Leros: the Return of the Accumulator Machine

Martin Schoeberl[1] and Morten Borup Petersen[1]

Department of Applied Mathematics and Computer Science
Technical University of Denmark, Kgs. Lyngby, Denmark
masca@dtu.dk, s152999@student.dtu.dk

Abstract. An accumulator instruction set architecture is simpler than an instruction set of a (reduced instruction set computer) RISC architecture. Therefore, an accumulator instruction set that does within one instruction less than a typical RISC instruction is probably more "reduced" than a standard load/store register based RISC architecture.
This paper presents Leros, an accumulator machine and its supporting C compiler. The hypothesis of the Leros instruction set architecture is that it can deliver the same performance as a RISC pipeline, but consumes less hardware and therefore also less power.

Keywords: Embedded Systems · Minimal Processor.

1 Introduction

The invention of the reduced instruction set computer (RISC) [9,12,13] in the early 80's was a sort of a revolution. Since then most embedded processors have been designed as RISC processors, and from the Pentium Pro, the x86, a typical complex instruction set computer (CISC), uses RISC style instructions internally. Recently the free RISC-V instruction set [19], also developed at the University of California, Berkeley is gaining momentum. First silicon implementations are available. Even a many-core architecture with more than 4096 RISC-V processors on a single die is under development by Esperanto [7] and expected to ship mid-2019.

The RISC architecture promises to provide a simpler instruction set that is cheaper to implement and more natural to pipeline to achieve high performance by a higher clock frequency and fewer clocks per instructions. A typical RISC architecture has: 32-bit instructions, 32 registers, operation with two source and one destination register, and load and store instructions with displacement addressing.

This paper takes the RISC approach one step further and provides an even more *RISCy* instruction set: Leros, an accumulator machine. An accumulator instruction set is even simpler than a RISC instruction set. This processor is named Leros, after the Greek island Leros.[1] Leros is an accumulator machine with direct addressing of 256 memory words. Those 256 words are considered a large register file. Leros implements basic logic and arithmetic operations with an accumulator and one of the registers or

[1] The initial version of the processor has been designed on the island Leros: https://www.leros.gr/en/.

a constant. Memory is accessed indirectly via an address register. All instructions are 16-bit wide. Leros can be configured to be a 16, 32, or 64-bit architecture. We optimize Leros for an FPGA, by using an on-chip memory for the large registers file.

The Leros architecture hypothesizes that it will deliver the same performance as a RISC pipeline, but consumes fewer hardware resources and therefore also less power. The Leros accumulator architecture will execute more instructions than a RISC architecture. However, the accumulator architecture compensates this by two facts: (1) The simple architecture shall allow clocking the pipeline with a higher clock frequency. (2) The shorter instructions (16 instead of 32 bits) need less instruction memory and instruction cache.

A further goal of Leros is to be a good target for a C compiler. Therefore, the data width shall be 32 bits. We present a port of the LLVM [10] C compiler for Leros.

The contributions of this paper are: (1) a definition of a minimal accumulator based instruction set architecture (ISA), (2) an implementation of that ISA in two simulators and in an FPGA, and (3) a C compiler ported to target the Leros ISA.

This paper is organized in 7 sections: The following section presents related work. Section 3 describes the Leros instruction set architecture. Section 4 describes one possible implementation of the Leros processor. Section 5 introduces the C compiler for Leros. Section 6 evaluates the architecture, the compiler, and an FPGA implementation of Leros. Section 7 concludes.

2 Related Work

Many small processor cores for FPGAs have been developed or are developed as assignments for courses in computer architecture. With Leros, we also aim to be an instruction set definition that can be used in teaching. In this section, we restrict the discussion to a few successful cores and point out the differences from our Leros design.

PicoBlaze is an 8-bit processor for Xilinx FPGAs [21]. PicoBlaze is optimized for resource usage and therefore restricts the maximum program size to 1024 instructions and data to 64 bytes. PicoBlaze can be implemented with one on-chip memory and 96 logic slices in a Spartan-3 FPGA. PicoBlaze provides 16 8-bit registers and executes one instruction in two clock cycles.

The central theme behind Leros is similar to PicoBlaze. However, we target a processor that is useful with a C compiler. Thus, the resource consumption of Leros is slightly higher as PicoBlaze. The PicoBlaze code is at a low level of abstraction composed out of Xilinx primitive components. Therefore, the design is optimized for Xilinx FPGAs and practically not portable. Leros is written in vendor agnostic Chisel [2] and compiles unmodified for Altera and Xilinx devices.

The SpartanMC is a small microcontroller with an instruction and data width of 18 bits [8]. The authors optimized this width for FPGAs that contain on-chip memories that can be 18-bit wide (the additional bits are initially for parity protection). The processor has two operand instructions with 16 registers and is implemented in a three-stage pipeline. The register file uses on-chip memory and a sliding register window is used to speed up function calls (similar to the SPARC architecture). SpartanMC per-

forms comparably to the 32-bit RISC processors LEON-II [6] and MicroBlaze [22] on the Dhrystone benchmark.

Compared to the SpartanMC, we further optimized Leros for FPGAs using fewer resources. Leros simplifies the access to registers in on-chip memory by implementing an accumulator architecture instead of a register architecture. Although an accumulator architecture is, in theory, less efficient, the resulting maximum achievable clock frequency may offset the higher instruction count.

Intel (former Altera) provides the Nios II [1] processor, which is optimized for Intel FPGAs. Nios is a 32-bit RISC architecture with an instruction set similar to MIPS [9] with three register operations. The sizes of its instruction and data caches are configurable.

Although Nios II represents a different design from Leros, it is a clear competitor, as one can configure Nios for different resource consumption and performance targets. Three different models are available: the *Fast* core is optimized for high performance, the *Standard* core is intended to balance performance and size, and the *Economy* core is optimized for smallest size. The smallest core needs less than 700 logic elements (LEs). It is a sequential implementation, and each instruction takes at least six clock cycles. Leros is a smaller, accumulator-based architecture, and with a pipelined implementation of Leros, most instructions can execute in a single clock cycle.

The *Supersmall* processor [15] is optimized for low resource consumption (half of the Nios economy version). This is achieved by serializing ALU operations to single bit operations. The LE consumption is comparable to Leros.

The Ultrasmall MIPS project [11] extends the Supersmall architecture. The main difference is the change of the ALU serialization to perform two-bit operations each cycle instead of single bits. Therefore, a 32-bit operation needs 16 clock cycles to complete. The Ultrasmall processor consumes 137 slices in a Xilinx Spartan-3E, which is 84 % of the resource consumption of Supersmall. The serialization of the ALU operations results in an average of 22 clock cycles per instructions. According to the authors, "Ultrasmall is the smallest 32-bit ISA soft processor in the world". We appreciate this effort of building the smallest 32-bit processor. With Leros, we aim similar for a small 32-bit processor.

Wolfgang Puffitsch developed the Ø processor.[2] It is an accumulator machine aiming at low resource usage. The bit width of the accumulator (and register width) is configurable. An instance of an 8-bit Ø processor executing a blinking function consumes 176 LEs and 32 memory bits.

An early processor targeting FPGAs is the DOP processor [3]. DOP is a 16-bit stack oriented processor with additional registers, such as address registers and a work register. As the work register is directly connected to the ALU, DOP is similar to Leros an accumulator oriented architecture. The authors do not provide resource consumptions for the DOP design.

Lipsi is a processor that aims to be one of the smallest processors available for an FPGA [18]. Lipsi is small as it can use just a single block RAM for instructions and data. Therefore, each instruction executes in at least two clock cycles. The datapath of Lipsi is 8-bit. The aims of Lipsi and Leros are similar to build small embedded processors.

[2] `https://github.com/jeuneS2/oe`

However, with Leros, we target a processor that is well suited for a modern C compiler. Therefore, the default datapath width is 32-bit but is configurable to be 16, 32, or 64 bits.

The first version of Leros [16] was a hardcoded 16-bit accumulator machine. It consisted of a two-stage pipeline, where the pipeline delays are visible in the instruction definition. Compared to this initial version of Leros, we make a clear definition of the instruction set architecture, independent from any implementation in this paper. Furthermore, we allow that the bit width is configurable. And we provide a port of the LLVM C compiler for Leros. The porting of the C compiler also provided feedback on the instruction set that we changed accordingly. Therefore, the presented version of Leros is not binary compatible with the early version of Leros.

3 The Leros Instruction Set Architecture

The instruction set architecture, or short ISA, is the most important interface of a processor. It defines the language that the processor understands. It is the interface between the hardware and the compiler. IBM first introduced an ISA with the 360 series of computers. IBM introduced several implementations of the 360 series, with different price tags, that all implemented the same ISA. Therefore, it was possible to reuse software and compilers on different computers.

The ISA defines the programmer visible state, e.g., registers and memory, and instructions that operate on this state. The processor state that is not visible to the programmer, e.g., caches, are not part of the ISA. Some parts of a processor, e.g., address translation and memory protection, are not visible in the user ISA, but only available in a supervisor mode (usually used by an operating system kernel).

Leros is an accumulator machine. Therefore, the dominant register is the accumulator A. Furthermore, Leros defines a small memory area that can be directly addressed. We call those 256 memory words registers. Leros performs operations with the accumulator and those registers. E.g., Adding a register to the accumulator, storing the accumulator into a register. Basic operations are also available with immediate values, e.g., adding a constant to A.

Memory operations use an address register, called AR, plus an 8-bit displacement. All memory accesses use this address register. The load destination is the accumulator, and the store source is also the accumulator.

All instructions are 16-bit. The data width of Leros is configurable to be: 16, 32, or 64 bits. The default implementation of Leros is 32-bit.

A set of branch instructions perform unconditional and conditional branches depending on A (zero, non-zero, positive, or negative). For larger branch targets, indirect jumps, and calls, Leros has a jump and link instruction that jumps to the address in A and stores the address of the next instruction in a register. Furthermore, we define a system call instruction for operating system calls.

Leros is designed to be simple, but still a good target for a C compiler. The description of the instruction set fits in less than one page, see Table 1. In that table A represents the accumulator, PC is the program counter, i is an immediate value (0 to 255), Rn a

Opcode	Function	Description
add	A = A + Rn	Add register Rn to A
addi	A = A + i	Add immediate value i to A (sign extend i)
sub	A = A - Rn	Subtract register Rn from A
subi	A = A - i	Subtract immediate value i from A (sign extend i)
shr	A = A >>> 1	Shift A logically right
and	A = A and Rn	And register Rn with A
andi	A = A and i	And immediate value i with A
or	A = A or Rn	Or register Rn with A
ori	A = A or i	Or immediate value i with A
xor	A = A xor Rn	Xor register Rn with A
xori	A = A xor i	Xor immediate value i with A
load	A = Rn	Load register Rn into A
loadi	A = i	Load immediate value i into A (sign extend i)
loadhi	A_{31-8} = i	Load immediate into second byte (sign extend i)
loadh2i	A_{31-16} = i	Load immediate into third byte (sign extend i)
loadh3i	A_{31-24} = i	Load immediate into fourth byte (sign extend i)
store	Rn = A	Store A into register Rn
jal	PC = A, Rn = PC + 2	Jump to A and store return address in Rn
ldaddr	AR = A	Load address register AR with A
loadind	A = mem[AR+(i << 2)]	Load a word from memory into A
loadindbu	A = mem[AR+i]_{7-0}	Load a byte unsigned from memory into A
storeind	mem[AR+(i << 2)] = A	Store A into memory
storeindb	mem[AR+i]_{7-0} = A	Store a byte into memory
br	PC = PC + o	Branch
brz	if A == 0 PC = PC + o	Branch if A is zero
brnz	if A != 0 PC = PC + o	Branch if A is not zero
brp	if A >= 0 PC = PC + o	Branch if A is positive
brn	if A < 0 PC = PC + o	Branch if A is negative
scall	scall A	System call (simulation hook)

Table 1: The Leros instruction set.

register n (0 to 255), o a branch offset relative to PC, and AR an address register for memory access.

4 A Leros Implementation

With the Leros ISA, we do not define any specific implementation. Sequential, single cycle, or pipelined implementations are all proper implementations of the Leros ISA. The initial Leros 16-bit processor [16] used the pipeline implementation as part of the ISA definition, which limits the usefulness of an ISA definition. Therefore, we remove this restriction with the current definition. Instruction dependencies within a pipeline need to be resolved in hardware (by forwarding or stalling). No pipeline effects shall be visible in the ISA (except in the execution time of an instruction).

As a golden reference, we have implemented a Leros ISA simulator in Scala. The simulator is a large `match/case` statement and is implemented in around 100 lines of code. The simulator also reflects the simplicity of the Leros ISA.

Writing an assembler with an expressive language like Scala is not a big project. Therefore, we wrote a simple assembler for Leros, which is possible within about 100 lines of code. We define a function `getProgram` that calls the assembler. For branch destinations, we need a symbol table, which we collect in a `Map`. A classic assembler runs in two passes: (1) collect the values for the symbol table and (2) assemble the program with the symbols obtained in the first pass. Therefore, we call the assembler twice with a parameter to indicate which pass it is.

The ISA simulator and the hardware implementation of Leros call the function `getProgram` to assemble a program at simulation or hardware generation time.

We have chosen Scala for the simulator and the assembler as we use Chisel, which is a Scala library, to describe the hardware implementation. We can share constants that define the instruction encoding between the simulator, the assembler, and the hardware implementation.

The 256 registers of Leros are similar to the work registers of the TMS9900 CPU, the processor that was used in the first 16-bit personal computer TI-99/4A.[3] The TMS9900 had 16 registers, which are kept in RAM. An implementation of Leros may map those registers into the main memory and cache it in a data cache. Or it can implement the registers in on-chip memory, also called scratchpad memory. The Leros ISA does not define this implementation details. The ISA specification does not assume that the registers can be read or written with memory load and store instructions.

For testing, we wrote a few test programs in assembler with the convention that at the end of the test the accumulator shall be 0. Those tests are executed in the software simulator of Leros and in the hardware simulation in Chisel.

Furthermore, as we implemented the hardware description and the software simulator in the same language, we can do co-simulation. With co-simulation, we compare after each instruction the content of A between the software simulation and the hardware. Any (relevant) difference/error will eventually show up in A as all data flows through A.

5 The Leros C Compiler

We implemented a C compiler and accompanying toolchain for the Leros instruction set with the LLVM compiler infrastructure. A detailed description of the compiler and tools for Leros can be found in [14].

The LLVM compiler infrastructure is a collection of toolchain applications built around the LLVM core libraries. The LLVM core is a modular compiler infrastructure, allowing for separate implementation of front-, optimizer, and backends. We implemented an LLVM backend that targets the Leros instruction set.

[3] `https://en.wikipedia.org/wiki/Texas_Instruments_TI-99/4A`

5.1 Using LLVM For Accumulator Machines

A difficulty in using LLVM for Leros arises when we directly use the intermediate representation (IR) of LLVM. LLVM follows the common notion of compiler IR wherein the IR should resemble the target instruction set format, to facilitate various steps such as optimizations and instruction selection. An example LLVM IR sequence may be the addition of two variables:

```
%c = add i32 %a, %b
```

The format of the LLVM IR resembles 3-operand RISC instruction sets, which facilitates instruction selection and emission for instruction sets such as ARM and RISC-V. For Leros, virtually no LLVM IR instructions can be directly matched to Leros instructions.

The method for matching the LLVM IR during instruction selection has been to implement a 3-operand version of the Leros instruction set, denoted as the Leros pseudo instruction set. An example expansion of a Leros pseudo instruction is as follows:

```
%c = add %a %b      load    %a
                    add     %b
                    store   %c
```

Having mappings such as shown above allows for instruction selection with the ease enjoyed by the 3-operand upstream backends of LLVM. After scheduling, SSA optimizations and register allocation the Leros pseudo instructions are expanded to their corresponding sequence of Leros machine instructions. Whilst incurring a code-size overhead, the method does not require any modifications to the DAG which is provided as the input for the backend as well as the built-in scheduler, SSA optimization- and register allocation passes, which are desired to be left as default to minimize implementation time as well as the possibility for compiler issues.

5.2 Accumulator Optimizations

A consequence of the pseudo instruction set is an overhead in the size of the compiled programs, mainly due to redundant instructions which are a side-effect of the pseudo instruction set. Therefore, we implemented various optimizations to detect and modify code sequences where a program may reuse the accumulator content. An example is the removal of redundant load and stores. Figure 1 shows an example of Leros machine code after pseudo instruction expansion. We can see that the intermittent load- and store to `%tmp` is redundant, and the compiler may remove it if the register `%tmp` is dead after the `load %tmp` instruction.

As of this writing, we have implemented three optimization passes in the backend:

Redundant loads: Identifies code sequences as shown in figure 1 where a register is loaded wherein the value of the register is already present in the accumulator.

Redundant stores: Identifies code sequences as shown in figure 1 where a register is used to store an intermediate result. Redundant store instructions are identified and removed by reverse traversal of a basic-block, checking register liveness and usage.

```
                          load   %a
                          add    %b            load   %a
%tmp = add i32 %a %b      store  %tmp          add    %b
%d = add i32 %a %tmp      load   %tmp          add    %a
                          add    %a            store  %d
                          store  %d
```

Fig. 1: Left: the LLVM IR sequence, center: expanded pseudo instructions, and right: an optimal sequence.

Redundant `ldaddr`: All `ldind` and `stind` instructions emit a `ldaddr` instruction, resulting in code sequences where multiple `ldaddr` instructions will load an unmodified value into the address register. This pass mimics the redundant store pass, tracking the usage of the register which is currently loaded into the address register and removes `ldaddr` instructions if deemed redundant.

5.3 Further Optimizations

Some pseudo instruction expansions require the use of bit masks. An example is the expansion of arithmetic right shift instructions. In this, the bitmask `0x80000000` is required for sign-extending a (logically right shifted) value. The compiler generates immediate values through subsequent `loadi#` instructions.

Given that the compiler knows the required set of constants for instruction expansion at compile time, these constant can be stored in registers. The abundance of registers in Leros allows for using some of the registers for constants. With this, we define some constant registers for Leros, which the `_start` function initializes. These constant registers are furthermore able to be referenced during instruction selection.

For custom inserters in which more instructions are required to express the action than what the compiler emits as function call overhead, we should move these to a runtime library. Using library functions addresses the current issue of identical pseudo instruction expansions being repeated multiple times throughout code. Furthermore, given that these runtime functions will be often called the addresses of these functions may be kept in registers. The expected effect of this is a slight performance decrease given the call overhead but a significant reduction in code size.

5.4 Toolchain

By leveraging the LLVM compiler infrastructure, a number of different tools have been integrated with support for the Leros instruction set. Clang is used as the C frontend of choice, as well as being a compiler driver for the remainder of the toolchain. LLVMs `lld` linker has been modified with support for the Leros relocation symbols, and shall be used in place of system linkers like `gold`. Furthermore, LLVM provides a slew of binary utilities akin to the GNU Binutils collection of applications such as `llvm-dis`, the LLVM disassembler, `llvm-readelf`, the LLVM ELF reader with support for Leros relocation flags, `llvm-objcopy, llvm-objdump` and others.

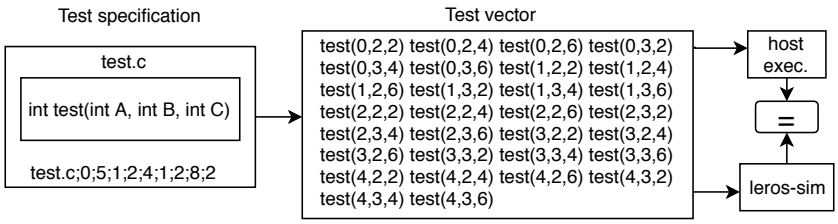

Fig. 2: The test suit compiles and executes the test specification (source file and input ranges) for all combinations of the input ranges on the host- and target systems, with host results serving as golden references.

For simpler simulators as well as executing Leros code on hardware the `llvm-objdump` tool may be used to extract the `.text` and `.data` segment of the compiled program, yielding a flat binary which may be executed from address `0x0`, removing the need for a simulater reading ELF files or some hardware to interpret ELF files.

6 Evaluation

6.1 Automated Test Suite

While LLVM contains many fuzzing tools used for verifying that a backend can select all instructions of the LLVM IR, it cannot check the semantics of the produced code. We developed an automated test suite to check the semantics of the generated code. The test suite compiles the programs with a host compiler and with our compiler for Leros and executes them on the host and in the Leros simulator. The test compares then the outputs of the two runs.

The test suite is a Python script which given a test specification file may control the host and Leros compilers as seen in figure 2. Each line in the test specification file contains a reference to a test file as well as a range and step for all input arguments. The test source file is compiled for the host system as well as using the Leros compiler, whereafter the program is executed using the set of all combinations of arguments. All test programs return a value. The test suit compares the test return value of the host and simulator execution. The test suite has proved a valuable asset in identifying issues and verifying the correctness of instruction expansion and optimization passes. Furthermore, it functions as a regression test suite allowing for fewer errors to propagate to the source repositories.

6.2 Leros ISA Performance

To validate the compiler as well as generate indicators of the efficacy of the ISA, we use the CoreMark benchmark [4]. CoreMark is a synthetic benchmark designed for embedded systems which aims to be an industry standard benchmark for embedded systems, replacing the older DhryStone benchmark [20].

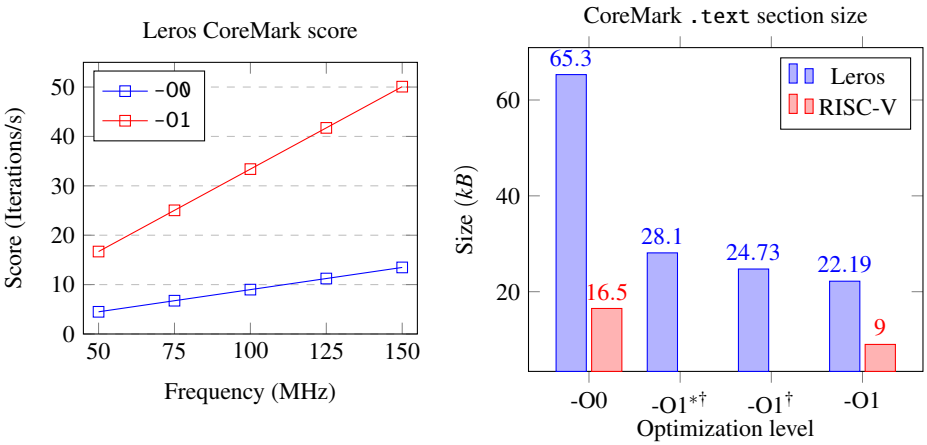

Fig. 3: Leros CoreMark results

Figure 3 shows the Leros CoreMark score and ELF `.text` section size for various optimization levels.
The CoreMark scores generated from the Leros simulator assumes a memory access time of 1 cycle and an IPC of 1. The Leros CoreMark score is comparable to other low-end embedded devices, such as the STMicroelectronics STM32L053 [5]. This device is based on the Arm Cortex-M0+ architecture and manages a score of 39.91 at a clock frequency of 16 MHz.

In Figure 3 we can see a significant code size difference between Leros and the RISC-V compilation. We can find several factors for this overhead. An accumulator-based instruction set as Leros will usually require more instructions to execute an action than a 3-operand instruction set (such as RISC-V). A single RISC instruction may need up to three instructions (`load`, `op`, `store`) in an accumulator machine.

The custom inserters used by Leros incurs an overhead through the requirement to emit many instructions, in place of what a single instruction in RISC-V can express, e.g., arbitrary shifts and sign-extended loads.

In general, code size will correlate to the efficacy of the instruction set. For CISC instruction sets code size will be smaller compared to the semantically equivalent code produced for a RISC instruction set. The same pattern shows for Leros in comparison to RISC-V, wherein Leros is arguably more RISC than RISC-V.

Comparing $-\text{O1}^{*\dagger}$ to $-\text{O1}^{\dagger}$, the accumulator optimization passes manage a code size reduction of 12.75%. Comparing $-\text{O1}^{\dagger}$ to $-\text{O1}$, the introduction of constant registers shows a further decrease of 10.82%. These are significant reductions in code size. We expect to decrease further when we implement more accumulator optimizations and build a runtime library for the custom inserters.

*No accumulator optimizations
†No constant registers

The successful compilation and execution of the CoreMark benchmark show that the Leros ISA is a valid C target.

6.3 Leros in Teaching

The simplicity of Leros makes it a good candidate for teaching an introductory class in computer architecture. The description of the Leros ISA fits in less than one page in this paper format, see Table 1. Therefore, one can quickly memorize the ISA. A simple exercise for a lab would be the implementation of a Leros software simulator and then explore the usage of the instructions from compiled C programs. In a larger project, for students with hardware design knowledge, implementing Leros in an FPGA would be a good project, as the infrastructure (C compiler, assembler, and simulator) are available.

Leros is used in a Chisel textbook [17] as a medium sized project in one of the last chapters. That chapter contains a detailed description of the hardware designed in Chisel and simulator and assembler in Scala. Leros serves as an example of the powerful combination of Chisel and the general purpose language Scala. E.g., an assembler, written in Scala, is executed as part of the hardware generation process.

6.4 Source Access

The Leros processor, compiler, and other related repositories are available in open source at `https://github.com/leros-dev`.

7 Conclusion

In this paper, we present a minimal instruction set architecture (ISA): the Leros accumulator machine. The idea behind this ISA is the same as the one for a RISC instruction set: provide just basic instructions and let the more complex functions be done by the compiler. Leros takes that step further and defines an even simpler ISA than a RISC processor, which shall still be a useful target for C.

That simple ISA leads to the simple implementation of simulators and hardware in an FPGA. We have ported the LLVM compiler to support Leros. Besides serving as a small embedded processor, the simplicity of Leros makes it also a good example for an introductory course in computer architecture. Leros also serves as a running example in a final chapter of a digital design textbook in Chisel.

References

1. Altera Corporation: Nios II processor reference handbook. Available from `http://www.altera.com/literature/lit-nio2.jsp` (May 2011), version NII5V1-11.0
2. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avizienis, R., Wawrzynek, J., Asanovic, K.: Chisel: constructing hardware in a scala embedded language. In: The 49th Annual Design Automation Conference (DAC 2012). pp. 1216–1225. ACM, San Francisco, CA, USA (June 2012)

3. Danecek, J., Drapal, F., Pluhacek, A., Salcic, Z., Servit, M.: Dop—a simple processor for custom computing machines. Journal of Microcomputer Applications **17**(3), 239 – 253 (1994). https://doi.org/10.1006/jmca.1994.1015
4. EEMBC: Coremark - an eembc benchmark (2018 (accessed December 12-12-2018)), https://www.eembc.org/coremark/
5. EEMBC: Coremark benchmark score - stmicroelectronics stm32l053 (2018 (accessed November 12-12-2018)), https://www.eembc.org/benchmark/reports/benchreport.php?suite=CORE&bench_scores=1689
6. Gaisler, J.: A portable and fault-tolerant microprocessor based on the SPARC v8 architecture. In: DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks. p. 409. IEEE Computer Society, Washington, DC, USA (2002). http://doi.ieeecomputersociety.org/10.1109/DSN.2002.1028926
7. Gwennap, L.: Esperanto makes out RISC-V. Tech. rep., The Linley Group, Microprocessor Report (December 2018)
8. Hempel, G., Hochberger, C.: A resource optimized processor core for FPGA based SoCs. In: Kubatova, H. (ed.) Proceedings of the 10th Euromicro Conference on Digital System Design (DSD 2007). pp. 51–58. IEEE (2007)
9. Hennessy, J.L.: VLSI processor architecture. Computers, IEEE Transactions on **C-33**(12), 1221–1246 (Dec 1984). 10.1109/TC.1984.1676395
10. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization (CGO'04). pp. 75–88. IEEE Computer Society (2004)
11. Nakatsuka, H., Tanaka, Y., Chu, T.V., Takamaeda-Yamazaki, S., Kise, K.: Ultrasmall: The smallest mips soft processor. In: 2014 24th International Conference on Field Programmable Logic and Applications (FPL). pp. 1–4 (Sept 2014). 10.1109/FPL.2014.6927387
12. Patterson, D.A.: Reduced instruction set computers. Commun. ACM **28**(1), 8–21 (1985). http://doi.acm.org/10.1145/2465.214917
13. Patterson, D.A., Sequin, C.H.: RISC I: A reduced instruction set VLSI computer. In: Proceedings of the 8th annual symposium on Computer Architecture. pp. 443–457. ISCA '81, IEEE Computer Society Press, Los Alamitos, CA, USA (1981)
14. Petersen, M.B.: A compiler backend and toolchain for the leros architecture. B.sc.eng. thesis, Technical University of Denmark (2019)
15. Robinson, J., Vafaee, S., Scobbie, J., Ritche, M., Rose, J.: The supersmall soft processor. In: Programmable Logic Conference (SPL), 2010 VI Southern. pp. 3 –8 (march 2010). 10.1109/SPL.2010.5483016
16. Schoeberl, M.: Leros: A tiny microcontroller for FPGAs. In: Proceedings of the 21st International Conference on Field Programmable Logic and Applications (FPL 2011). pp. 10–14. IEEE Computer Society, Chania, Crete, Greece (September 2011)
17. Schoeberl, M.: Digital Design with Chisel. TBD V 0.1 (2018), available at https://github.com/schoeberl/chisel-book
18. Schoeberl, M.: Lipsi: Probably the smallest processor in the world. In: Architecture of Computing Systems – ARCS 2018. pp. 18–30. Springer International Publishing (2018). 10.1007/978-3-319-77610-1_2
19. Waterman, A.: Design of the RISC-V Instruction Set Architecture. Ph.D. thesis, EECS Department, University of California, Berkeley (Jan 2016)
20. Weicker, R.P.: Dhrystone: a synthetic systems programming benchmark. Communications of the ACM (1984). 10.1145/358274.358283
21. Xilinx: PicoBlaze 8-bit embedded microcontroller user guide (2010)
22. Xilinx Inc.: MicroBlaze processor reference guide (2008), version 9.0