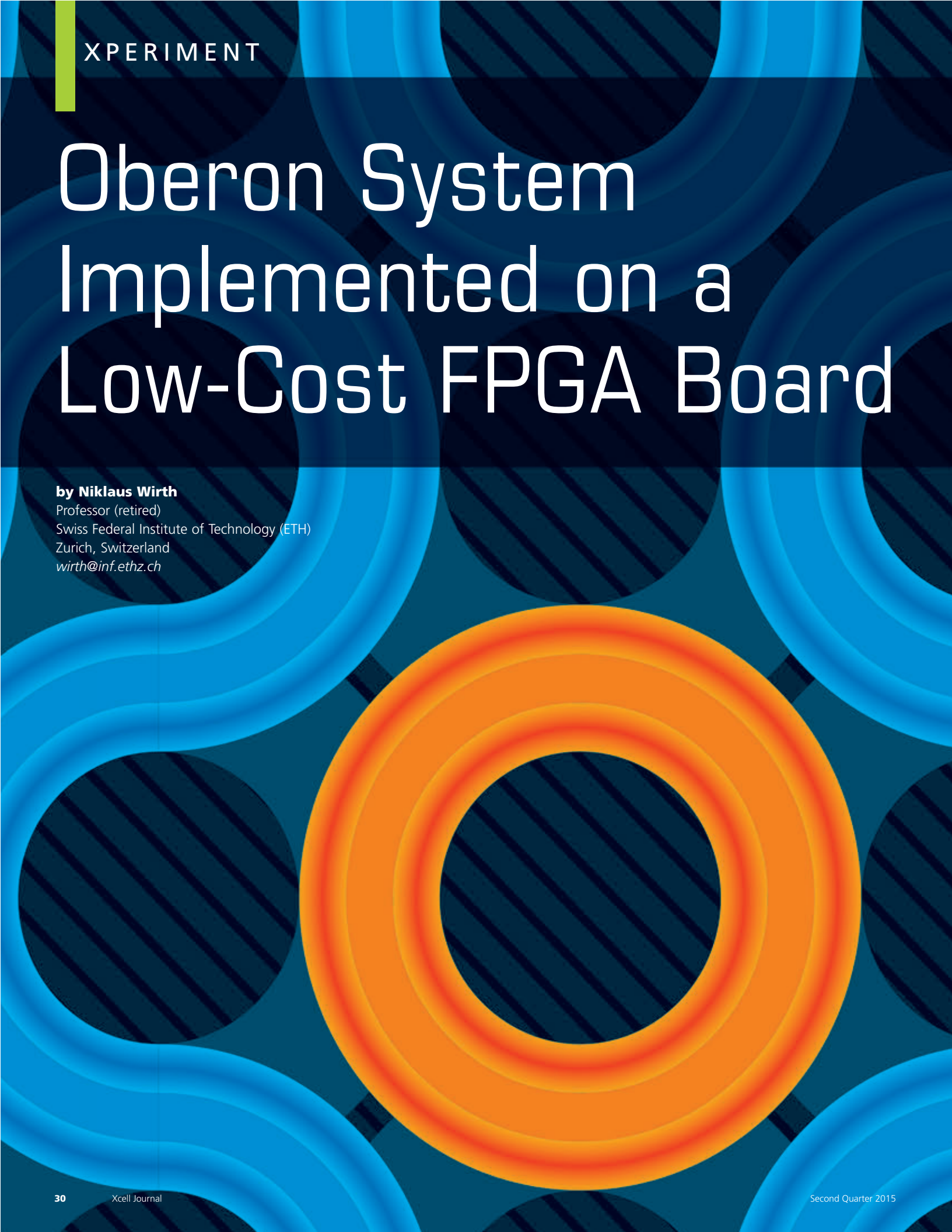
Oberon System Implemented on a Low-Cost FPGA Board

by Niklaus Wirth
Professor (retired)
Swiss Federal Institute of Technology (ETH)
Zurich, Switzerland
wirth@inf.ethz.ch



A Xilinx Spartan-3 board becomes the basis for revamping the author's Oberon programming language and compiler for use in software education.

In 1988, Jürg Gutknecht and I completed and published the programming language Oberon [1,2] as the successor to two other languages, Pascal and Modula-2, which I had developed earlier in my career. We originally designed the Oberon language to be more streamlined and efficient than Modula-2 so that it could better help academics teach system programming to computer science students. To advance this endeavor, in 1990 we developed the Oberon operating system (OS) as a modern implementation for workstations that use windows and have word-processing abilities. We then published a book that details both the Oberon compiler and and the operating system of the same name. The book, entitled *Project Oberon*, includes thorough instructions and source code.

A few years ago, my friend Paul Reed suggested that I revise and reprint the book because of its value for teaching system design, and because it serves as a good starting point to help would-be innovators build dependable systems from scratch.

There was a big obstacle, however. The compiler I originally developed targeted a processor that has essentially disappeared. Thus, my solution was to rewrite a compiler for a modern processor. But after doing quite a bit of research, I couldn't find a processor that satisfied my criteria for clarity, regularity and simplicity. So I designed my own. I was able to bring this idea to fruition because of the modern FPGA, as it

Xcell Journal *is honored to publish this article by industry legend Niklaus Wirth, who invented Pascal and several derivative programming languages and has pioneered some classic approaches to computer and software engineering. A recipient of the ACM Turing Award and of the IEEE Computer Pioneer Award, Professor Wirth has retired from teaching but continues to help educators develop and inspire the innovators of tomorrow.*

Choosing a Xilinx FPGA allowed me to update the system while keeping the design as close as possible to the original version from 1990.

allowed me to design the hardware as well as the system software. What's more, choosing a Xilinx® FPGA allowed me to update the system while keeping the design as close as possible to the original version from 1990.

The new processor is called RISC, and it was implemented on the low-cost Digilent Spartan®-3 development board, hosting a 1-Mbyte static RAM (SRAM) memory. The only system hardware additions I made were to add an interface for a mouse and an SD card to replace the hard-disk drive in the older system.

The book and the source code for the entire system are available on *projectoberon.com* [3,4,5]. Also available at the same site is a single file called S3RISCinstall.zip. This file contains instructions, an SD-card filesystem image and FPGA configuration bit files (in the form of PROM files for the Spartan-3 board's Platform Flash), together with construction details for the SD-card/mouse interface hardware.

THE RISC PROCESSOR

The processor consists of an arithmetic-logic unit; an array of 16 registers of 32 bits; and a control unit with instruction register, IR and program counter PC. The processor is represented by the Verilog module RISC5.

The processor features 20 instructions: four for moving, shifting and rotating; four for logic operations; four for integer arithmetic; four for floating-point arithmetic; two for memory access; and two for branching.

RISC5 is imported by RISC5Top, the environment, which contains the interfaces to various (memory-mapped) devices and the SRAM (256M x 32 bit). The entire system (Figure 1) consists of the follow-

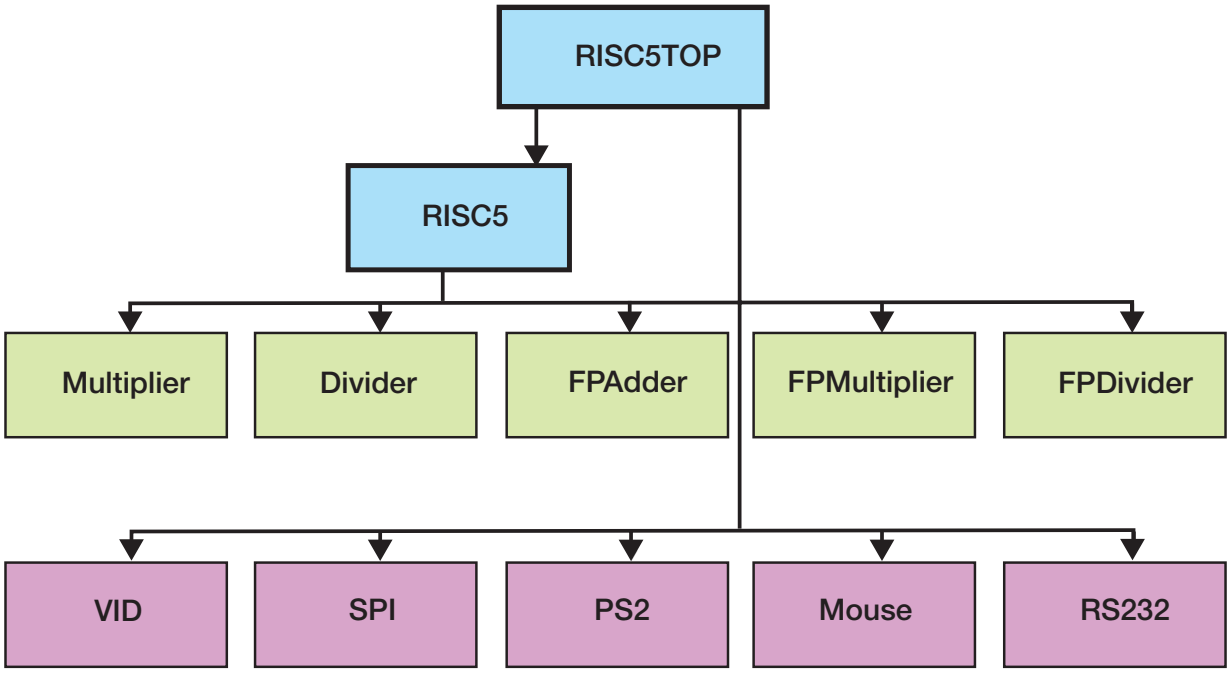


Figure 1 – Diagram of the system and the Verilog modules it contains

RISC5Top	environment	194
RISC5	processor	201
Multiplier	integer arithmetic	47
Divider		24
FPAdder	floating-point arithmetic	98
FPMultiplier		33
FPDivider		35
SPI	SD card and transmitter/receiver	25
VID	1024 x 768 video controller	73
PS2	keyboard	25
Mouse	mouse	95
RS232T	RS232 transmitter	23
RS232R	RS232 receiver	25

ing Verilog modules (line counts shown):

I memory-mapped the black-and-white VGA display so that it occupies 1024 x 768 x 1 bit per pixel = 98,304 bytes, which is essentially 10 percent of the available main memory of 1 Mbyte. The SD card replaces the 80 Mbytes in the original system. It is accessed over a standard SPI interface, which accepts and serializes bytes or 32-bit words. The keyboard and the mouse are connected by standard, serial PS-2 interfaces. Furthermore, there is a serial, asynchronous RS-232 line and a general-purpose, 8-bit parallel I/O interface. Module RISC5Top also provides a counter, incremented every millisecond.

THE OBERON OPERATING SYSTEM

The operating system software consists of a core that includes a memory allocator with a garbage collector and a file system, along with the loader, a text system, a viewer system and a text editor.

The module called "Oberon" is the central task dispatcher while "System" is the basic command module. An action is evoked by clicking the middle button on the text "M.P" in any viewer on the display, where P is the name of a procedure declared in module M. If M is not present, it is automatically loaded. Most text-editing commands, however, are evoked by simple mouse clicks, where the left button serves to set a caret, marking a text position, and the right button serves to select a text stretch.

The module "Kernel" contains the disk-store management and the garbage collector. I ensured that the viewers were tiled and do not overlap. The standard layout shows two vertical tracks with any number of viewers. You can enlarge them, make them smaller or move them by simply dragging their title bars. Figure 2 shows the user interface running on the monitor, along with the Spartan-3 board, keyboard and mouse.

The system when loaded occupies 112,640 bytes in module space (21 percent) and 16,128 bytes of the heap (3 percent). It consists of the following modules (line counts shown), as depicted in Figure 3:

Kernel	271	(inner core)
FileDir	352	
Files	505	
Modules (loader)	226	
Viewers	216	(outer core)
Texts	532	
Oberon	411	
MenuViewers	208	
TextFrames	874	
System	420	
Edit	233	

It is remarkable that system initialization at power-on or reset takes only about 2 seconds. This includes a garbage-collecting scan of the file directory.

THE OBERON COMPILER

The compiler, hosted on the system itself, uses the simple top-down recursive-descent parsing method. You can activate the compiler on a module's selected source text by using the command `ORP.Compile @`. The parser inputs symbols from the scanner delivering identifiers, numbers and special symbols (like BEGIN, END, + and so on). This scheme has proven to be useful and elegant in many applications. It is described in detail in my book *Compiler Construction* [6,7].

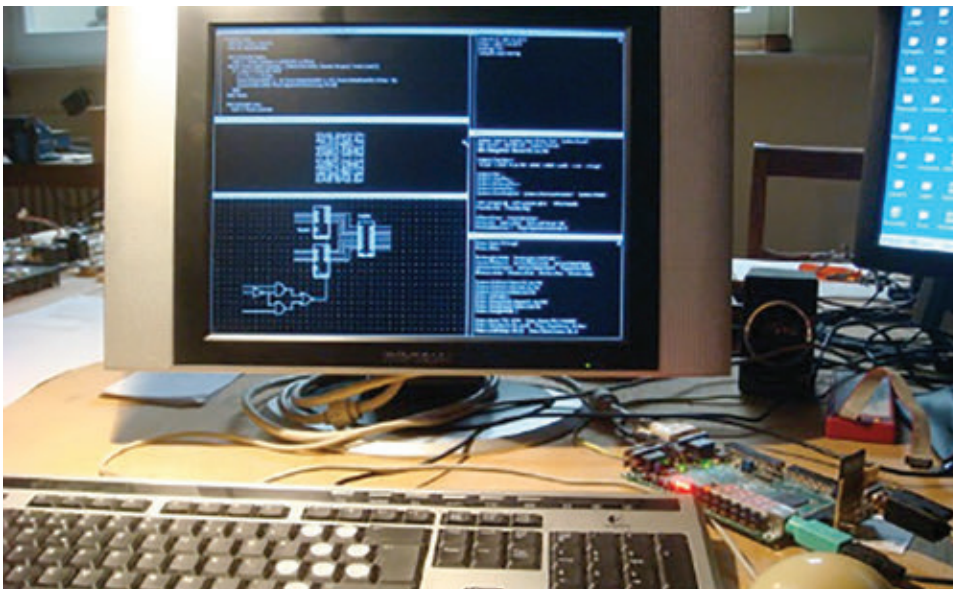


Figure 2 – Monitor showing user interface, with Spartan-3 board at right

The parser calls procedures in the code generator module. They directly append instructions in a code array. Forward-branch instructions are provided with jump addresses at the end of a module's compilation (fixups), when all branch destinations are known.

All variable addresses are relative to a base register. This is R14 (stack pointer) for local variables (set at procedure entry at run-time), or R13 for global and imported variables. The base addresses are loaded on demand from a system-global module table, whose address is held in register R12. R15 is used for return addresses (link), as determined by the RISC architecture. Thus, R0 - R11 are available for expression evaluation and for passing procedure parameters.

The entire compiler consists of four relatively small and efficient modules (line counts shown):

ORP	parser	968
ORG	code generator	1120
ORB	base def	435
ORS	scanner	311

The compiler occupies 115,912 bytes (22 percent) of the module space and 17,508 bytes (4 percent) of the heap (before any compilation). Its source code is about 65 kbytes long. Compilation of the compiler itself

The Lola HDL and its Translation to Verilog

The hardware-description language (HDL) called Lola was defined in 1990 as a means for teaching the basics of hardware design. This was the period when textual definitions started to replace circuit diagrams, and when the first FPGAs became available, although had not yet reached mainstream design. Lola was implemented by a compiler that generated bit files to be loaded onto FPGAs. Bit-file formats were disclosed by Algotronix Inc. and Concurrent Logic Inc. Both featured cells of rather simple structures, which seemed to be optimal for automatic placement and routing.

In the wake of my project to reimplement Oberon on an FPGA, the idea now popped up also to revive Lola. Because the cells of Xilinx's FPGAs feature a rather complex structure, we did not venture into the effort of implementing placement and routing, quite apart from the fact that Xilinx refuses to disclose its bit-file format for proprietary reasons.

The obvious path was to build a Lola compiler that does not generate proprietary bit files, but translates into a language for which Xilinx provides a synthesis tool. We chose Verilog. This solution implies a rather extravagant detour: First, a Lola module has to be parsed, then translated, then parsed again. With all these steps, we ensured that the Lola compiler had proper error reporting and type-consistency checking capabilities.

To drive the development of Lola-2, we had the ambition to reformulate in Lola all modules of the RISC5 processor. This has now been achieved.

THE LOLA LANGUAGE

Lola is a small, terse language in the style of Oberon (see *http://www.inf.ethz.ch/personal/wirth/Lola/Lola2.pdf*).

For the sake of brevity, we show here only a single example of a Lola text (Figure 1). The unit of source text is called a module. Its heading specifies the name and its input and output parameters with their names and types. The heading is followed by

```
MODULE Counter0 (IN CLK50M, rstIn: BIT;
  IN swi: BYTE; OUT leds: BYTE);

TYPE IBUFG := MODULE (IN I: BIT; OUT O: BIT) ^;
VAR clk, tick0, tick1: BIT;
  clkInBuf: IBUFG;
REG (clk) rst: BIT;
  cnt0: [16] BIT;  (*half milliseconds*)
  cnt1: [10] BIT;  (*half seconds*)
  cnt2: BYTE;

BEGIN leds := swi.7 -> swi : swi.0 -> cnt1[9:2] : cnt2;
  tick0 := (cnt0 = 49999);
  tick1 := tick0 & (cnt1 = 499);
  rst := ~rstIn;
  cnt0 := ~rst -> 0 : tick0 -> 0 : cnt0 + 1;
  cnt1 := ~rst -> 0 : tick1 -> 0 : cnt1 + tick0;
  cnt2 := ~rst -> 0 : cnt2 + tick1;
  clkInBuf (CLK50M, clk)
END Counter0.
```

Figure 1 — Lola text showing a counter counting milliseconds and seconds, displaying the latter on the board's LEDs

takes only a few seconds on the 25-MHz RISC processor [8].

The compiler always generates checks for array indices and for references through pointers with value NIL. It causes a trap in case of violation. This technique ensures a high degree of security against errors and corruption. In fact, system integrity can be violated only by the use of operations in the pseudo-module SYSTEM, namely PUT and COPY. These operations must be restricted to driver modules accessing device interfaces, and they are easily recognized by SYSTEM in their import lists. The entire system is programmed in Oberon itself, and no use of assembler code is required.

I chose the Digilent Spartan-3 development board because of its low cost and simplicity, which makes it suitable for educational institutions to acquire sets of the kits for classrooms. An important advantage is the presence of static RAM on this board, which makes interfacing straightforward (even for byte selection). Unfortunately, newer boards all feature dynamic RAM, which is, albeit larger, very much more complicated to interface, requiring circuits for refresh and initialization (calibration). This circuitry alone can be as complex as the entire processor with static RAM. Even if a controller is provided on-chip,

a section containing declarations of local objects, such as variables and registers. Then follows the section defining the values of variables and registers through assignments. BYTE denotes an array of 8 bits.

THE LOLA COMPILER

The compiler uses the simple top-down recursive-descent parsing method. It is activated on the selected Lola source text by the command `LSC.Compile @`. The parser inputs symbols from the scanner delivering identifiers, numbers and special symbols (like BEGIN, END, +, etc.). This scheme has proved to be useful and elegant in many applications. It is described in the book *Compiler Construction (Part 1* and *Part 2)*.

Instead of generating Verilog texts directly on the fly, statements are present in the parser which, as a side effect of parsing, generate a tree representing the input text in a way appropriate for further processing. This structure has the advantage that various, different outputs may easily be generated by calling on different translators. One of them is a translator to Verilog. The command is `LSV.List outputfile.v`. Another command may translate to VHDL, or simply list the tree. Yet another might generate a netlist for further processing by a placer and router.

Thus, the entire compiler consists of at least four relatively small and efficient modules (line counts shown):

LSS	scanner	159
LSB	base	52
LSC	compiler/parser	503
LSV	Verilog generator	215

Instructions on translating from Lola to Verilog can be found at *http://www.inf.ethz.ch/personal/wirth/Lola/Lola-Compiler.pdf*.

THE DIFFERENCE BETWEEN SOFTWARE AND HARDWARE 'PROGRAMS'

Many efforts have been undertaken in the past to make HDLs look like "ordinary" programming languages (PLs). We also note that HDLs typically have a counterpart in the set of PLs, adopting the respective style. Thus, Verilog has its ancestor in C, VHDL in Ada and Lola in Oberon. We consider it important to recognize the fundamental differences between the two classes, in particular in the presence of syntactic similarities or even identities. What are these fundamental differences?

In order to simplify an explanation, we restrict our analysis to synchronous circuits—that is, circuits in which all registers tick with the same clock. It is indeed a healthy design paradigm to stick to synchronous circuits in general, if possible. Then, quite obviously, all elements of a circuit operate concurrently, literally at the same time. Every variable and register is defined by one and only one expression (combinational circuit). Multiple assignments do not make sense. We can simply imagine every HDL program to be enclosed in a big repeat-forever clause, because the assignments to registers and variables are repeated in every clock cycle.

It was the ingenious idea of John von Neumann to introduce a processor architecture with a sequencer. The sequencer contains an instruction register, according to which (in every cycle) certain circuits are selected and others ignored, thus cleverly reusing the different parts (of the ALU). Now, cycles or steps have become inherently sequential, and it is possible to reassign values to the same variables, as the program counter relates them to certain places in the program and in the instruction sequence. It is this idea of the sequencer that makes it possible to execute enormous programs by relatively simple circuits.

In summary, Lola-2 is an HDL in the style of the PL Oberon. The compiler presented here translates Lola modules into Verilog modules. Lola's advantages lie in the language's simple and regular structure, and in the compiler's emphasis on type checking and improved error diagnostics. The entire set of modules for the RISC processor have been expressed in Lola (see *http://www.inf.ethz.ch/personal/wirth/Lola/index.html*).

— Niklaus Wirth

this counteracts our principle of laying everything open for inspection.

CLOSING THOUGHTS

More than 40 years ago, C.A.R. Hoare remarked that in all branches of science and technology, students are exposed to many master examples of design before they are asked to experiment with their own attempts. Programming and software design stood in marked contrast to this sensible paradigm. Here the student was asked to write programs right from the start, before having read any examples.

The reason for this dire fact was that there existed almost no literature with sizable master examples. I therefore decided to remedy the situation somewhat, and I wrote the book on *Algorithms and Data Structures* in 1975. Subsequently (with J. Gutknecht), having been charged with the task of teaching a course on operating systems, I designed the system Oberon (1986-88).

Since then, the teaching of programming has not markedly improved, whereas systems have dramatically increased in size and complexity. Although the open-source endeavor is to be welcomed, it has not really changed the situation, since most programs have been constructed "to run," but not really for human consumption, for being understood.

I continue to make the daring proposal that all programs should be designed not just for computers, but for human reading. They should be publishable. This is a task much harder than creating executable programs, even correct and efficient ones. It implies that no part must be specified in assembler code.

The result of ignoring this "human factor" is that in many places new applications are not carefully designed, but rather debugged into existence, sometimes with dismal consequences. An important ingredient to achieve understandability is to adhere to simplicity and regularity, and to renounce unnecessary embellishments, to avoid bells and whistles and to distinguish between conventional and convenient.

The small size of this system is witness to *how much can be achieved with how little*. The Oberon system's dimensions are ridiculously small compared with most modern operating systems, although it includes a file system, a text editor and a viewer (windows) management. The side ef-

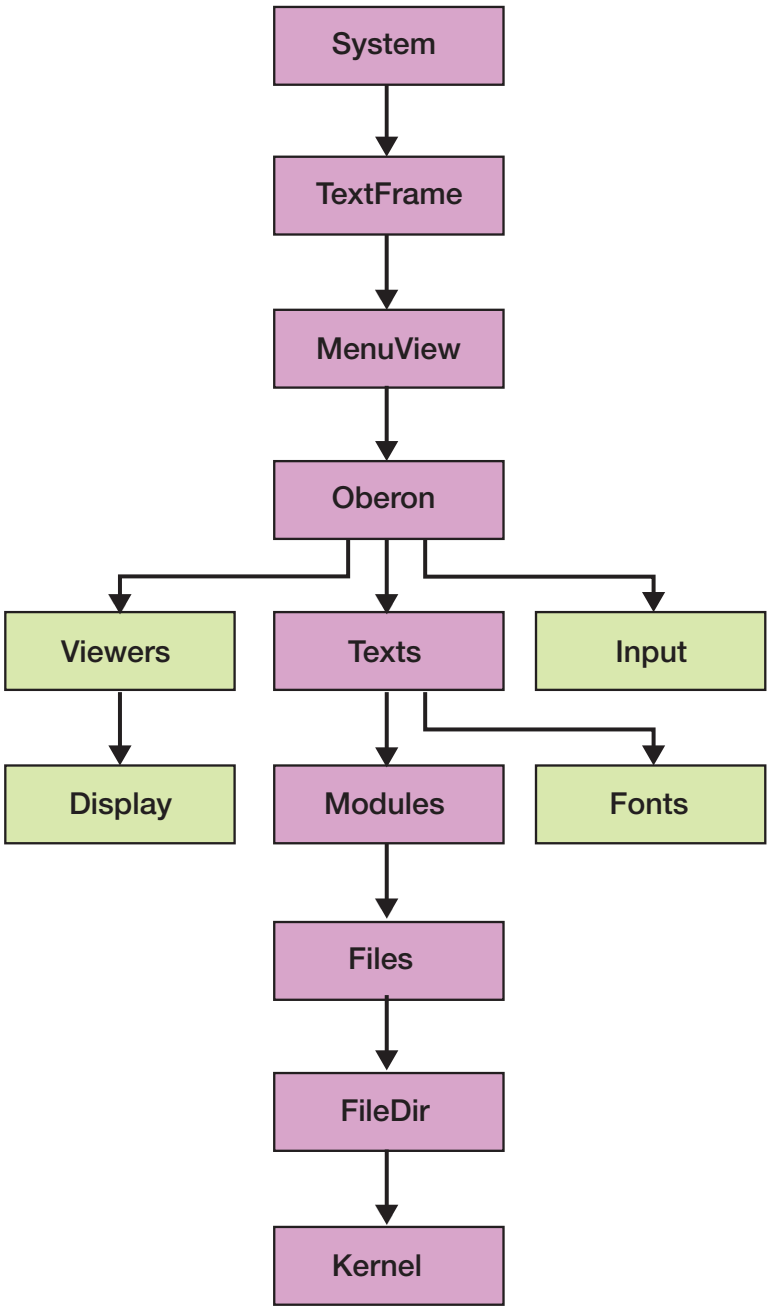


Figure 3 – The system and its modules

fect is that it rests on just a few simple rules, and that therefore it is easy to learn how to use it.

Finally, another advantage of this terseness is that one can safely build upon this basic system without being afraid of the existence of unknown features, such as back doors. This is an essential property in view of the increasing danger of attacks on the integrity of a system, indispensable for safety-critical applications. Notably, also the hardware of our system is free of such hidden parts. After all, no guarantee can be given for any system that is built upon a base that is so large that nobody can understand it in its entirety.

ACKNOWLEDGEMENT

I gratefully acknowledge the invaluable contributions of Paul Reed. He suggested that I re-edit the book Project Oberon *and also suggested reimplementing the entire system on an FPGA. Paul was an essential source of encouragement. He thought of replacing the disk with an SD card, and he contributed the SPI, the PS-2 and the VID interfaces in Verilog.*

References

1. *http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon07.Report.pdf*
2. *http://www.inf.ethz.ch/personal/wirth/Oberon/PIO.pdf*
3. *www.inf.ethz.ch/personal/wirth/ProjectOberon/index.html*
4. *www.inf.ethz.ch/personal/wirth/Oberon/PIO.pdf*
5. *www.inf.ethz.ch/personal/wirth/Oberon/Oberon07.Report.pdf*
6. *http://www.inf.ethz.ch/personal/wirth/CompilerConstruction/CompilerConstruction1.pdf*
7. *http://www.inf.ethz.ch/personal/wirth/CompilerConstruction/CompilerConstruction2.pdf*
8. *http://www.inf.ethz.ch/personal/wirth/ProjectOberon/PO.Applications.pdf* (Ch. 12)