a new architecture is neither an easy nor a straight forward task. An in-depth knowledge of the underlying architecture is only the most fundamental prerequisite, and several hundred pages of documentation on the internal and external programming interfaces of each member make sure that the learning curve is kept at high level. Unfortunately, the official documentation turns out to be of little use when asked for “the big picture”, let alone, for a systematic porting guide. As a result, it has become common practice to adapt an existing implementation for a similar architecture until it reflects the characteristics of the own architecture. Although this approach indeed serves the creation of a “functional toolchain” in most cases, it is far away from the optimum: it misses a systematic, scientific, methodology and often results in unnecessarily bloated implementations which are neither easy to understand nor to modify [SD07]. In the recent years, many smaller organizations have specialized on the integration of systems software, of which we would like to name the people at *embecosm.com* for sharing their impressing insights on their portings of various members of the GNU Project to the OpenRISC 1000[1] architecture in [Ben08b], [Ben08a], [Ben10].

Despite this, our elaborations on the SPEAR2 hardware/software interface resulted in lasting documentations of what has to be implemented within the software development tools, but not necessarily how this is done. Thereby, our work has laid a solid foundation for the integration of a well-matched toolchain, which has proven to be functional during the Hardware/Software Codesign class held at the Embedded Computing Systems Group of the Department of Computer Engineering at the Vienna University of Technology in 2010.

We hope that this work may serve as a starting point in order to have support for the architecture continued, as for example: continuous maintenance of the GNU toolchain or the porting of the Linux kernel, certainly a different beast, which would require the extension of the application binary interface by a syscall interface, as well as support for dynamic linking to be integrated into the recent GNU Binutils port. Current plans are to have all sources made publicly available under an open source license on *opencores.org*.

[1] The OpenRISC 1000 Architecture, http://opencores.org/project,or1k

Appendix A

SPEAR2 Instruction Set

A.1 Characteristics

A.1.1 Architecture Classification

The SPEAR2 embodies a classical *load/store architecture*, or *register-register architecture*, which comes with little surprise though: maintaining a register-register architecture is a key feature of the *reduced instruction set computer* (RISC) paradigm, which demands that arithmetic and logical operations may only be executed upon registers or immediate values. Therefore, operands which reside in memory must first be loaded into registers before an arithmetic or logical operation can be applied. The result of the operation will as well be written to a register from where it can later be stored back in memory, as depicted in Figure A.1 [Nur07, Chapter 2], [DAP06, Section 2.2]. As a consequence, in a load/store architecture, the memory holding the operands, i.e., the data memory, may only be accessed via dedicated *load/store instructions*, as described in Section A.2.7.

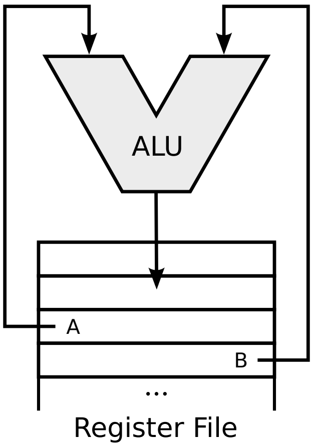


Figure A.1: Load/Store Architecture

A.1.2 Addressing Modes

The addressing modes of an architecture define how the processor calculates the *effective address* of an instruction operand in memory. The modes by which the memories of the SPEAR2 architecture can be addressed are presented in Table A.1.

Mode	**Description**	**Instruction Memory**	**Data Memory**
Immediate	Used for constants	-	-
Register	Used for values in a register	-	-
Absolute	Used for address constants	N	N
Register indirect	Used for address values in a register	Y	Y
FP-relative	Used for displacement values to be added to a frame pointer	N	Y
PC-relative	Used for displacement values to be added to the program counter	Y	N

Table A.1: SPEAR2 Memory Addressing Modes

A.1.3 Addressing Quantities

In modern architectures, memories are typically accessed in quantities of integer multiples of a byte, where the quantities are usually related to the *word size* of the architecture. The quantities by which the data memory of the SPEAR2 architecture can be addressed are presented in Table A.2.

Quantitiy	**Size** [bits]	
	16-bit	32-bit
byte	8	8
halfword	16	16
word	-	32

Table A.2: SPEAR2 Data Memory Addressing Quantities

Please bear in mind that the names of the quantities in the SPEAR2 instruction set are always related to the word size of the 32-bit architecture variant. Therefore, a word access by the 16-bit architecture variant is referred to as a halfword access in the instruction set.

A.1.4 Instruction Coding

The SPEAR2 architecture employs a *fixed length instruction coding* of 16 bits with the instruction formats illustrated in Table A.2.

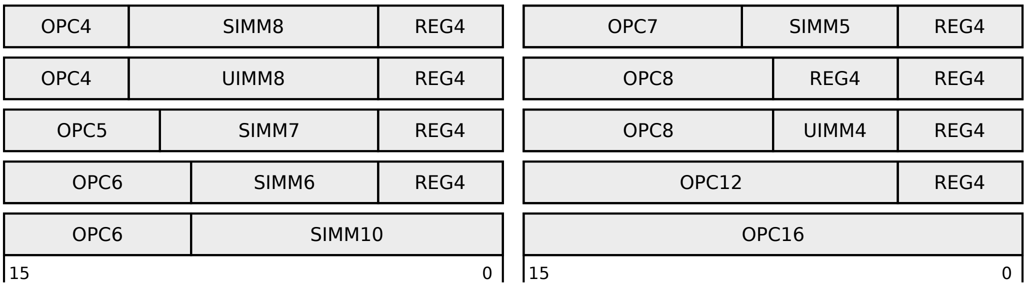


Figure A.2: SPEAR2 Instruction Formats

A.2 Instructions

In the following, the instruction set of the SPEAR2 architecture is presented. The symbols used to explain the instructions are described in Table A.3.

Symbol	Description
+	Addition operator
-	Subtraction operator
$>$	Greater than operator
$<$	Less than operator
=	Equality operator
$:=$	Assignment operator
|	Bitwise logical or operator
&	Bitwise logical and operator
ˆ	Bitwise exclusive or operator
˜	Bitwise complement operator
$\ll$	Bitwise shift left operator
$\gg$	Bitwise shift right operator
(int)	Casts the operand to a sign-extended integer of natural width
(int$X$)	Casts the operand to a sign-extended integer of X bits in width
(uint)	Casts the operand to a non sign-extended integer of natural width
(uint$X$)	Casts the operand to a non sign-extended integer of X bits in width
carry	The *carry flag*
cond	The *condition flag*
opc$X$	An opcode of X bits in width
rX, rY	A general purpose register operand
simm$X$	A signed immediate operand of X bits in width
uimm$X$	An unsigned immediate operand of X bits in width
mem(α)	A memory access at address α of natural width
mem$_X$(α)	A memory access at address α of X bits in width

Table A.3: SPEAR2 Instruction Set - Explanation of Symbols

A.2.1 Arithmetic/Logical Instructions

The arithmetic/logical instructions provide arithmetic and logical operations on integers.

Add (add)

OPC8								REG4				REG4			
1	1	1	0	0	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	add rX, rY	carry := ((int64)rX + (int64)rY) ≫ 32
		rX := rX + rY

16-bit	add rX, rY	carry := ((int32)rX + (int32)rY) ≫ 16
		rX := rX + rY

Add if cond-flag false (add_cf)

OPC8								REG4				REG4			
1	1	0	0	0	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
32-bit	add_cf rX, rY	if (!cond) carry := ((int64)rX + (int64)rY) ≫ 32 rX := rX + rY
16-bit	add_cf rX, rY	if (!cond) carry := ((int32)rX + (int32)rY) ≫ 16 rX := rX + rY

Add if cond-flag true (add_ct)

OPC8								REG4				REG4			
1	1	0	1	0	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
32-bit	add_ct rX, rY	if (cond) carry := ((int64)rX + (int64)rY) ≫ 32 rX := rX + rY
16-bit	add_ct rX, rY	if (cond) carry := ((int32)rX + (int32)rY) ≫ 16 rX := rX + rY

Add immediate (addi)

OPC6						SIMM6						REG4			
1	0	1	0	1	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`addi rX, simm6`	carry := ((int64)rX + (int64)simm6) $\gg$ 32
		rX := (int)rX + (int)simm6

16-bit	`addi rX, simm6`	carry := ((int32)rX + (int32)simm6) $\gg$ 16
		rX := (int)rX + (int)simm6

Add immediate if cond-flag false (addi_cf)

OPC6						SIMM6						REG4			
1	0	0	0	1	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`addi_cf rX, simm6`	if (!cond)
		carry := ((int64)rX + (int64)simm6) $\gg$ 32
		rX := (int)rX + (int)simm6

16-bit	`addi_cf rX, simm6`	if (!cond)
		carry := ((int32)rX + (int32)simm6) $\gg$ 16
		rX := (int)rX + (int)simm6

Add immediate if cond-flag true (addi_ct)

OPC6						SIMM6						REG4			
1	0	0	1	1	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`addi_ct rX, simm6`	if (cond)
		carry := ((int64)rX + (int64)simm6) $\gg$ 32
		rX := (int)rX + (int)simm6

16-bit	`addi_ct rX, simm6`	if (cond)
		carry := ((int32)rX + (int32)simm6) $\gg$ 16
		rX := (int)rX + (int)simm6

Add with carry (addc)

OPC8								REG4				REG4			
1	1	1	0	0	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`addc rX, rY`	carry’ := ((int64)rX + (int64)rY) ≫ 32
		rX := rX + rY + carry
		carry := carry’

16-bit	`addc rX, rY`	carry’ := ((int32)rX + (int32)rY) ≫ 16
		rX := rX + rY + carry
		carry := carry’

Add with carry if cond-flag false (addc_cf)

OPC8								REG4				REG4			
1	1	0	0	0	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
32-bit	`addc rX, rY`	if (!cond) carry’ := ((int64)rX + (int64)rY) ≫ 32 rX := rX + rY + carry carry := carry’
16-bit	`addc rX, rY`	if (!cond) carry’ := ((int32)rX + (int32)rY) ≫ 16 rX := rX + rY + carry carry := carry’

Add with carry if cond-flag true (addc_ct)

OPC8								REG4				REG4			
1	1	0	1	0	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
32-bit	`addc rX, rY`	if (cond) carry’ := ((int64)rX + (int64)rY) ≫ 32 rX := rX + rY + carry carry := carry’
16-bit	`addc rX, rY`	if (cond) carry’ := ((int32)rX + (int32)rY) ≫ 16 rX := rX + rY + carry carry := carry’

Bitwise logical and (and)

OPC8								REG4				REG4			
1	1	1	0	0	1	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
16/32-bit	`and rX, rY`	`rX := rX & rY`

Bitwise logical and if cond-flag false (and_cf)

OPC8								REG4				REG4			
1	1	0	0	0	1	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`and_cf rX, rY`	`if (!cond)`
		`rX := rX & rY`

Bitwise logical and if cond-flag true (and_ct)

OPC8								REG4				REG4			
1	1	0	1	0	1	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`and_ct rX, rY`	`if (cond)`
		`rX := rX & rY`

Bit clear (bclr)

OPC7							UIMM5					REG4			
1	0	1	0	0	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

OPC8								UIMM4				REG4			
1	0	1	0	0	1	1	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
32-bit	`bclr rX, uimm5`	`rX := rX & ~(1 ≪ uimm5)`
16-bit	`blcr rX, uimm4`	`rX := rX & ~(1 ≪ uimm4)`

Bit clear if cond-flag false (bclr_cf)

OPC7							UIMM5					REG4			
1	0	0	0	0	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

OPC8								UIMM4				REG4			
1	0	0	0	0	1	1	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`bclr_cf rX, uimm5`	`if (!cond)`
		`rX := rX & ~(1 ≪ uimm5)`

16-bit	`blcr_cf rX, uimm4`	`if (!cond)`
		`rX := rX & ~(1 ≪ uimm4)`

Bit clear if cond-flag true (bclr_ct)

OPC7							UIMM5					REG4			
1	0	0	1	0	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

OPC8								UIMM4				REG4			
1	0	0	1	0	1	1	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`bclr_ct rX, uimm5`	`if (!cond)`
		`rX := rX & ~(1 ≪ uimm5)`

16-bit	`blcr_ct rX, uimm4`	`if (!cond)`
		`rX := rX & ~(1 ≪ uimm4)`

Bit set (bset)

OPC7							UIMM5					REG4			
1	0	1	0	0	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

OPC8								UIMM4				REG4			
1	0	1	0	0	1	0	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
32-bit	`bset rX, uimm5`	`rX := rX | (1 ≪ uimm5)`
16-bit	`bset rX, uimm4`	`rX := rX | (1 ≪ uimm4)`

Bit set if cond-flag false (bset_cf)

OPC7							UIMM5					REG4			
1	0	0	0	0	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

OPC8								UIMM4				REG4			
1	0	0	0	0	1	0	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`bset_cf rX, uimm5`	`if (!cond)`
		`rX := rX | (1 ≪ uimm5)`

16-bit	`bset_cf rX, uimm4`	`if (!cond)`
		`rX := rX | (1 ≪ uimm4)`

Bit set if cond-flag true (bset_ct)

OPC7							UIMM5					REG4			
1	0	0	1	0	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

OPC8								UIMM4				REG4			
1	0	0	1	0	1	0	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`bset_ct rX, uimm5`	`if (!cond)`
		`rX := rX | (1 ≪ uimm5)`

16-bit	`bset_ct rX, uimm4`	`if (!cond)`
		`rX := rX | (1 ≪ uimm4)`

Bit test (btest)

OPC7							UIMM5					REG4			
1	0	1	1	0	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

OPC8								UIMM4				REG4			
1	0	1	1	0	1	1	0	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
32-bit	`btest rX, uimm5`	cond := (rX & (1 ≪ uimm5)) != 0
16-bit	`btest rX, uimm4`	cond := (rX & (1 ≪ uimm4)) != 0

Bitwise logical exclusive or (eor)

OPC8								REG4				REG4			
1	1	1	0	0	1	1	1	S_3	S_2	S_1	S_0	R_3	R_2	R_1	R_0
15															0

Architecture	Assembly Code	Description
16/32-bit	`eor rX, rY`	rX := rX ˆ rY

Bitwise logical exclusive or if cond-flag false (eor_cf)

OPC8								REG4				REG4			
1	1	0	0	0	1	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`eor_cf rX, rY`	if (!cond)
		rX := rX ˆ rY

Bitwise logical exclusive or if cond-flag true (eor_ct)

OPC8								REG4				REG4			
1	1	0	1	0	1	1	1	S_3	S_2	S_1	S_0	R_3	R_2	R_1	R_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`eor_ct rX, rY`	if (cond)
		rX := rX ˆ rY

Negative (neg)

OPC12												REG4			
0	0	0	0	0	0	0	0	0	0	0	0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
16/32-bit	`neg rX`	`rX := ~rX + 1`

Negative if cond-flag false (neg_cf)

OPC12												REG4			
1	1	0	0	1	1	0	1	0	0	0	0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`neg_cf rX`	`if (!cond)`
		`rX := ~rX + 1`

Negative if cond-flag true (neg_ct)

OPC12												REG4			
1	1	0	1	1	1	0	1	0	0	0	0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`neg_ct rX`	`if (cond)`
		`rX := ~rX + 1`

Bitwise logical not (not)

OPC12												REG4			
1	1	1	0	1	1	0	0	0	0	0	0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
16/32-bit	`not rX`	`rX := ~rX`

Bitwise logical not if cond-flag false (not_cf)

OPC12												REG4			
1	1	0	0	1	1	0	0	0	0	0	0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`not_cf rX`	`if (!cond)`
		`rX := ~rX`

Bitwise logical not if cond-flag true (not_ct)

OPC12												REG4			
1	1	0	1	1	1	0	0	0	0	0	0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`not_ct rX`	`if (cond)`
		`rX := ~rX`

Bitwise logical or (or)

OPC8								REG4				REG4			
1	1	1	0	0	1	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
16/32-bit	`or rX, rY`	`rX := rX | rY`

Bitwise logical or if cond-flag false (or_cf)

OPC8								REG4				REG4			
1	1	0	0	0	1	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`or_cf rX, rY`	`if (!cond)`
		`rX := rX | rY`

Bitwise logical or if cond-flag true (and_ct)

OPC8								REG4				REG4			
1	1	0	1	0	1	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`or_ct rX, rY`	`if (cond)`
		`rX := rX | rY`

Rotate right with carry (rrc)

OPC12												REG4			
1	1	1	0	1	0	1	0	0	0	0	0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`rrc rX`	carry' := rX & 1
		rX := rX ≫ 1
		rX := rX | carry ≪ 31
		carry := carry'

16-bit	`rrc rX`	carry' := rX & 1
		rX := rX ≫ 1
		rX := rX | carry ≪ 15
		carry := carry'

Rotate right with carry if cond-flag false (rrc_cf)

OPC12												REG4			
1	1	0	0	1	0	1	0	0	0	0	0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`rrc_cf rX`	if (!cond)
		carry' := rX & 1
		rX := rX ≫ 1
		rX := rX | carry ≪ 31
		carry := carry'

16-bit	`rrc_cf rX`	if (!cond)
		carry' := rX & 1
		rX := rX ≫ 1
		rX := rX | carry ≪ 15
		carry := carry'

Rotate right with carry if cond-flag true (rrc_ct)

OPC12												REG4			
1	1	0	1	1	0	1	0	0	0	0	0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`rrc_ct rX`	if (cond)
		carry' := rX & 1
		rX := rX ≫ 1
		rX := rX | carry ≪ 31
		carry := carry'

16-bit	`rrc_ct rX`	if (cond)
		carry' := rX & 1
		rX := rX ≫ 1
		rX := rX | carry ≪ 15
		carry := carry'

Shift left (sl)

OPC8								REG4				REG4			
1	0	1	0	0	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`sl rX, rY`	`carry := (rX ≫ (32 - rY)) & 1`
		`rX := rX ≪ (rY & 0x1F)`

16-bit	`sl rX, rY`	`carry := (rX ≫ (16 - rY)) & 1`
		`rX := rX ≪ (rY & 0x0F)`

Shift left if cond-flag false (sl_cf)

OPC8								REG4				REG4			
1	0	0	0	0	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`sl_cf rX, rY`	`if (!cond)`
		`carry := (rX ≫ (32 - rY)) & 1`
		`rX := rX ≪ (rY & 0x1F)`

16-bit	`sl_cf rX, rY`	`if (!cond)`
		`carry := (rX ≫ (16 - rY)) & 1`
		`rX := rX ≪ (rY & 0x0F)`

Shift left if cond-flag true (sl_ct)

OPC8								REG4				REG4			
1	0	0	1	0	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`sl_ct rX, rY`	`if (cond)`
		`carry := (rX ≫ (32 - rY)) & 1`
		`rX := rX ≪ (rY & 0x1F)`

16-bit	`sl_ct rX, rY`	`if (cond)`
		`carry := (rX ≫ (16 - rY)) & 1`
		`rX := rX ≪ (rY & 0x0F)`

Shift left immediate (sli)

OPC8								UIMM4				REG4			
1	0	1	0	0	0	0	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`sli rX, uimm4`	`carry := (rX ≫ (32 - uimm4)) & 1`
		`rX := rX ≪ uimm4`

16-bit	`sli rX, uimm4`	`carry := (rX ≫ (16 - uimm4)) & 1`
		`rX := rX ≪ uimm4`

Shift left immediate if cond-flag false (sli_cf)

OPC8								UIMM4				REG4			
1	0	0	0	0	0	0	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`sli_cf rX, uimm4`	`if (!cond)`
		`carry := (rX ≫ (32 - uimm4)) & 1`
		`rX := rX ≪ uimm4`

16-bit	`sli_cf rX, uimm4`	`if (!cond)`
		`carry := (rX ≫ (16 - uimm4)) & 1`
		`rX := rX ≪ uimm4`

Shift left immediate if cond-flag true (sli_ct)

OPC8								UIMM4				REG4			
1	0	0	1	0	0	0	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`sli_ct rX, uimm4`	`if (cond)`
		`carry := (rX ≫ (32 - uimm4)) & 1`
		`rX := rX ≪ uimm4`

16-bit	`sli_ct rX, uimm4`	`if (cond)`
		`carry := (rX ≫ (16 - uimm4)) & 1`
		`rX := rX ≪ uimm4`

Shift right (sr)

OPC8								REG4				REG4			
1	0	1	0	0	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`sr rX, rY`	`if (cond)`
		`carry := rX & (1 ≪ (rY - 1))`
		`rX := rX ≫ (rY & 0x1F)`

16-bit	`sr rX, rY`	`if (cond)`
		`carry := rX & (1 ≪ (rY - 1))`
		`rX := rX ≫ (rY & 0x0F)`

Shift right if cond-flag false (sr_cf)

OPC8								REG4				REG4			
1	0	0	0	0	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`sr_cf rX, rY`	`if (!cond)`
		`carry := rX & (1 ≪ (rY - 1))`
		`rX := rX ≫ (rY & 0x1F)`

16-bit	`sr_cf rX, rY`	`if (!cond)`
		`carry := rX & (1 ≪ (rY - 1))`
		`rX := rX ≫ (rY & 0x0F)`

Shift right if cond-flag true (sr_ct)

OPC8								REG4				REG4			
1	0	0	1	0	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`sr_ct rX, rY`	`if (cond)`
		`carry := rX & (1 ≪ (rY - 1))`
		`rX := rX ≫ (rY & 0x1F)`

16-bit	`sr_ct rX, rY`	`if (cond)`
		`carry := rX & (1 ≪ (rY - 1))`
		`rX := rX ≫ (rY & 0x0F)`

Shift right arithmetic (sra)

OPC8								REG4				REG4			
1	1	1	0	1	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`sra rX, rY`	`carry := rX & (1 ≪ (rY - 1))`
		`rX := (int)rX ≫ (rY & 0x1F)`

16-bit	`sra rX, rY`	`carry := rX & (1 ≪ (rY - 1))`
		`rX := (int)rX ≫ (rY & 0x1F)`

Shift right arithmetic if cond-flag false (sra_cf)

OPC8								REG4				REG4			
1	1	0	0	1	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`sra_cf rX, rY`	`if (!cond)`
		`carry := rX & (1 ≪ (rY - 1))`
		`rX := (int)rX ≫ (rY & 0x1F)`

16-bit	`sra_cf rX, rY`	`if (!cond)`
		`carry := rX & (1 ≪ (rY - 1))`
		`rX := (int)rX ≫ (rY & 0x1F)`

Shift right arithmetic if cond-flag true (sra_ct)

OPC8								REG4				REG4			
1	1	0	1	1	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`sra_ct rX, rY`	`if (cond)`
		`carry := rX & (1 ≪ (rY - 1))`
		`rX := (int)rX ≫ (rY & 0x1F)`

16-bit	`sra_ct rX, rY`	`if (cond)`
		`carry := rX & (1 ≪ (rY - 1))`
		`rX := (int)rX ≫ (rY & 0x1F)`

Shift right arithmetic immediate (srai)

OPC8								UIMM4				REG4			
1	1	1	0	1	0	0	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`srai rX, uimm4`	`carry := rX & (1 ≪ (uimm4 - 1))`
		`rX := (int)rX ≫ uimm4`

Shift right arithmetic immediate if cond-flag false (srai_cf)

OPC8								UIMM4				REG4			
1	1	0	0	1	0	0	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`srai_cf rX, uimm4`	`if (!cond)`
		` carry := rX & (1 ≪ (uimm4 - 1))`
		` rX := (int)rX ≫ uimm4`

Shift right arithmetic immediate if cond-flag true (srai_ct)

OPC8								UIMM4				REG4			
1	1	0	1	1	0	0	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`srai_ct rX, uimm4`	`if (cond)`
		` carry := rX & (1 ≪ (uimm4 - 1))`
		` rX := (int)rX ≫ uimm4`

Shift right immediate (sri)

OPC8								UIMM4				REG4			
1	0	1	0	0	0	1	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`sri rX, uimm4`	`carry := rX & (1 ≪ (uimm4 - 1))`
		`rX := rX ≫ uimm4`

Shift right immediate if cond-flag false (sri_cf)

OPC8								UIMM4				REG4			
1	0	0	0	0	0	1	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`sri_cf rX, uimm4`	`if (!cond)`
		`carry := rX & (1 ≪ (uimm4 - 1))`
		`rX := rX ≫ uimm4`

Shift right immediate if cond-flag true (sri_ct)

OPC8								UIMM4				REG4			
1	1	0	1	1	0	0	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`sri_ct rX, uimm4`	`if (cond)`
		`carry := rX & (1 ≪ (uimm4 - 1))`
		`rX := rX ≫ uimm4`

Subtract (sub)

OPC8								REG4				REG4			
1	1	1	0	0	0	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`sub rX, rY`	`carry := ((int64)rX - (int64)rY) ≫ 32`
		`rX := rX - rY`

16-bit	`sub rX, rY`	`carry := ((int32)rX - (int32)rY) ≫ 16`
		`rX := rX - rY`

Subtract if cond-flag false (sub_cf)

OPC8								REG4				REG4			
1	1	0	0	0	0	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`sub_cf rX, rY`	`if (!cond)`
		` carry := ((int64)rX - (int64)rY) ≫ 32`
		` rX := rX - rY`

16-bit	`sub_cf rX, rY`	`if (!cond)`
		` carry := ((int32)rX - (int32)rY) ≫ 16`
		` rX := rX - rY`

Subtract if cond-flag true (sub_ct)

OPC8								REG4				REG4			
1	1	0	1	0	0	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`sub_ct rX, rY`	`if (cond)`
		` carry := ((int64)rX - (int64)rY) ≫ 32`
		` rX := rX - rY`

16-bit	`sub_ct rX, rY`	`if (cond)`
		` carry := ((int32)rX - (int32)rY) ≫ 16`
		` rX := rX - rY`

Subtract with carry (subc)

OPC8								REG4				REG4			
1	1	1	0	0	1	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`subc rX, rY`	`carry’ := ((int64)rX - (int64)rY) ≫ 32`
		`rX := rX - rY + carry`
		`carry := carry’`

16-bit	`subc rX, rY`	`carry’ := ((int32)rX - (int32)rY) ≫ 16`
		`rX := rX - rY + carry`
		`carry := carry’`

Subtract with carry if cond-flag false (subc_cf)

OPC8								REG4				REG4			
1	1	0	0	0	1	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`subc_cf rX, rY`	`if (!cond)`
		` carry’ := ((int64)rX - (int64)rY) ≫ 32`
		` rX := rX - rY + carry`
		` carry := carry’`

16-bit	`subc_cf rX, rY`	`if (!cond)`
		` carry’ := ((int32)rX - (int32)rY) ≫ 16`
		` rX := rX - rY + carry`
		` carry := carry’`

Subtract with carry if cond-flag true (subc_ct)

OPC8								REG4				REG4			
1	1	0	1	0	1	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

32-bit	`subc_ct rX, rY`	`if (cond)`
		` carry’ := ((int64)rX - (int64)rY) ≫ 32`
		` rX := rX - rY + carry`
		` carry := carry’`

16-bit	`subc_ct rX, rY`	`if (cond)`
		` carry’ := ((int32)rX - (int32)rY) ≫ 16`
		` rX := rX - rY + carry`
		` carry := carry’`

A.2.2 Comparison Instructions

The comparison instructions provide comparison operations on integers.

Compare equal (cmp_eq)

OPC8								REG4				REG4			
1	0	1	1	0	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
16/32-bit	`cmp_eq rX, rY`	`cond := ((int)rX = (int)rY`

Compare greater than (cmp_gt)

OPC8								REG4				REG4			
1	0	1	1	0	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
16/32-bit	`cmp_gt rX, rY`	`cond := ((int)rX > (int)rY`

Compare less than (cmp_lt)

OPC8								REG4				REG4			
1	0	1	1	0	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
16/32-bit	`cmp_lt rX, rY`	`cond := ((int)rX < (int)rY`

Compare equal immediate (cmpi_eq)

OPC5					SIMM7							REG4			
1	0	1	1	1	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
16/32-bit	`cmpi_eq rX, simm7`	`cond := ((int)rX = (int)simm7)`

Compare greater than immediate (cmpi_gt)

OPC5					SIMM7							REG4			
0	0	1	1	1	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
16/32-bit	`cmpi_gt rX, simm7`	`cond` := `((int)rX` > `(int)simm7)`

Compare less than immediate (cmpi_lt)

OPC5					SIMM7							REG4			
0	0	1	1	0	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
16/32-bit	`cmpi_lt rX, simm7`	`cond` := `((int)rX` < `(int)simm7)`

Compare greater than (cmpu_gt)

OPC8								REG4				REG4			
1	0	1	1	0	1	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
16/32-bit	`cmpu_gt rX, rY`	`cond` := `((uint)rX` > `(uint)rY`

Compare less than unsigned (cmpu_lt)

OPC8								REG4				REG4			
1	0	1	1	0	0	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
16/32-bit	`cmpu_lt rX, rY`	`cond` := `((uint)rX` < `(uint)rY`

A.2.3 Constant Manipulating Instructions

The constant manipulating instructions provide operations for the loading of constants into registers.

Load high byte immediate (ldhi)

OPC4				SIMM8								REG4			
0	0	0	1	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Relocation Type
32-bit	`ldhi rX, simm8`	`rX := rX | ((int)simm8 ≪ 8)`	R_SPEAR2_32_(LO|HI|3RD|4TH)
16-bit	`ldhi rX, simm8`	`rX := rX | ((int)simm8 ≪ 8)`	R_SPEAR2_16_(LO|HI)

Load low byte immediate (ldli)

OPC4				SIMM8								REG4			
0	0	0	0	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Relocation Type
32-bit	`ldli rX, simm8`	`rX := rX | ((int)simm8 & 0xFF)`	R_SPEAR2_32_(LO|HI|3RD|4TH)
16-bit	`ldli rX, simm8`	`rX := rX | ((int)simm8 & 0xFF)`	R_SPEAR2_16_(LO|HI)

Load low byte immediate without sign extension (ldliu)

OPC4				UIMM8								REG4			
0	0	1	0	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Relocation Type
32-bit	`ldliu rX, uimm8`	`rX := rX | ((uint)simm8 & 0xFF)`	R_SPEAR2_32_(LO|HI|3RD|4TH)
16-bit	`ldliu rX, uimm8`	`rX := rX | ((uint)simm8 & 0xFF)`	R_SPEAR2_16_(LO|HI)

The SPEAR2 architecture requires a sequence of instructions to be executed in order to load the value of a symbol, a mnemonic identifying an object in memory, into a register. The intended use, a combination of constant manipulating instructions and the assembly language instructions *lo()*, *hi()*, *3rd()*, *4th()*, which extract the n-th byte from an n-byte argument, is presented in Table A.4.

```
; load value of symbol 'handler' into r13
ldhi  r13, 4th(handler)
ldliu r13, 3rd(handler)
sli   r13, 8
ldliu r13, hi(handler)
sli   r13, 8
ldliu r13, lo(handler)
```

Table A.4: spear2_32-none-eabi-as Assembly Code Sample: Load Symbol into Register

A.2.4 Control Transfer Instructions

The control transfer instructions provide operations for altering the control flow.

Jump (jmp)

OPC12												REG4			
1	1	1	0	1	1	1	1	0	0	0	0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode	Relocation Type
16/32-bit	`jmp rX`	`PC := rX`	register indirect	none

Jump if cond-flag false (jmp_cf)

OPC12												REG4			
1	1	0	0	1	1	1	1	0	0	0	0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode	Relocation Type

16/32-bit	`jmp rX`	`if (!cond)`
		`PC := rX`
			register indirect	none

Jump if cond-flag true (jmp_ct)

OPC12												REG4			
1	1	0	1	1	1	1	1	0	0	0	0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode	Relocation Type

16/32-bit	`jmp_ct rX`	`if (cond)`
		`PC := rX`
			register indirect	none

Jump immediate (jmpi)

OPC6						SIMM10									
1	0	1	0	1	1	I_9	I_8	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
15															0

Architecture	Assembly Code	Description	Addressing Mode	Relocation Type
16/32-bit	`jmpi simm10`	`PC := PC + simm10`	PC-relative	R_SPEAR2_(16|32)_PCREL_10

Jump immediate if cond-flag false (jmpi_cf)

OPC6						SIMM10									
1	0	0	0	1	1	I_9	I_8	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
15															0

Architecture	Assembly Code	Description	Addressing Mode	Relocation Type

16/32-bit	`jmpi_cf simm10`	`if (!cond)`
		`PC := PC + simm10`
			PC-relative	R_SPEAR2_(16|32)_PCREL_10

Jump immediate if cond-flag true (jmpi_ct)

OPC6						SIMM10									
1	0	0	1	1	1	I_9	I_8	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
15															0

Architecture	Assembly Code	Description	Addressing Mode	Relocation Type

16/32-bit	`jmpi_ct simm10`	`if (cond)`
		`PC := PC + simm10`
			PC-relative	R_SPEAR2_(16|32)_PCREL_10

Jump to subroutine (jsr)

OPC12												REG4			
1	1	1	0	1	1	1	0	0	0	0	0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode	Relocation Type

16/32-bit	`jsr rX`	`r14 := PC`
		`PC := rX`
			register indirect	none

Jump to subroutine if cond-flag false (jsr_cf)

OPC12												REG4			
1	1	0	0	1	1	1	0	0	0	0	0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode	Relocation Type

16/32-bit	`jsr_cf rX`	`if (!cond)`
		` r14 := PC`
		` PC := rX`
			register indirect	none

Jump to subroutine if cond-flag true (jsr_ct)

OPC12												REG4			
1	1	0	1	1	1	1	0	0	0	0	0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode	Relocation Type

16/32-bit	`jsr_ct rX`	`if (cond)`
		` r14 := PC`
		` PC := rX`
			register indirect	none

Return from subroutine (rts)

OPC16															
1	1	1	1	1	0	0	0	0	0	0	0	1	1	1	0
15															0

Architecture	Assembly Code	Description

16/32-bit	`rts`	`status := status'`
		`PC := r14`

A.2.5 Data Movement Instructions

The data movement instructions provide operations for moving data between registers.

Move (mov)

OPC8								REG4				REG4			
1	1	1	0	0	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
16/32-bit	`mov rX, rY`	`rX := rY`

Move if cond-flag false (mov_cf)

OPC8								REG4				REG4			
1	1	0	0	0	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`mov_cf rX, rY`	`if (!cond)`
		`rX := rY`

Move if cond-flag true (mov_ct)

OPC8								REG4				REG4			
1	1	0	1	0	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`mov_ct rX, rY`	`if (!cond)`
		`rX := rY`

A.2.6 Exception Instructions

The exception instructions provide operations for maintaining the exception vector table and for handling exceptions.

Illegal Operation (illop)

OPC16															
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
15															0

Architecture	Assembly Code	Description
16/32-bit	`illop`	`Raises illop exception.`

Load Exception Vector (ldvec)

OPC7							IMM5					REG4			
1	1	1	1	1	0	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
16/32-bit	`ldvec_cf rX, simm5`	`rX := EVT(simm5)`

No operation (nop)

OPC16															
1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
15															0

Architecture	Assembly Code	Description
16/32-bit	`nop`	

Return from exception (rte)

OPC16															
1	1	1	1	1	0	0	1	0	0	0	0	1	1	1	1
15															0

Architecture	Assembly Code	Description

16/32-bit	`rte`	`status := status'`
		`PC := r15`

Store Exception Vector (stvec)

OPC7							IMM5					REG4			
1	1	1	1	1	0	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description
16/32-bit	`stvec rX, simm5`	`EVT(simm5) := rX`

Trap (trap)

OPC8								UIMM4				REG4			
1	1	1	0	1	0	1	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`trap rX, uimm4`	`status' := status`
		`r15 := PC`
		`PC := EVT(uimm4)`

Trap if cond-flag false (trap_cf)

OPC8								UIMM4				REG4			
1	1	0	0	1	0	1	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`trap_cf rX, uimm4`	`if (!cond)`
		` status' := status`
		` r15 := PC`
		` PC := EVT(uimm4)`

Trap if cond-flag true (trap_ct)

OPC8								UIMM4				REG4			
1	1	0	1	1	0	1	1	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description

16/32-bit	`trap_ct rX, uimm4`	`if (cond)`
		` status' := status`
		` r15 := PC`
		` PC := EVT(uimm4)`

A.2.7 Load/Store Instructions

The load/store instructions provide operations for loading and storing data from and to memory.

Load byte (ldb)

OPC8								REG4				REG4			
1	1	1	1	0	0	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

16/32-bit	`ldb rX, rY`	`byte :=` $\texttt{mem}_8$`(rY)`
		`rX := (int)byte`
			register indirect

Load byte unsigned (ldbu)

OPC8								REG4				REG4			
1	1	1	1	0	1	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

16/32-bit	`ldbu rX, rY`	`byte :=` $\texttt{mem}_8$`(rY)`
		`rX := (uint)byte`
			register indirect

Load (half-)word with frame pointer W (ldfpw)

OPC6						SIMM6						REG4			
0	1	1	0	0	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode
16/32-bit	`ldfpw rX, simm6`	`rX := mem(FPW + (int)simm6)`	FPW-relative

Load (half-)word with frame pointer W and decrement W (ldfpw_dec)

OPC7							SIMM5					REG4			
0	1	0	0	0	0	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`ldfpw_dec rX, simm5`	`rX := mem(FPW + (int)simm5)`
		`FPW := FPW - 4`
			FPW-relative

16-bit	`ldfpw_dec rX, simm5`	`rX := mem(FPW + (int)simm5)`
		`FPW := FPW - 2`
			FPW-relative

Load (half-)word with frame pointer W and increment W (ldfpw_inc)

OPC7							SIMM5					REG4			
0	1	0	0	0	0	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`ldfpw_inc rX, simm5`	`rX := mem(FPW + (int)simm5)`
		`FPW := FPW + 4`
			FPW-relative

16-bit	`ldfpw_inc rX, simm5`	`rX := mem(FPW + (int)simm5)`
		`FPW := FPW + 2`
			FPW-relative

Load (half-)word with frame pointer X (ldfpx)

OPC6						SIMM6						REG4			
0	1	1	0	0	1	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode
16/32-bit	`ldfpx rX, simm6`	`rX := mem(FPX + (int)simm6)`	FPX-relative

Load (half-)word with frame pointer X and decrement X (ldfpx_dec)

OPC7							SIMM5					REG4			
0	1	0	0	0	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`ldfpx_dec rX, simm5`	`rX := mem(FPX + (int)simm5)`
		`FPX := FPX - 4`
			FPX-relative

16-bit	`ldfpx_dec rX, simm5`	`rX := mem(FPX + (int)simm5)`
		`FPX := FPX - 2`
			FPX-relative

Load (half-)word with frame pointer X and increment X (ldfpw_inc)

OPC7							SIMM5					REG4			
0	1	0	0	0	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`ldfpx_dec rX, simm5`	`rX := mem(FPX + (int)simm5)`
		`FPX := FPX + 4`
			FPX-relative

16-bit	`ldfpx_dec rX, simm5`	`rX := mem(FPX + (int)simm5)`
		`FPX := FPX + 2`
			FPX-relative

Load (half-)word with frame pointer Y (ldfpy)

OPC6						SIMM6						REG4			
0	1	1	0	1	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode
16/32-bit	`ldfpy rX, simm6`	`rX := mem(FPY + (int)simm6)`	FPY-relative

Load (half-)word with frame pointer Y and decrement Y (ldfpy_dec)

OPC7							SIMM5					REG4			
0	1	0	0	1	0	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`ldfpy_dec rX, simm5`	`rX := mem(FPY + (int)simm5)`
		`FPY := FPY - 4`
			FPY-relative

16-bit	`ldfpy_dec rX, simm5`	`rX := mem(FPY + (int)simm5)`
		`FPY := FPY - 2`
			FPY-relative

Load (half-)word with frame pointer Y and increment Y (ldfpy_inc)

OPC7							SIMM5					REG4			
0	1	0	0	1	0	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`ldfpy_dec rX, simm5`	`rX := mem(FPY + (int)simm5)`
		`FPY := FPY + 4`
			FPY-relative

16-bit	`ldfpy_dec rX, simm5`	`rX := mem(FPY + (int)simm5)`
		`FPY := FPY + 2`
			FPY-relative

Load (half-)word with frame pointer Z (ldfpz)

OPC6						SIMM6						REG4			
0	1	1	0	1	1	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode
16/32-bit	`ldfpz rX, simm6`	`rX := mem(FPZ + (int)simm6)`	FPZ-relative

Load (half-)word with frame pointer Z and decrement Z (ldfpz_dec)

OPC7							SIMM5					REG4			
0	1	0	0	1	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`ldfpz_dec rX, simm5`	`rX := mem(FPZ + (int)simm5)`
		`FPZ := FPZ - 4`
			FPZ-relative

16-bit	`ldfpz_dec rX, simm5`	`rX := mem(FPZ + (int)simm5)`
		`FPZ := FPZ - 2`
			FPZ-relative

Load (half-)word with frame pointer Z and increment Z (ldfpz_inc)

OPC7							SIMM5					REG4			
0	1	0	0	1	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`ldfpz_dec rX, simm5`	`rX := mem(FPZ + (int)simm5)`
		`FPZ := FPZ + 4`
			FPZ-relative

16-bit	`ldfpz_dec rX, simm5`	`rX := mem(FPZ + (int)simm5)`
		`FPZ := FPZ + 2`
			FPZ-relative

Load half-word (ldh)

OPC8								REG4				REG4			
1	1	1	1	0	0	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`ldh rX, rY`	hword := mem$_{16}$(rY)
		rX := (int)hword
			register indirect

16-bit	`ldh rX, rY`	rX := mem(rY)	register indirect

Load half-word unsigned (ldhu)

OPC8								REG4				REG4			
1	1	1	1	0	0	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`ldhu rX, rY`	hword := mem$_{16}$(rY)
		rX := (uint)hword
			register indirect

Load word (ldw)

OPC8								REG4				REG4			
1	1	1	1	0	0	0	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode
32-bit	`ldw rX, rY`	rX := mem(rY)	register indirect

Store byte (stb)

OPC8								REG4				REG4			
1	1	1	1	0	1	1	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

16/32-bit	`stb rX, rY`	`byte := rX(7:0)`
		`mem`$_8$`(rY) := byte`
			register indirect

Store (half-)word with frame pointer W (stfpw)

OPC6						SIMM6						REG4			
0	1	1	1	0	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode
16/32-bit	`stfpw rX, simm6`	`mem(FPW + (int)simm6) := rX`	register indirect

Store (half-)word with frame pointer W and decrement W (stfpw_dec)

OPC7							SIMM5					REG4			
0	1	0	1	0	0	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`stfpw_dec rX, simm5`	`mem(FPW + (int)simm5) := rX`
		`FPW := FPW - 4`
			FPZ-relative

16-bit	`stfpw_dec rX, simm5`	`mem(FPW + (int)simm5) := rX`
		`FPW := FPW - 2`
			FPZ-relative

Store (half-)word with frame pointer W and increment W (stfpw_inc)

OPC7							SIMM5					REG4			
0	1	0	1	0	0	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`stfpw_dec rX, simm5`	`mem(FPW + (int)simm5) := rX`
		`FPW := FPW + 4`
			FPW-relative

16-bit	`stfpw_dec rX, simm5`	`mem(FPW + (int)simm5) := rX`
		`FPW := FPW + 2`
			FPW-relative

Store (half-)word with frame pointer X (stfpx)

OPC6						SIMM6						REG4			
0	1	1	1	0	1	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode
16/32-bit	`stfpx rX, simm6`	`mem(FPX + (int)simm6) := rX`	register indirect

Store (half-)word with frame pointer X and decrement X (stfpx_dec)

OPC7							SIMM5					REG4			
0	1	0	1	0	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`stfpx_dec rX, simm5`	`mem(FPX + (int)simm5) := rX`
		`FPX := FPX - 4`
			FPX-relative

16-bit	`stfpx_dec rX, simm5`	`mem(FPX + (int)simm5) := rX`
		`FPX := FPX - 2`
			FPX-relative

Store (half-)word with frame pointer X and increment X (stfpx_inc)

OPC7							SIMM5					REG4			
0	1	0	1	0	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`stfpx_dec rX, simm5`	`mem(FPX + (int)simm5) := rX`
		`FPX := FPX + 4`
			FPX-relative

16-bit	`stfpx_dec rX, simm5`	`mem(FPX + (int)simm5) := rX`
		`FPX := FPX + 2`
			FPX-relative

Store (half-)word with frame pointer Y (stfpy)

OPC6						SIMM6						REG4			
0	1	1	1	1	0	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode
16/32-bit	`stfpy rX, simm6`	`mem(FPY + (int)simm6) := rX`	register indirect

Store (half-)word with frame pointer Y and decrement Y (stfpy_dec)

OPC7							SIMM5					REG4			
0	1	0	1	1	0	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`stfpy_dec rX, simm5`	`mem(FPY + (int)simm5) := rX`
		`FPY := FPY - 4`
			FPY-relative

16-bit	`stfpy_dec rX, simm5`	`mem(FPX + (int)simm5) := rX`
		`FPY := FPY - 2`
			FPY-relative

Store (half-)word with frame pointer Y and increment Y (stfpy_inc)

OPC7							SIMM5					REG4			
0	1	0	1	1	0	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`stfpy_dec rX, simm5`	`mem(FPY + (int)simm5) := rX`
		`FPY := FPY + 4`
			FPY-relative

16-bit	`stfpy_dec rX, simm5`	`mem(FPX + (int)simm5) := rX`
		`FPY := FPY + 2`
			FPY-relative

Store (half-)word with frame pointer Z (stfpz)

OPC6						SIMM6						REG4			
0	1	1	1	1	1	I_5	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode
16/32-bit	`stfpz rX, simm6`	`mem(FPZ + (int)simm6) := rX`	register indirect

Store (half-)word with frame pointer Z and decrement Z (stfpz_dec)

OPC7							SIMM5					REG4			
0	1	0	1	1	1	1	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`stfpz_dec rX, simm5`	`mem(FPZ + (int)simm5) := rX`
		`FPZ := FPZ - 4`
			FPZ-relative

16-bit	`stfpz_dec rX, simm5`	`mem(FPZ + (int)simm5) := rX`
		`FPZ := FPZ - 2`
			FPZ-relative

Store (half-)word with frame pointer Z and increment Z (stfpz_inc)

OPC7							SIMM5					REG4			
0	1	0	1	1	1	0	I_4	I_3	I_2	I_1	I_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode

32-bit	`stfpz_dec rX, simm5`	`mem(FPZ + (int)simm5) := rX`
		`FPZ := FPZ + 4`
			FPZ-relative

16-bit	`stfpz_dec rX, simm5`	`mem(FPZ + (int)simm5) := rX`
		`FPZ := FPZ + 2`
			FPZ-relative

Store half-word (sth)

OPC8								REG4				REG4			
1	1	1	1	0	1	1	0	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode
32-bit	`sth rX, rY`	`hword := rX(15:0)` $\texttt{mem}_{16}$`(rY) := hword`	register indirect
16-bit	`sth rX, rY`	`mem(rY) := rX`	register indirect

Store word (stw)

OPC8								REG4				REG4			
1	1	1	1	0	1	0	1	Y_3	Y_2	Y_1	Y_0	X_3	X_2	X_1	X_0
15															0

Architecture	Assembly Code	Description	Addressing Mode
32-bit	`stw rX, rY`	`mem(rY) := rX`	register indirect

Bibliography

[Ben08a] Jeremy Bennett. Howto: GDB Remote Serial Protocol - Writing a RSP Server. http://www.embecosm.com/download/ean4.html, 2008. Issue 2.

[Ben08b] Jeremy Bennett. Howto: Porting the GNU Debugger - Practical Experience with the OpenRISC 1000 Architecture. http://www.embecosm.com/download/ean3.html, 2008. Issue 2.

[Ben10] Jeremy Bennett. Howto: Porting Newlib - A Simple Guide. http://www.embecosm.com/download/ean9.html, 2010. Issue 1.

[Dan05] Sivarama P. Dandamudi. *Introduction to Assembly Language Programming - For Pentium and RISC Processors.* Springer, second edition, 2005.

[DAP06] John L. Hennessy David A. Patterson. *Computer Architecture. A Quantitative Approach.* Morgan Kaufmann Publishers, fourth edition, 2006.

[DAP07] John L. Hennessy David A. Patterson. *Computer Organization and Design. The Hardware/Software Interface.* Morgan Kaufmann Publishers, third edition, 2007.

[Del07] Martin Delvai. Spear2: Erweiterungskonzept. Technical report, Department of Computer Engineering, Embedded Computing Systems Group, Vienna University of Technology, April 2007.

[DH07] Sarah Harris David Harris. *Digital Design and Computer Architecture.* Morgan Kaufmann Publishers, 2007.

[Eag07] Michael J. Eager. Introduction to the DWARF Debugging Format. http://www.dwarfstd.org/doc/DebuggingusingDWARF.pdf, 2007.

[Fle08] Martin Fletzer. Spear2 - an improved version of spear, February 2008.

[Gat] Bill Gatliff. Porting and Using Newlib in Embedded Systems. http://www.billgatliff.com/newlib.html.

[Lea] Doug Lea. A Memory Allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[Lev99] John R. Levine. *Linkers and Loaders.* Morgan Kaufmann Publishers, 1999.

[Mau08] Wolfgang Mauerer. *Professional Linux Kernel Architecture.* John Wiley & Sons, 2008.

[Mei08] Ludwig Meier. Portierung des gdb für die spear2 architektur. Technical report, Department of Computer Engineering, Embedded Computing Systems Group, Vienna University of Technology, November 2008.

[Mos08] Josef Mosser. Amba4spear2: An amba extension module for the spear2 processor core, March 2008.

[Noe05] Tammy Noergaard. *Embedded Systems Architecture - A Comprehensive Guide for Engineers and Programmers.* Newnes, 2005.

[Nur07] Jari Nurmi. *Processor Design - System-on-Chip Computing for ASICs and FPGAs.* Springer, 2007.

[Puf04] Wolfgang Puffitsch. cas - a c-style macro assembler. Technical report, Department of Computer Engineering, Embedded Computing Systems Group, Vienna University of Technology, 2004.

[Puf07] Wolfgang Puffitsch. Softwaretools für den spear. Technical report, Department of Computer Engineering, Embedded Computing Systems Group, Vienna University of Technology, Mai 2007.

[Red09] Swami Reddy. Binutils Porting Guide To A New Target Architecture. http://sourceware.org/binutils/binutils-porting-guide.txt, 2009.

[SD07] Uday P. Khedker Sameera Deshpande. Incremental Machine Descriptions for GCC. http://www.cse.iitb.ac.in/~uday/soft-copies/incrementalMD.pdf, 2007.

[Sei07] Roman Seiger. miniuart dokumentation. Technical report, Department of Computer Engineering, Embedded Computing Systems Group, Vienna University of Technology, July 2007.

[Shi07] Sajjan G. Shiva. *Computer Organization, Design and Architecture.* Auerbach Publications, fourth edition, 2007.

[SPHI02] Guy L. Steele Jr. Samuel P. Harbison III. *C: A Reference Manual.* Prentice Hall, fifth edition, 2002.

[Swe06] Dominic Sweetman. *See MIPS Run.* Morgan Kaufmann Publishers, second edition, 2006.

[Und] Understanding Memory. http://www.ualberta.ca/CNS/RESEARCH/LinuxClusters/mem.html.

[wik] wiki.netbsd.org. ELF Executables for PowerPC. http://wiki.netbsd.org/ELF_Executables_for_PowerPC.

[WRS03] M. Andrew Rudoff W. Richard Stevens, Bill Fenner. *Unix Network Programming: The Sockets Networking API, Volume 1.* Prentice Hall, third edition, 2003.

[Yiu07] Joseph Yiu. *The Definitive Guide to the Arm Cortex-M3.* Newnes Publishers, 2007.