Lipsi: Probably the Smallest Processor in the World

Schoeberl, Martin

Published in:
Architecture of Computing Systems

Link to article, DOI:
10.1007/978-3-319-77610-1_2

Publication date:
2018

Document Version
Peer reviewed version

Citation (APA):
Schoeberl, M. (2018). Lipsi: Probably the Smallest Processor in the World. In *Architecture of Computing Systems* (Vol. 10793, pp. 18-30). Springer. Lecture Notes in Computer Science Vol. 10793 https://doi.org/10.1007/978-3-319-77610-1_2

Lipsi: Probably the Smallest Processor in the World

Martin Schoeberl

Department of Applied Mathematics and Computer Science
Technical University of Denmark
masca@dtu.dk

Abstract. While research on high-performance processors is important, it is also interesting to explore processor architectures at the other end of the spectrum: tiny processor cores for auxiliary functions. While it is common to implement small circuits for such functions, such as a serial port, in dedicated hardware, usually as a state machine or a combination of communicating state machines, these functionalities may also be implemented by a small processor. In this paper, we present Lipsi, a very tiny processor to make it possible to implement classic finite state machine logic in software at a minimal cost.

1 Introduction

This paper presents Lipsi, a tiny microcontroller optimized for utility functions in an FPGA. Lipsi can be used to implement a peripheral device or a state machine as part of a larger system-on-chip. The design goal of Lipsi is a very small hardware design built around a single block RAM for instructions and data.

Using a single block RAM for instructions and data means that this memory is time shared between instruction fetch and data read. Therefore, Lipsi is a sequential and not a pipelined architecture. Most instructions execute in two clock cycles.

Lipsi is such a simple processor that it is possible to completely describe its datapath, instruction set and instruction encoding in a paper. Besides being a useful processor for auxiliary functions, we also envision Lipsi being used in teaching basic computer architecture. For example, it can be used to learn programming at the machine level. Or it should be possible for students to develop a simulator for Lipsi in a single lab session.

Lipsi is part of a family of processors, which all have been designed during an inspiring vacation on Greek islands, which gave the processors their names. The name for each processor was chosen from that island where the first sketches were drawn. The three sisters are: Patmos, Leros, and Lipsi. Patmos is a dual issue, 32-bit RISC pipeline optimized for real-time systems [1] and used in the multicore T-CREST platform [2]. Leros is a 16-bit processor for small embedded systems [3] and can execute a small Java virtual machine [4]. Lipsi is the smallest sister and an 8-bit accumulator architecture using a single on-chip block RAM, which is the topic of this paper.

This paper is organized in 5 sections: The following section presents related work. Section 3 describes the design of Lipsi. Section 4 evaluates and discusses the design. Section 5 concludes.

2 Related Work

Altera provides a softcore, the Nios II [5], for Altera FPGAs. The Nios RISC architecture implements a 32-bit instruction set like the MIPS instruction set architecture. Although Nios II represents a different design point from Lipsi, it is interesting to note that Nios II can be customized to meet the application requirements. Three different models are available [5]: 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 can be implemented in less than 700 logic elements (LEs). It is a sequential implementation and each instruction takes at least 6 clock cycles. Lipsi is a smaller (8-bit), accumulator-based architecture, and most instructions execute in two clock cycles.

PicoBlaze is an 8-bit microcontroller for Xilinx FPGAs [6]. The processor is highly optimized for low resource usage. This optimization results in restrictions such as a maximum program size of 1024 instructions and 64 bytes data memory. The benefit of this puristic design is a processor that 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 interface to I/O devices is minimalistic in the positive sense: it is simple and very efficient to connect simple I/O devices to the processor.

The Lipsi approach is, like the concept of PicoBlaze, to provide a small processor for utility functions. Lipsi is optimized to balance the resource usage between on-chip memory and logic cells. Therefore, the LE count of Lipsi is slightly lower than the one of PicoBlaze. PicoBlaze is coded at a very low level of abstraction by using Xilinx primitive components such as LUT4 or MUXCY. Therefore, the design is optimized for Xilinx FPGAs and practically not portable. Lipsi is written in vendor agnostic Chisel and compiles unmodified for Altera and Xilinx devices.

The SpartanMC is a small microcontroller optimized for FPGA technology [7]. One interesting feature is that the instruction width *and* the data width are 18 bits. The argument is that current FPGAs contain on-chip memory blocks that are 18-bit wide (originally intended to contain parity protection). The processor is a 16 register RISC architecture with two operand instructions and is implemented in a three-stage pipeline. To avoid data forwarding within the register file, the instruction fetch and the write-back stage are split into two phases, like the original MIPS pipeline [8]. This decision slightly complicates the design as two phase-shifted clocks are needed. We assume that this phase splitting also limits the maximum clock frequency. As on-chip memories for register files are large, this resource is utilized by a sliding register window to speedup function calls. SpartanMC performs comparable to the 32-bit RISC processors LEON-II [9] and MicroBlaze [10] on the Dhrystone benchmark.

Compared to the SpartanMC, Lipsi is further optimized for FPGAs using fewer resources and avoiding unusual clocking of pipeline stages. Lipsi 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 offsets the higher instruction count.

The *Supersmall* processor [11] is optimized for low resource consumption (half of the NIOS economy version). Resources are reduced by serializing ALU operations to single bit operations. The LE consumption is comparable to Lipsi, but the on-chip memory consumption is not reported.

The Ultrasmall MIPS project [12] is based on 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. It is reported that Ultrasmall consumes 137 slices in a Xilinx Spartan-3E, which is 84 % of the resource consumption of Supersmall. Due to the serialization of the ALU operations, the average clocks per instructions is in the range of 22 for Ultrasmall. 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 and are in line with that argument to build the smallest (8-bit) processor of the world.

The Ø processor by Wolfgang Puffitsch[1] is an accumulator machine aiming at low resource usage. The bit width of the accumulator (and register width) is freely configurable. Furthermore, hardware is only generated for instructions that are used in the program. An instance of an 8-bit Ø processor executing a blinking function consumes 176 LEs and 32 memory bits. The Ø processor is designed with a similar mind set to Lipsi.

A very early processor targeting FPGAs is the DOP processort [13]. DOP is a 16-bit stack oriented processor with additional registers, such as address registers and a work register. As this work register is directly connected to the ALU, DOP is similar to Lipsi an accumulator oriented architecture. No resource consumption is given for the DOP design.

Leros is, like Lipsi, an accumulator machine [3]. The machine word in Leros is 16-bit and Leros uses two on-chip memories: one for instructions and one for data. Therefore, Leros is organized as a two-stage pipeline and can execute one instruction every clock cycle. The Leros 16-bit architecture is powerful enough to run a small Java virtual machine [4].

3 The Lipsi Design

Lipsi is an 8-bit processor organized as an accumulator machine and has been designed and optimized around FPGA specific block RAMs. The focus of the design is to use just a single block RAM.

Different FPGA families contain differently organized and differently sized on-chip memories, which are also called block RAMs. The current minimum block RAM[2] is 4096 bits (or 512 bytes) large and has an independent read and write port. Lipsi is an 8-bit processor in its purest form. Therefore, we can use 256 bytes from that memory as instructions and 256 bytes for register and data. We use the lower half of the memory for the program, as the address register powers up at zero to fetch the first instruction.

Using a single block RAM for instructions and data means that this memory is time shared between instruction fetch and data read. Therefore, Lipsi is a sequential and not a pipelined architecture.

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

[2] This number is for relative old FPGAs, such as Xilinx Spartan-3 and Altera Cyclone II. Actual FPGAs from Xilinx have 16 Kbit and Altera have 8 Kbit memory blocks.

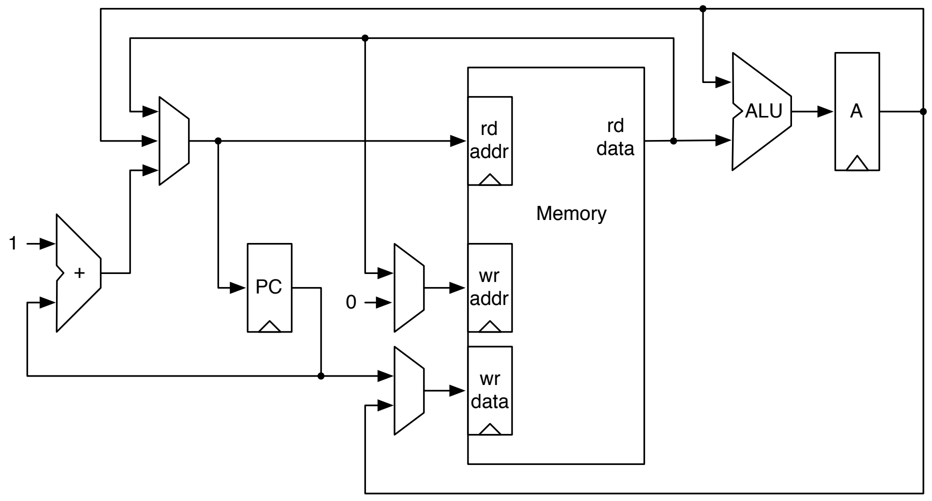

Fig. 1. The datapath of Lipsi.

Most instructions execute in two clock cycles: one for instruction fetch and one for data access and ALU operation. As on-chip memories in FPGAs usually have independent read and write ports, a store instruction can execute in a single cycle.

Most instructions are single byte. Only immediate and branch instructions contain a second byte for the immediate value or the branch target.

3.1 The Datapath

Figure 1 shows the datapath of Lipsi. The processor consists of a program counter (PC), an on-chip memory, an arithmetic-logic unit (ALU), and an accumulator register (A). Besides those basic components, one adder and three multiplexers are needed. The decode logic, which basically drives the multiplexers and the ALU function is not shown in the figure.

The memory is divided into three areas: (1) program area, (2) a register file, and (3) data memory. A single on-chip memory in most FPGAs is 512 bytes. This memory is split into two 8-bit addressable areas: one for instructions and one for data. The data area itself is split into 16 bytes treated specially as a register file, while the rest is for general data storage.

We can perform a back of an envelope estimation of the resource usage, the number of logic elements (LE). For each LE we assume a 4-bit lookup table for combinational logic and one register. As the design will be dominated by the logic used, we estimate the resource consumption based on combinational logic. The ALU supports addition and subtraction. With careful coding, it should be possible to implement both functions together in 8 LEs. Four logic functions (and, or, xor, and load) can be implemented in a single LE per bit. The shift operations should consume one LE per bit. The selection between adder/subtractor, the logic function, and the shift needs a 3:1 multiplexer with

Table 1. Lipsi instruction set with encoding

Encoding	Instruction	Meaning	Operation
`0fff rrrr`	*f* rx	ALU register	A = A *f* m[r]
`1000 rrrr`	st rx	store A into register	m[r] = A
`1001 rrrr`	brl rx	branch and link	m[r] = PC, PC = A
`1010 rrrr`	ldind (rx)	load indirect	A = m[m[r]]
`1011 rrrr`	stind (rx)	store indirect	m[m[r]] = A
`1100 -fff nnnn nnnn`	*fi* n	ALU immediate	A = A *f* n
`1101 --00 aaaa aaaa`	br	branch	PC = a
`1101 --10 aaaa aaaa`	brz	branch if A is zero	PC = a
`1101 --11 aaaa aaaa`	brnz	branch if A is not zero	PC = a
`1110 --ff`	sh	ALU shift	A = shift(A)
`1111 aaaa`	io	input and output	IO = A, A = IO
`1111 1111`	exit	exit for the tester	PC = PC

two LEs per bit. Therefore, the ALU should consume about 32 LEs. The adder will consume 8 LEs, the two 2:1 multiplexers each 8 LEs and the 3:1 multiplexer 16 LEs. This sums up to 64 LEs. Branch condition on zero or nonzero of A consumes 3 LEs Instruction decoding is performed on 4 bits, which fit into one LE. Therefore, 4 LEs are needed for the multiplexer driving and another LE for the PC register enable. The multiplexer and add/sub selector in the ALU decode from 3 function bits and need another 3 LEs. Therefore, Lipsi should consume around 84 LEs.

3.2 The Instruction Set

The instruction set of Lipsi includes ALU instructions with register and immediate operands, accumulator store, register indirect load and store, unconditional and conditional branch, branch and link for function call, and shift operations. Instruction length is one or two bytes.

Table 1 shows all instructions of Lipsi and their encoding. `A` represents the accumulator, *f* an ALU function, `PC` the program counter, `m[]` the memory, `r` a register number in the range of 0 to 15, `n` an immediate constant, `a` an 8-bit address, and `IO` an input/output device. As Lipsi is an accumulator machine, all operations (except unconditional branch) involve the accumulator register `A`. Furthermore, we use the notion of additional registers, which are the first 16 bytes in the data memory. Lipsi implements ALU operations with those registers and with immediate values. The accumulator `A` can be stored in any one of the registers. Memory load and store operations are implemented as register indirect. Those operations need three memory accesses: fetch the instruction, read the register content for the address, and finally load from memory into `A` or a store `A` in the memory. Register indirect load executes therefore in 3 clock cycles and an indirect store in 2 clock cycles.

Table 2 lists all ALU operations, including addition, subtraction, and logic operations. For an 8-bit architecture it is also useful to support addition with carry and subtraction with borrow for arithmetic on larger numbers. With careful coding these

Table 2. ALU operation and encoding

Encoding	Name	Operation
`000`	add	$A = A + op$
`001`	sub	$A = A - op$
`010`	adc	$A = A + op + c$
`011`	sbb	$A = A - op - c$
`100`	and	$A = A \wedge op$
`101`	or	$A = A \vee op$
`110`	xor	$A = A \oplus op$
`111`	ld	$A = op$

additional operations are almost for free (by adding one lower bit to the adder, setting one input to 1 and using the carry flag as second input). Furthermore, current FPGAs have an dedicated Xor gate in front of the LUT, so that an adder can also be used as subtractor (when using the additional input bit as well.).

Furthermore, three logic operations and a bypass operation for a load instruction are available. Again, we could be very minimalistic to support only a single inverting logic function, such as nand. However, implementation of these base operations is very cheap in an FPGA.

3.3 Implementation and Assembly in Hardware

For the implementation of Lipsi we use the relatively new hardware construction language Chisel [14]. In Chisel, the hardware is described in two classes: one for the processor and one for the memory. Describing the memory component in its own class allows future optimization to use an initialized memory (described in VHDL), which is currently not possible with Chisel.

The hardware abstraction level of Chisel is not so different from VHDL or Verilog. Hardware is described at the register transfer level. However, the power of Chisel lies in that Chisel is a language embedded in Scala [15], a modern general-purpose programming language. Scala itself runs on top of the JVM and can use libraries written in Java. Therefore, all these libraries and a modern object oriented and functional language are available at hardware construction time.

One of the first tools a processor developer needs is an assembler. A common approach is to write an assembler in some general-purpose language, e.g., Java, and spit out a VHDL table for the code that shall go into the ROM. This approach is also used for generating any hardware table which is needed, such as for function lookup or binary to binary-coded-decimal translation. As we can read in data with Scala and then generate a hardware table from Scala, the assembler can now instead generate a binary file that we read in at hardware construction time.

We have, however, gone a step further and have written the assembler itself in Scala, invoking it at hardware generation time, reading in the assembler code, and directly generating the hardware table to the ROM. With the power of the Scala `match` statement the assembler itself is just a handful of lines of code. Figure 2 shows this statement,

```
val tokens = line.trim.split(" ")
val Pattern = "(.*:)".r
val instr = tokens(0) match {
  case "#" => // comment
  case Pattern(l) => if (!pass2) symbols += (l.substring(0, l.length - 1) -> pc)
  case "add" => 0x00 + regNumber(tokens(1))
  case "sub" => 0x10 + regNumber(tokens(1))
  // and similar pattern
  case "addi" => (0xc0, toInt(tokens(1)))
  case "subi" => (0xc1, toInt(tokens(1)))
  // and similar pattern
  case "st" => 0x80 + regNumber(tokens(1))
  case "ldind" => 0xa0 + regIndirect(tokens(1))
  case "stind" => 0xb0 + regIndirect(tokens(1))
  case "br" => (0xd0, if (pass2) symbols(tokens(1)) else 0)
  case "brz" => (0xd2, if (pass2) symbols(tokens(1)) else 0)
  case "brnz" => (0xd3, if (pass2) symbols(tokens(1)) else 0)
  case "io" => 0xf0 + toInt(tokens(1))
  case "exit" => (0xff)
  case "" => // empty line
  case t: String => throw new Exception("Assembler error: unknown instruction")
  case _ => throw new Exception("Assembler error")
}
```

Fig. 2. The central statement of the Lipsi assembler in Scala

which is the core of the assembler. The full assembler is less than 100 lines of code and was written in a few hours.

3.4 Simulation and Testing

Chisel supports testing of hardware with a so-called tester. Within the tester one sets input signals with `poke`, advances the simulation by one clock cycle with `step`, and reads signals with `peek`. This is similar to a testbench in VHDL, except that the tester is written in Scala with the full power of a general purpose language available.

Furthermore, the tester also generates waveforms that can be inspected with ModelSim or gtkwave. We used this form of testing for the initial design.

As a next step, we wrote some test programs in assembly code with the convention that the test shall result in a zero in the accumulator at the end of the program. Furthermore, we defined an IO instruction to mark the end of the program. The testing against the zero in the accumulator has been integrated into the tester. With a handful of assembler programs we have, with minimal effort, achieved a first regression test.

As a further step, we have implemented a software simulator for Lipsi in Scala. The software simulator reuses the assembler that was written in the context of the hardware generation. Having a software simulator of Chisel opens up for testing of the hardware with co-simulation. As the hardware and the software simulator for Lipsi are all written in the same language (Scala with the Chisel library) it is possible to execute both together. Therefore, we also implemented a tester that executes the Lipsi hardware and the software simulation in lock step and compares the content of the program counter and the accumulator at every clock cycle. As all data will pass through the accumulator any error in the implementation (hardware of software simulator) will manifest itself at some stage as a difference in the accumulator.

The assembly of code and co-simulation of hardware and a software simulator in the very same language shows the power of Chisel as a hardware construction language. This usage of Chisel/Scala is probably just scratching the surface of new approaches to hardware design and testing.

With two implementations of Lipsi available, we can also explore random testing. As a next step, we plan to generate random byte patterns, which result in random instructions, and to compare the execution of the hardware and software simulator.

3.5 Developing a Processor

Although it is unusual to write about the history of the development in a scientific paper, we will provide here a brief history of the project. Actually, the development of Lipsi follows a pattern that we have observed several times. Therefore, the description of this development pattern for a moderately small digital design project, such as a processor, may be a contribution on its own.

Initially the processor was designed on paper in a notebook. Not really starting from scratch, as the author of this paper has designed several processors before. The Leros processor had been designed just a few days before, on paper as well. The pattern is that one often builds on previous designs. However, one should not restrict always oneself to reuse older designs, as this might restrict the design to not try entirely different approaches for a new system. And the author is convinced that a sketch of the datapath and some timing diagrams of execution traces on a piece of paper is important before coding any hardware.

From the detailed datapath design on paper, almost identical to Figure 1, and an initial instruction set encoding we started with coding of the hardware in Chisel. First we setup the infrastructure, describe a small part of the datapath in hardware, and started with simple testers (the name of test benches in Chisel).

From there we bootstrapped the implementation of the first instructions, the immediate instructions. We provide test code in very small programs as hexadecimal values in a static array that is then translated into a hardware table (ROM). First tests are manual checks with a Chisel tester (*printf debugging*) and manual inspection of waveforms. In parallel we also setup a Quartus FPGA project to observe the hardware cost development as we add features to the processor.

As manual assembly becomes too tedious, we developed an assembler. First just for the instructions that have already been assembled by hand for the test of the assembler by comparing with the manually generated instructions.

From that point in time on, the instructions in Lipsi and the assembler were developed in tandem. With more instructions being implemented, some automation of the testing is desirable. Especially some regression testing to make sure that newly added functionality does not break older functionality.

To perform some form of automated testing we need two functions: stopping of the test and an indication of success or failure. To stop the simulation (tester), we *invented* an `exit` instruction (which is just an IO instruction to a special address). For an indication of test success, we defined as success that the accumulator has to contain zero at the end of the test. All tests are written to have a dependent data flow from all operations into the accumulator. The tester checks at the end for zero and exits itself with an exit value different from zero when a test fails. This will also exit the `make` based automation of the testing code so we can observe the failure.

For further testing of Lipsi we wrote a software simulator of Lipsi, also in Scala. That software simulator is designed to be cycle accurate, modeling the timing of the Lipsi hardware. With that additional implementation we can perform co-simulation of the hardware description and the software simulator.

Now those tests are triggered manually with a `make` target. This project is too small for automated regression test. However, for larger projects, such as the Patmos project, we use nightly regression tests that follow a similar pattern.

Maybe this design flow with a relatively early automation of testing sounds like a lot of work and distracts from the fun of hardware design. The opposite is true. From the creation of the first file to contain Chisel code until the automation of the tests and implementation of around 2/3 of the functionality of Lipsi, just 8 hours have been spent on coding and testing. This very short development time was *because* of early automation with an assembler and smart testing not *despite* of it. The message of this subsection is to start early with very low effort automation and testing. Invest into the infrastructure of your project just what is needed at the moment.

4 Evaluation and Discussion

For the evaluation, we have synthesized Lipsi for a Cyclon IV FPGA, as this is the FPGA on the popular DE2-115 FPGA board. We used Quartus Prime Lite Edition 16.1 with the default settings and did not introduce any constraints related to the maximum clock frequency. Cyclone IV is the last generation of Cyclone FPGAs where a logic element (LE) contains a 4-bit lookup table (LUT).[3] Therefore, we can compare the resource numbers with designs on older FPGAs (e.g., Xilinx Spartan 3).

4.1 Resource Consumption

Table 3 shows the resource consumption in LEs and on-chip memory blocks, the maximum clock frequency, and the FPGA used for obtaining the results for different small processors. We synthesized Lipsi with a test program that slowly counts and puts the result on the LEDs. This configuration also contains one input port and one output port. Indeed, we can see that Lipsi is the smallest processor in this table. However, it is closely followed by Leros, which is a 16-bit, pipelined processor. With respect to the maximum clock frequency, Lipsi is in the same range as the other processors. We can see that the two pipeline stages of Leros result in a higher clock frequency than Lipsi where the critical path is in a memory read and an ALU operation.

[3] Newer generations is FPGAs use a 6-bit LUT, which can be split into two smaller LUTs.

Table 3. Comparison of Lipsi with Leros, PicoBlaze, Ultrasmall, and SpartanMC

Processor	Logic (LE)	Memory (blocks)	Fmax (MHz)	FPGA
Lipsi	162	1	136	Cyclone IV
Leros	189	1	160	Cyclone IV
PicoBlaze	177	1	117	Spartan 3
Ultrasmall	235	3	65	Spartan 3E
SpartanMC	1271	3	50	Sparten 3

The main reason why Lipsi is not even smaller is that with the current version of Chisel we cannot express an initialized block RAM. Therefore, the program is described in a table, which is then synthesized to logic. This logic for the instruction memory consumes 66 out of the 162 LEs. With the current work-around (using an on-chip memory and a logic table) we also need an additional multiplexer at the output of the memory component. Therefore, the processor core is smaller than 100 LEs.

As future work, we plan to describe the block RAM, including the initialization data, in VHDL or Verilog and instantiating it as a black box in Chisel. However, this solution is not very elegant as we mix languages and need to use different implementations of the memory for testing and synthesis. Another approach would be to extend Chisel to generate Verilog for initialized memory.

4.2 The Smallest Processor?

Is Lipsi now the smallest possible processor? No – if we really want a minimal implementation that can compute, we could drop several instructions. E.g., subtraction can be performed with xor and addition.

However, our target was a very small but useful processor. When we compare Lipsi with other processors, we think we have achieved that goal. With around 100 LEs and one block RAM we can fit many Lipsi cores into a low-cost FPGA.

4.3 A Lipsi Manycore Processor

We have explored how many Lipsi cores we can fit into the low-cost EP4CE115 FPGA from the DE2-115 board. Each processor contains one input and one output port. All processors are connected into a pipeline, which is the minimum useful connection of those processors. The first processors's input port is connected to the keys on the FPGA board and the last processor's output port is connected to the LEDs. Each processor reads the input, adds one to it, and puts the result to the output port.

The EP4CE115 contains 432 memory blocks. Therefore, we have configured 432 Lipsi processors in this computing pipeline. The resource consumption in the FPGA 67,130 is LEs out of 114,480 LEs, which is a resource consumption of 59 %. This shows that this kind of design is memory bound and we can add more functionality to the processor for a balanced use of the available resources.

This experiment is just meant as a proof of concept to build a manycore processor in a low-cost FPGA. Future work will be to use the remaining resources to add a simple network-in-chip to the 432 processor cores. This will enable more flexible communication paths and enable exploring network-on-chip designs within a high count of processing cores.

4.4 Lipsi in Teaching

The instruction set of Lipsi is so simple that it can be explained completely in this paper. However, it is complete enough to write useful programs. Therefore, we envision that Lipsi can serve as an example processor for a first semester introduction course in computer systems. Besides writing small assembler programs and running them on a simulator for Lipsi, writing a full simulator for Lipsi can serve as an exercise for a two-hour lab.

4.5 Source Access

We strongly believe in open-source designs for research, as far as legal possible. Especially when the research is funded by public funds, the full results (data and source code, not only a paper) shall be available to the public. Open-source enables independent researches to reproduce the published results. Furthermore, it also simplifies to build future research on top of the published research.

Lipsi's source is available at GitHub: `https://github.com/schoeberl/lipsi`. The `README.md` describes which tools are need to be installed and how to build Lipsi.

5 Conclusion

This paper presents Lipsi, a very tiny processor core. We believe that Lipsi is one of the smallest processors available. The intention of a small processor is to serve for auxiliary functions like an intelligent peripheral device, such as a serial port with buffering. Lipsi and the supporting assembler are all written in the same language, Chisel, which itself is based on Scala. This gives the power that the whole compilation flow from assembling the program till testing and hardware generation is driven by one description. Besides being a processor for peripheral devices, Lipsi can also serve as a small, but non-trivial example for the relatively new hardware construction language Chisel. Furthermore, as the processor structure is so simple that it can be drawn on half a page, it can also be used in an introductory course on computer architecture.

References

1. Schoeberl, M., Schleuniger, P., Puffitsch, W., Brandner, F., Probst, C.W., Karlsson, S., Thorn, T.: Towards a time-predictable dual-issue microprocessor: The Patmos approach. In: First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011), Grenoble, France (March 2011) 11–20

2. Schoeberl, M., Abbaspour, S., Akesson, B., Audsley, N., Capasso, R., Garside, J., Goossens, K., Goossens, S., Hansen, S., Heckmann, R., Hepp, S., Huber, B., Jordan, A., Kasapaki, E., Knoop, J., Li, Y., Prokesch, D., Puffitsch, W., Puschner, P., Rocha, A., Silva, C., Sparsø, J., Tocchi, A.: T-CREST: Time-predictable multi-core architecture for embedded systems. Journal of Systems Architecture **61**(9) (2015) 449–471
3. Schoeberl, M.: Leros: A tiny microcontroller for FPGAs. In: Proceedings of the 21st International Conference on Field Programmable Logic and Applications (FPL 2011), Chania, Crete, Greece, IEEE Computer Society (September 2011) 10–14
4. Caska, J., Schoeberl, M.: Java dust: How small can embedded Java be? In: Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2011), York, UK, ACM (September 2011) 125–129
5. Altera Corporation: Nios II processor reference handbook. Available from `http://www.altera.com/literature/lit-nio2.jsp` (May 2011) Version NII5V1-11.0.
6. Xilinx: PicoBlaze 8-bit embedded microcontroller user guide (2010)
7. 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), IEEE (2007) 51–58
8. Hennessy, J.L.: VLSI processor architecture. Computers, IEEE Transactions on **C-33**(12) (Dec. 1984) 1221–1246
9. 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, Washington, DC, USA, IEEE Computer Society (2002) 409
10. Xilinx Inc.: MicroBlaze processor reference guide (2008) Version 9.0.
11. Robinson, J., Vafaee, S., Scobbie, J., Ritche, M., Rose, J.: The supersmall soft processor. In: Programmable Logic Conference (SPL), 2010 VI Southern. (march 2010) 3 –8
12. 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). (Sept 2014) 1–4
13. Danecek, J., Drapal, F., Pluhacek, A., Salcic, Z., Servit, M.: Dop—a simple processor for custom computing machines. Journal of Microcomputer Applications **17**(3) (1994) 239 – 253
14. 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 Groeneveld, P., Sciuto, D., Hassoun, S., eds.: The 49th Annual Design Automation Conference (DAC 2012), San Francisco, CA, USA, ACM (June 2012) 1216–1225
15. Venners, B., Spoon, L., Odersky, M.: Programming in Scala, 3rd Edition. Artima Inc (2016)