NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4501 - Computer Science, Specialization Project

Porting the Oberon system to the RISC-V instruction set architecture

Author:
Rikke Solbjørg

December, 2020

Contents

List of Figures

List of Tables

1 Introduction

A common adage within computer science is that "software is getting slower more rapidly than hardware becomes faster"[5]. As software grows in size and complexity, they become more demanding of hardware, and of their users if they wish to comprehend it. Operating systems are not excepted from this issue, and commonly used operating systems such as GNU/Linux and Windows have seen similar increases in complexity over time.

The Oberon system is an operating system developed by Niklaus Wirth and Jürg Gutknecht, designed to show that this trend is unnecessary. While it lacks many of the niceties of modern operating systems, it is fully featured, including both a file editor and a compiler. It does not contain, say, a web browser, but does have facilities for exchanging files between workstations over a network connection. On the whole, it is very small, and the entire operating system, along with the compiler, occupies less than $200KB$ when compiled[8]. However, it runs on an instruction set architecture (ISA) defined specifically for this operating system; as such, its latest edition is not supported by commercial hardware, instead requiring its users to put a compatible processor on a field-programmable gate array to run it.

This project focused on porting the Oberon system from its own instruction set architecture to another. For the focus of this report, the instruction set architecture mainly defines how the programmer interacts with hardware: the number of registers available, the operations the programmer can instruct the processor to perform, and so on. These decisions have implications for hardware design, but that is outside the scope of this report. Rather, the focus will be on the implications it has for *software*. Here, the targeted instruction set is RISC-V, which has seen increased adoption in recent years as a free alternative to other ISAs.

This is the foundation for the research question this report will focus on answering: *What will have to be done differently to make the operating system function on an instruction set architecture besides its own? Will it run as efficiently, and take up as little space?*

This report will first relate important background and prerequisites to the project in section 2; more detail on the two instruction set architectures discussed in the report is given in sections 2.1 and 2.3.1. Next, the report will answer the first question, i.e. what must be done differently. Section 3 goes into detail on how to emulate both instruction set architectures. The emulator allows for running the ported operating system; the process of porting the compiler as well as the operating system is described in sections 4 and 5 respectively. How to build the ported operating system into an image that can be run by an emulator or by hardware is given in section 6. Finally, to answer the second question, an evaluation of the system in RISC-V compared to in RISC-5 is given in section 7, before concluding in section 8.

2 Background

2.1 RISC-V

RISC-V is a rather new instruction set architecture, the goal of which is to become completely universal[2]. In other words, it should be able to accomodate all possible cores that desire to implement it, whether they are in-order or out-of-order; as well as all technologies a core can be fabricated with, whether it's on an FPGA or ASIC.

It is unique for many reasons, and not all of them will be recounted here. Of particular importance to this project, RISC-V is an example of a RISC (reduced instruction-set computing) design, meaning it favours combining multiple simple instructions to do something complex, as opposed to performing the complex instruction in hardware. This is as opposed to a CISC (complex instruction-set computing) design, wherein the processor understands many more instructions that can perform very specific operations.

Another important aspect to take heed of in this project is that it is a *modular* ISA[2]. It offers a basic set of instructions that every RISC-V processor is guaranteed to implement, and then a set of extensions that a processor can choose to implement depending on what it targets. For instance, a small implementation might use `RV32I`; the RV signifies that it's RISC-V, *32* signifies a 32-bit processor, and *I* signifies the most basic extension, which includes only instructions essentially deemed necessary, 47 in total. Other extensions can be added on top of this; for instance, instructions for multiplication and division are defined in the *M* extension; an implementation that also includes these instructions would be a `RV32IM` processor. There are many other extensions, such as F and D for single- and double-precision floating point respectively.

For this project, a `RV32IM` architecture was targeted, as it is a reasonable architecture for embedded applications, including the kinds of constraints under which Oberon performs well. As mentioned, that means basic instructions, as well as multiplication and division. Notably, it also means that floating-point operations are not supported by hardware.

Another feature offered by RISC-V is various modes that signify different levels of privilege. It offers three: machine-mode, supervisor-mode, and user-mode[4]. While many modern operating systems require different privilege levels, this is not necessary for the operating system targeted in this project. As such, a system using only machine-mode was targeted, often ideal for simple embedded systems[4].

2.2 Lean software

The development of the Oberon system is not driven by the same interests as e.g. GNU/Linux or MS Windows. Niklaus Wirth is, among other accomplishments, famous for his dedication to lean software, as opposed to "fat software". This dedication has influenced much of the decision making in the design process of the

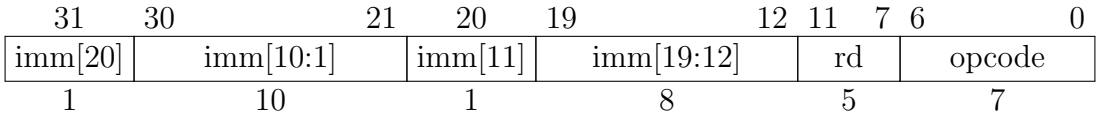


Figure 1: The J-type instruction format in RISC-V[2].

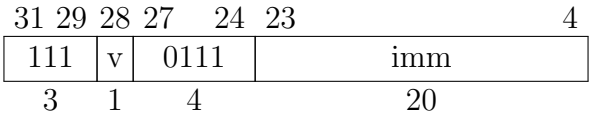


Figure 2: The branch instruction format in RISC-5, with condition set to be an unconditional jump[7]. The v bit controls whether the return address is deposited or not, and can essentially be ignored for the purposes of this figure.

language, compiler, and operating system as a whole, which stands as an example of Wirth's tenets[5]. Among some of the core principles outlined, perhaps of most interest to Oberon is the following: "A system that is not understood in its entirety, or at least to a significant degree of detail by a single individual, should probably not be built." In contrast to modern operating systems, such as GNU/Linux, the Oberon system is small enough that, with enough effort, an individual can have a complete understanding of the entire codebase.

2.3 The building blocks of the Oberon system

Here, the pieces underpinning the Oberon system are explained in some detail. Take special care to note that both the instruction set architecture, the programming language, and the operating system are co-designed, leading to a high degree of compatibility between them.

2.3.1 RISC-5

From 2013 and onwards, Wirth started working on a revised edition of the Oberon system, named *Project Oberon 2013* or simply *Project Oberon*. In this document, it will be referred to as *the Oberon system* to distinguish it from the programming language of the same name. While overall quite similar to previous versions, a major difference in Project Oberon 2013 is the use of a new processor, along with a new instruction set architecture. Rather than use an already extant instruction set architecture – say, the ARM architecture – Wirth opted to design his own[8]. Rather confusingly for this report, the latest and most stable version of this architecture is named *RISC-5*, and the reader must take care not to confuse this with *RISC-V*.

Wirth defined RISC-5 to be as simple as possible, although some care must be made to specify that this simplicity is most easily appreciated from a software perspective, rather than from a hardware perspective. A short, illustrative example will be helpful.

RISC-V was designed to reduce hardware complexity - for instance, the sign bit of

an immediate is always placed in bit 31 in every instruction encoding, to "speed sign-extension circuitry"[3]. This is part of the reason why the J-type instruction in RISC-V looks so much more complex, cf. figure 1. In contrast, the equivalent instruction in RISC-5 looks a lot simpler, cf. figure 2. Indeed, the contiguous immediate is much simpler to work with for e.g. a compiler, but this comes at the cost of disallowing some hardware optimisations. Some of the costs of this increased complexity in immediate encoding in RISC-V are discussed in more detail in section 5.3.

2.3.2 The Oberon programming language

The Oberon programming language, designed by Niklaus Wirth, is currently the latest language in Wirth's family of Algol-like languages[6]. In contrast to e.g. C, it is a far more strictly typed language, with static typing, while still being suited for systems programming. An example of this strictness is that, while the programming language does have pointers, it only allows pointers to records (which are similar to structs in C). Of particular note here, the Oberon programming language is quite minimal, but has enough features that the entirety of the Oberon system can be written in it.

2.3.3 The Oberon system

The Oberon system began development in 1986, with a primary goal to be both (1) a fully comprehensive system, and (2) to be able to be understood as a whole by a single person[8]. In other words, it must be simple enough that all parts of it can be comprehended by a single person all at once, but not so simple that it isn't useful.

The Oberon system accomplishes this by avoiding large amounts of optimisation where it isn't necessary, such that no part becomes so complex as to require a lot of study, and keeping the set of features minimal.

Another important aspect is the fact that in its latest revision, i.e. Project Oberon 2013, the system has been designed alongside its instruction set architecture, RISC-5. This allows it to take some shortcuts both in the design of both the compiler and the operating system, wherein parts of the ISA have been designed for easier use in the Oberon system. This will become more evident in section 4.

2.4 Compiler Construction

In deciding upon solutions to problems that arose in porting the compiler to RISC-V, some effort was taken not to break entirely with the design philosophy behind it. A core component of this design philosophy is an emphasis on simplicity; one can see this both in the programming language itself, as well as in some of the techniques used – and more importantly, some of the techniques that were explicitly decided against.

The compiler performs very few optimisations as a result of this simplicity, as Wirth focused more on the compiler running quickly rather than creating optimised programs. A simple metric, in part popularised by Wirth, is the *speed of self-compilation*[1]. This metric suggests that the only increase in complexity desirable in a compiler is one that improves compilation speed enough that it makes up for the increased code size required to express the optimisation.

3 Emulation

This section will focus on the tools used to emulate the computer running the Oberon system. Although many different emulators could be used, for ease of development a simple one tailor-made for the Oberon system was put in use, created by Peter de Wachter[1]. The basics of how this emulator works are described in section 3.1. Then, the process of making it emulate RISC-V is described in section 3.2.

3.1 RISC-5 emulator

The RISC-5 emulator is fairly simple, and focused on making Oberon run efficiently on another platform, rather than accuracy. For instance, it does not emulate stalling while waiting for input, as that would be very noticeable, but instead skips emulation if it notices the Oberon system continually reading I/O ports without receiving any new input. Similarly, it also does not simulate e.g. memory latency, bandwidth, etc., and instead immediately returns data read from memory without delays.

3.2 Converting the emulator to RISC-V

Making the Oberon emulator work for RISC-V was fairly simple. The only issues that needed solving were to add an emulator for RISC-V, and to make that emulator interface with the rest, so that display, input, etc. was handled properly. At first, a custom emulator was written, mostly to become more comfortable with the RISC-V instruction set. While this worked fine for most cases, it failed in certain edge cases. Therefore, for most of the project, an already existing RISC-V emulator was integrated instead; this to avoid having to debug a compiler, an operating system, and an emulator all at once. For this purpose, a RV32I emulator by Ted Fried[2] was chosen, as its small size made it easy to integrate. This did not take too long, and worked for the most part. The instructions found in the `M` extension for RISC-V also had to be added, but as this extension only adds eight instructions, this did not present a large obstacle. Another issue, that was only noticed when it caused errors in the Oberon system, is that this emulator did not handle LB (load byte) instructions correctly: a LB instruction only loaded the first byte of whichever word

[1]This emulator can be found here: https://github.com/pdewacht/oberon-risc-emu/

[2]This RISC-V emulator can be found here: https://github.com/MicroCoreLabs/Projects/blob/master/RISCV_C_Version/C_Version/riscv.c

it addressed, rather than the actually addressed byte. Although fixing this is trivial, it did cause some unfortunate side effects during the port before it was discovered, where strings were not loaded properly.

4 Compiler

Porting the compiler was a large part of making Oberon work on RISC-V. In this section, I will go into detail on some of the changes necessary to make it work on RISC-V, and in particular, some of the challenges that arose because of the differences in design of RISC-5 and RISC-V.

4.1 Goal for the compiler

The ultimate goal for the compiler was to be able to compile the Oberon System for RISC-V. As such, the main consideration for any feature was whether or not the Oberon System needed it. While the Oberon System does need most features offered by the Oberon programming language, it does not make any use of neither real numbers (i.e. floating-point) nor interrupts. Even though some programs may want or need these features, none of them are necessary for a complete Oberon system. As such, they are not implemented.

4.2 Open-source

Around three weeks into the project, I discovered Sam Folvo's version of an Oberon compiler for RISC-V, also based on Wirth's RISC-5 compiler[3]. As this was open-source, and seemed to be a good foundation for porting the rest of the Oberon system, I moved over to using this compiler instead.

However, I discovered that large parts of the compiler did not function quite right, such that many of these edge cases had to be fixed for the full Oberon System to be able to be compiled. This is not too surprising, as a compiler and an operating system are some of the most complex programs one may want to compile. However, this ended up taking quite a lot more time than expected, as bugs in a compiler can be very difficult to track down. Seeing as the compiler is a major part of the process of porting the Oberon System in particular, I will focus on some of the issues faced in making this compiler fully functional.

4.3 Large conditional jumps

A problem particularly influenced by the difference in ISA occurs in large conditional jumps. An example would be an if condition whose body, the code executed if the

[3]This version can be found here: https://github.com/sam-falvo/project-norebo/tree/master/OberonRV

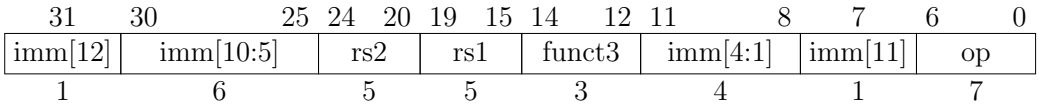
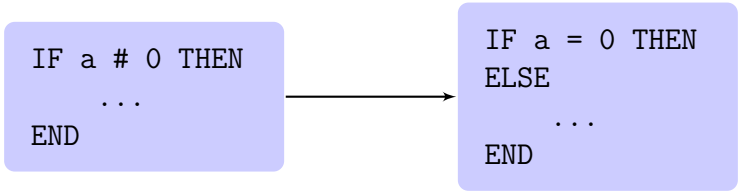


Figure 3: The B-type instruction format in RISC-V.

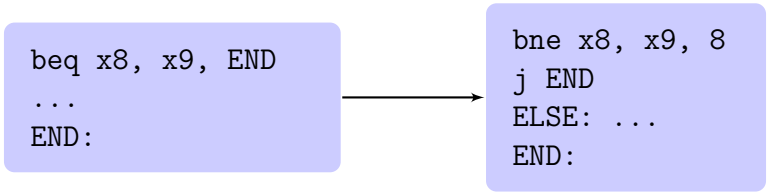
condition is true, spans very many instructions. An implementation of conditional jumps for a single-pass compiler, and the one used by the Oberon compiler, works as follows: place a link for a forward branch; compile the body of the conditional; finally, fixup the previously placed forward branch such that it branches to the end of the conditional's body. (For a more detailed explanation of fixups, see section 5.3, which discusses fixups necessary to link and load modules; this is in principle the same procedure, done during compilation.)

A problem arises when the number of instructions one needs to jump over exceeds the amount of bits allotted to memory displacement in a branch instruction. However, the branch instruction in Wirth's RISC-5 architecture allots 20 bits to memory displacement following a branch, meaning the body of the conditional would have to exceed 524288 instructions before this became an issue - certainly not something that occurs in the Oberon operating system. The roughly equivalent instructions in RISC-V are the branch instructions, with the B-type instruction format, shown in figure 3. Of particular note, then, is the fact that the immediate of the B-type instruction format only allots 12 bits to branch displacement - meaning, due to two's complement, the largest conditional forward jump possible is 2048 instructions, a much lower magnitude. When *Modules* exceeded this number, with an if condition whose body spanned 7512 instructions, this led to behaviour that was difficult to pin down. The resulting instruction's immediate was computed as 7512 mod 4096, which results in a two's complement encoding of -680. In sum, this meant the instruction `bne x8, x9, 7512` was instead encoded as `bne x8, x9, -680`.

The most basic solution for this that was implemented was to, upon fixup, check whether the branch was too large to be possible to branch over, and fire an assertion if this was the case. While certainly not an ideal solution, this allowed the programmer to rewrite offending programs. This rewrite was also quite simple, as one only needed to change a conditional as follows:

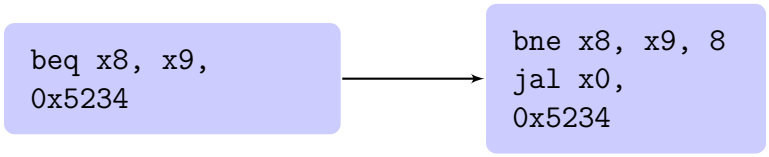


The end of the body of an IF condition will always contain an unconditional jump over the instructions encompassed by the ELSE condition; importantly, unconditional jumps in RISC-V have more bits for offset, allowing larger jumps. Thus, this produces the following instructions:



However, this is hardly a good solution. For one thing, it makes it much more cumbersome to read the program, especially if the programmer also has an ELSE case to take into account. Furthermore, it's something any programmer will assume the compiler ought to handle properly.

A large problem with implementing a better solution, however, was the single-pass nature of the compiler. In a multi-pass compiler, one could for instance replace the offending branch instruction at the head of the if condition with a branch and a jump that can reach farther, like so:



However, with a single-pass compiler, this becomes inordinately clunkier. As it requires two instructions rather than one, it becomes necessary to shift all the code already generated forward to make room for the jump instruction. Furthermore, the list of instructions pending fixups placed within the block that are dependent on the linker – e.g. fixups for retrieving variables from imported modules – must also be rewritten to account for the shift. This was deemed a poor solution, as it is optimising much too heavily for what is a fairly rare case. Alternatively, one could always generate two instructions for fixup, and generate one into a *nop* if it isn't necessary. However, this has a negative impact on performance and code size to, again, optimise for a rare case.

The solution that was eventually put in use is slightly less efficient, but more in-line with what can be easily done in a single-pass compiler. As this is a rare case, using slightly more instructions than necessary in return for much lower complexity in the compiler is a worthwhile trade-off, given some of the principles set out in Section 2.4. Essentially, a few more instructions are appended to the end of the body of the already generated if instruction, that perform the actual branching part; this transformation can be seen in figure 4. This way, the jumps over large portions of code are always performed by jump instructions, which allow for larger offsets.

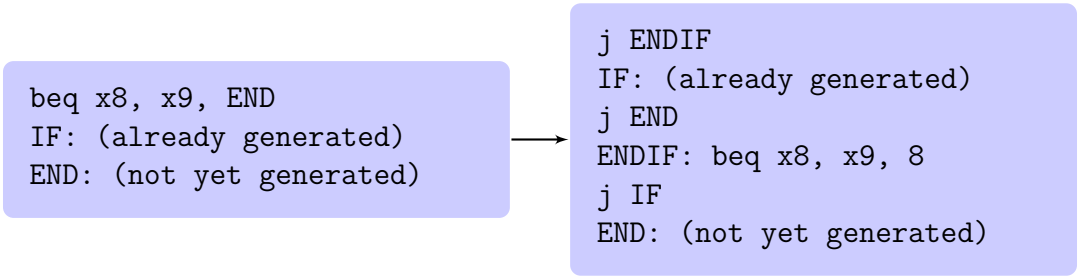


Figure 4: Transformation performed if the instructions generated between IF and END are too large for `beq` to branch over.

4.4 Comparisons

Another stark difference between the RISC-5 and the RISC-V ISAs are in their use of flags. RISC-5 uses *condition flags*, similar to e.g. ARM, wherein a side effect of an arithmetic operation is that it deposits condition codes based on the result in an auxiliary register. For instance, in RISC-5, `sub r0, r1, 2` will deposit a high bit in the auxiliary register N if the result of the subtraction is negative, and similarly, a high bit in the auxiliary register Z if the result is zero. This allows for e.g. determining equality of the two registers. By contrast, RISC-V does not use such condition codes, as by adding extra state they "needlessly complicate the dependence calculation for out-of-order execution"[2] – that is, an instruction can be implicitly dependent on a prior instruction simply because of the condition codes it set. Instead, RISC-V uses branch instructions where the comparison between two registers and the branch from the result of the comparison is done in a single instruction. A quick example of the difference in how the ISAs prescribe branching (note that `cmp` is merely an alias for `sub`):

RISC-5

```
cmp r0, r1
beq 0x5234
```

RISC-V

```
beq x8, x9, 0x5234
```

This has implications for compiler design. In the original RISC-5 compiler, whenever it encountered a comparison, it could immediately perform the comparison, and expect the resulting condition codes to be used by e.g. an upcoming branch. As *performing the comparison* in RISC-5 is simply subtracting the two numbers to be compared, this meant that all relations could be treated similarly, without having to take the actual comparison in question into regard. Regardless of if the comparison is checking for equality or less-than, the subtraction will be the same. This design is well-suited for a simple, single-pass compiler, as a comparison introduces no additional state from the perspective of the compiler. However, as should be clear from the above discussion, this scheme will not work in RISC-V.

The solution for RISC-V, as was already developed by Sam Folvo and Peter de Wachter and in use in the open-source Oberon compiler I continued work on (cf.

section 4.2), was to introduce additional state to track which registers to compare in a branch instruction. With the two separate stages of conditional branching – performing the comparison; branching – combined, the state in each separate stage must be known altogether to generate the correct instruction. Two new fields are added to the *Item* object, which carries the state of syntactic objects[8], to track the registers a conditional is comparing; the object was already holding information of what comparison to use.

However, the increased state to track introduces additional complexity, and so is more error-prone. For instance, intermediary calculations affect the register stack, used to allocate available registers; if one is not careful, these can affect the values stored in registers necessary for a later branch instruction. This can cause bugs that are both subtle and potentially system-breaking! A quick example to illustrate this follows.

Oberon supplies a SET data type, representing a 32-bit word upon which operators effectively become bitwise operators; e.g. plus represents OR, multiplication represents AND, etc. Oberon also allows testing for membership of a number in a SET, e.g. `3 IN SectorMap`, which will be true if the fourth bit in the SET is high. Less subtly than checking the relation between two integers, this is still a comparison; in this case, performed by moving the bit of interest (e.g. 3) into the sign bit of the word, and then checking whether this is less than zero. If the value at the location of interest is 1, the word will be negative; if not, it will be positive.

However, an issue that arose from this happened in the case where membership of *the result of an expression* in a SET was tested. Membership of constants and variables worked, but as the result of the expression was located elsewhere on the register stack than the comparison expected, the branch ended up comparing the wrong registers. With luck, this bug would merely be unfortunate; but as the Oberon System used exactly such a comparison to allocate sectors in storage, the file system ended up becoming corrupted when attempting to store a buffer on disk, as it failed to allocate an empty sector, instead allocating an already filled one.

The solution, as opposed to the bug, was fairly simple, and merely required explicitly telling the compiler which register to compare with, rather than relying on its position in the register stack. However, tracking down the bug itself was far more difficult, as such an effect from a failed comparison is not immediately obvious.

4.5 Data section and global variables

This issue is similar to the one discussed in section 4.3, but with a less ideal solution space.

Each module has a separate segment for storing global variables, similar to data segments in ELF binaries. To resolve access of variables in modules' data segments, the linker, discussed in more detail in section 5.3, is needed to resolve such accesses. For access of global variables within the module wherein they're accessed, this is quite simple to resolve: all the linker has to do is fixup an instruction loading the

base of the module's data section into an already specified register already set aside for this purpose.[4] A crucial aspect is that for global variables *within the compiling module*, the offset into the data section for the desired variable is known in compile-time. Therefore, loading a variable with an offset larger than the space possible to reach with a load instruction[5] simply requires loading the offset into a register first. This requires an additional instruction, but this is only generated when necessary.

However, for accessing global variables in other modules' data segments, this solution is less feasible, and a similar decision to the one in section 4.3 has to be made. The issue is simply that the offset to load a global variable from a different module is not known at compile-time, so the compiler cannot generate the instructions required if the offset is larger than the immediate space in a load instruction.[6] However, as mentioned, the solution space here is much smaller: more code *cannot* be generated for the program without increasing the complexity by an order of magnitude, as this would have to be done by the linker, not the compiler. This means, to fully solve the problem, the compiler would have to generate two instructions for every access to an external variable, even if this isn't strictly necessary. The linker could then fixup both of these instructions as required. However, there are several things to consider with this solution:

- It is quite inefficient for most programs, as few modules' global variable space exceeds what offsets can be encoded, worsening performance and size for a rare case.
- This increases the linker's complexity even more, as it now upon any reference to an external module has to fixup three instructions, one to load the base of the module and two to load the variable. As discussed in more detail in section 7, the linker is already a fair share more complex in RISC-V, so this was not desirable.
- The original Oberon System faces a similar issue, although less so, as its offset allows for 64kB of global variables, as opposed to 2kB. However, this difference in size is less stark than it might at first seem: many of the programs that run into issues due to the 2kB limitation on RISC-V can easily run into the same in RISC-5. For example, the array holding all generated instructions in the compiler's code generator, *ORG*, is unable to compile programs larger than 64kB due to this limitation.

An alternative solution could be to put information about offsets in symbol tables, which can be read during compile-time. However, the offsets are likely to change if any part of the program changes, even if no new variables or procedures are exported; this would require creating a new symbol table, and hence recompiling all

[4]Here, this purpose is reserved for x3/gp, often given the purpose of being a global pointer in RISC-V convention.

[5]Currently $2kB$, but easily made $\pm 2kB$ if offset from 2kB into the data segment

[6]For RISC-5, this is not a problem: as the instruction set was designed specifically for the Oberon system (as mentioned in section 2.3), they were able to allot as many bits as required for immediates in their own architecture.

dependent modules. This harms the extensibility of the system, and as such is also a poor solution.

These factors led me to decide against a "complete" solution to this problem, as opposed to the one in section 4.3. Instead, the compiler warns the programmer that external access to a module's data segment may lead to undefined behaviour if it exceeds 2kB. If the programmer requires more than can reasonably fit within the global data space, this can easily be allocated from heap space instead. While only a "workaround", this is beneficial for the Oberon System as a whole: modules are more likely to be able to be loaded; checking whether a memory allocation failed is already standard practice[7]; and it more easily allows for large blocks of data without going over the limit of what can be stored in global variables, including the already extant limit of 64kB in the original Oberon system. Furthermore, it can lead to more efficient use of memory: the garbage collector can free memory in the heap, whereas the data segment is fixed to be as large as need be, occupying space until the user unloads the module.

There are some downsides to this alternative, particularly for the compiler, which was used as a case study for this solution. The compiler allocates memory in the heap upon compilation of a new module, and removes those references when it finishes compilation of the module so the garbage collector can free memory. However, the garbage collector only runs every second; this means that, if the compiler is called again immediately, it will have less memory to work with, as the memory it last allocated to use has not been freed. To circumvent this, one could either reuse allocated memory, implying it must remain allocated for as long as the module is loaded, losing some efficiency of use of memory; force the garbage collector to run between compilation units; or put some span of time between each compilation. The last is currently in use, but the other two can easily be used too.

4.6 State of the compiler

The compiler can compile most programs from within the Oberon System, although not all. The MagicSquares program, which attempts to generate a magic square of the dimensions specified, compiles and runs, but generates invalid magic squares.

There are also a few known imperfections left. There was a bug that led to incorrect reading of symbol tables in compile-time from within the Oberon system. This led to failed compilation on any program that makes a procedure call to an external module, if that procedure expects a CHAR array (i.e. a string) as a parameter, as it read the expected length of the parameter incorrectly; where it should have read `0xFFFFFFFF`, the compiler read `0x7F`. This is likely a result of the compiler only reading a byte where it should read a word, as the value written into the symbol table is correct. Although this bug may sound as if it triggers rarely, it affects any program that attempts to print a string, meaning many programs fail to compile.

One can work around this bug either by compiling all desired modules from outside

[7] Although the Oberon system fails to check if it got a pointer into the heap or merely NIL most of the time, which of course can lead to errors – I have remedied this in my port

the Oberon System using the cross-compiler, which compiles the programs correctly, *or* by adding an edge case to the parser, to make it accept parameters of size 0x7F as equivalent to size `0xFFFFFFFF`. No bugs as a result of this workaround have been observed, and this allows programs to compile successfully, although it leaves the underlying error in the compiler unfixed.

4.6.1 Traps

Another improvement that would be desirable is to make traps more optimal. Currently, traps are performed very similarly to in RISC-5, where a trap cause is deposited in unused bits of an instruction; this is an example of the ISA being designed for the operating system, as mentioned in section 2.3.3. However, because such unused bits do not exist in RISC-V, this is circumvented as follows:

```
jalr ra, mt, 0
jal x0, 0x8
[trap number and position]
```

The kernel, which handles traps, must then look in the instruction after where RA points to find out which trap was triggered and where. The `jal` instruction is necessary to avoid executing the trap number as an instruction, which would fail.

This is not an ideal solution, as it is quite cumbersome and takes several instructions to do what RISC-V *can* support in one. However, as mentioned in section 2.1, the decision to only use machine-mode was made. This becomes a problem for a hardware solution, as RISC-V only supports vertical traps, i.e. traps that increase privilege level. To perform a horizontal trap – a trap within the same privilege level – one must perform a vertical trap, and then return control to a trap handler in the lower privilege level[4]. This is impossible without multiple modes, which would complicate many other parts of the operating system. As such, the current solution is kept.

5 Porting Oberon to RISC-V

5.1 Bootloader

Porting the bootloader of Oberon to RISC-V, or indeed to any platform in which a compiler is already implemented, is surprisingly easy. This is due to two factors: the rather simple assumptions the Oberon system makes about hardware, and compiler support for writing bare-metal software.

Firstly, as the Oberon system does not rely on neither interrupts nor an MMU, configuring these in startup code is not necessary. Furthermore, the Oberon system effectively assumes the entire operating system is run in machine-mode, so entering

user mode is at no point necessary either. In other words, no aspect of booting the Oberon system necessitates writing assembly code.

Secondly, the Oberon compiler has specific syntax for compiling to bare-metal code. In other words, the compiler can compile a program written in Oberon, and so long as the program does not rely on any part of the Oberon system or make use of abstractions that necessitate an operating system, such as heap memory allocation, it will compile into machine code that can be run without an operating system. Additionally, the compiler places a jump to memory location 0x0 in the end of execution of bare-metal code, meaning the build of the operating system in storage being loaded must have a jump into its entry point there.

The result of these two factors is that the bootloader for the Oberon system can be written *entirely* in Oberon. No handwritten assembly code is necessary. The bootloader consists in large part of a driver that can read from an SD card, as its only purpose is to load the necessary modules from storage into main memory such that the Oberon system can boot.

To be certain, some modification of the bootloader is still necessary: for instance, it needs to set the value of the stack pointer, and the register of the stack pointer will differ depending on the instruction set architecture. Furthermore, if the system does not boot from an SD card, a new driver will have to be written to support the desired storage system. However, in this project, the assumption is a similar system aside from the change in ISA, and as such the driver itself remained untouched.

5.2 Kernel

The kernel in the Oberon system is responsible for a few things – though not nearly as many as a monolithic kernel would. It serves four purposes: it organizes memory, taking care of memory allocation in the heap; it contains a device driver for the SD card; it has a trap procedure, for use before the outer core of the Oberon system has been loaded; and finally, it contains some miscellaneous procedures for tracking time. Neither the device driver nor functions for returning time need elaboration, as they are unchanged between RISC-5 and RISC-V. However, the trap procedure must be treated somewhat differently, as the Kernel is responsible for placing a jump instruction into the trap procedure where the compiler expects it to be. In essence, the compiler places a jump into a trap vector table, and the kernel is responsible for placing a jump to the trap procedure into that vector table.

5.3 Linker and loader

A more complex part of the porting process was fixing the linker and loader in the Oberon system. These are part of the *Modules* module.

It must be stressed that this is a separate step from porting the compiler. Upon encountering any Oberon code that relies on external references - e.g. calling a function that resides in another module - the compiler, rather than generating working

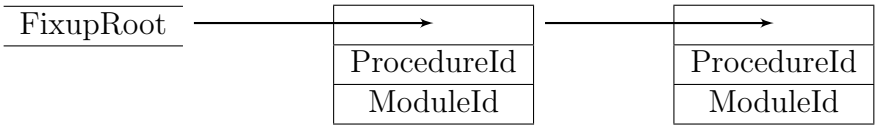


Figure 5: The linked list traversed by the linker to perform necessary fixups. The compiler constructs the list, and the linker changes every instruction within it into a valid RISC-V instruction upon traversal.

RISC-V code, leaves a word indicating what object it is referencing, as well as where the next instruction in need of linking in the module is. This process is known as a *fixup*, and the linked list of instructions formed within the program is a *fixup list*. The header of the program stores a reference to the first element of each fixup list, which the linker can then traverse, changing every instruction found in the list to match what the programmer intended; this mechanism is shown in figure 5. In the process of doing this, external references go from using addresses relative to the base of the module, meant for interpretation by the linker, to absolute addresses that work in the runtime system the program is meant for.

This job can be done by a *static linker*, which would package all the dependencies of the program into the final binary, creating a standalone binary. This is an older technique, although it still has its uses today to create binaries without assumptions about what libraries exist on the target system. The Oberon system instead uses a dynamic linker, i.e. a linker that links program dependencies in runtime. There are several reasons for this choice[8]:

- It avoids recompilation of a module if one of its dependencies has changed.
- Improved extensibility: new modules can more easily depend on modules that have already been loaded.
- Space: statically linked binaries are often much larger than dynamically linked binaries, which becomes a greater concern on a system with 1MB of RAM.

As mentioned, the *Modules* module takes care of this, and is in fact the first module entered from the bootloader. What changes were necessary to make this work in RISC-V, then?

Large parts of the linker and loader could remain untouched. Traversal of the lists, for instance, worked the same both in the old system as well as in the RISC-V port. The change necessary was largely in formatting the fixups such that they would become valid RISC-V instructions. Of particular importance here is the fact that encoding RISC-V instructions in software is more difficult than RISC-5 instructions, as discussed in 2.3.1.

```
SYSTEM.PUT(adr, (offset MOD 1000000H) + 0F7000000H);
```

Rewritten for RISC-V

```
(* Returns a `jal ra, imm` instruction. *)
PROCEDURE Jal(imm: INTEGER) : INTEGER;
  VAR imm20, imm19to12, imm11, imm10to1: INTEGER;
BEGIN
  imm20 := imm DIV 100000H;
  imm19to12 := (imm - imm20 * 100000H) DIV 1000H;
  imm11 := (imm - (imm20 * 100000H + imm19to12 * 1000H)) DIV 800H;
  imm10to1 := (imm - (imm20 * 100000H + imm19to12 * 1000H + imm11 *
    800H)) DIV 2H;
  RETURN ((((imm20 * 400H + imm10to1) * 2H + imm11) * 100H +
    imm19to12) * 20H + 1) * 80H + 111
END Jal;
(* ... fixup code here ... *)
SYSTEM.PUT(adr, Jal(offset MOD 200000H));
```

Figure 6: RISC-V increases complexity of performing fixups. Shown here, an equivalent fixup of calling functions in external modules, in RISC-5 and RISC-V.

6 Building the Oberon system

In this section, I will go into some detail on how to build the Oberon system for RISC-V. Currently, this has only been done targeting an emulator, but there is nothing suggesting it should be very different in hardware, aside from changing device drivers to target different I/O ports.

6.1 Project Norebo

The build system used throughout this project was based on Peter de Wachter's Project Norebo[8], a free and open-source project to run programs intended for the Oberon system on POSIX systems. It does this through emulation, using the RISC-5 emulator discussed in section 3.1, but redirecting text and file output to the POSIX system it runs on rather than in the Oberon system. In addition, it offers a Python script to build images, and the images built by it can also be run by the RISC-5 emulator. The images it generates can also be installed on SD cards, to be used on an FPGA running an appropriate core; this has not been tested yet, but should in theory work. This build system allows for building a complete Oberon system from a GNU/Linux system, which is much faster than building it from an emulated full

[8]The source code for the original Project Norebo can be found here: https://github.com/pdewacht/project-norebo

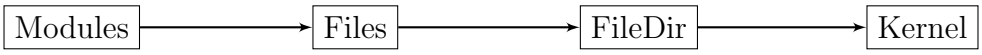


Figure 7: The *inner core* of Oberon, which must be linked beforehand. Arrows show dependencies, such that Modules depends on Files, etc.

Oberon system would be.

To make Norebo build a RISC-V system instead of a RISC-5 system, a few steps were necessary. Most obviously, it had to use the RISC-V compiler, discussed in section 4, as a cross-compiler. This means the compiler was built to target a *RISC-5* system, such that it could run in Norebo's emulation layer, and compile the rest of the files desired to RISC-V, to be run in a RISC-V emulator or system. This change was rather simple, and only required adding another set of functions for RISC-V building, pointing it to the RISC-V compiler rather than use the RISC-5 compiler.

The linker used by Norebo, a module named *CoreLinker*, also required changes to work with RISC-V. To allow the build system to create images for both RISC-V and RISC-5, a module named *RVCoreLinker* (where the RV signifies RISC-V) was created, to link RISC-V programs. One might be tempted to ask why a linker should be necessary here, since the Oberon system dynamically links its modules in runtime. This is on the whole true, but the *Modules* module, responsible for dynamically linking and loading programs, discussed in more detail in 5.3, has a dependency on the *Files* module, to retrieve the files it loads. As it cannot link itself in runtime, this particular module must be linked by the build system. As this is at the top of the chain of dependencies of Oberon's *inner core*[8], of which the full chain is shown in figure 7, this entire part of the Oberon system must be prelinked in the build stage. The bootloader is responsible for placing the linked inner core into memory contiguously, such that the already linked memory references are correct.

Norebo was extended as Oberon was ported, according to the needs demanded by the project. Functionality to generate images that only loaded a particular module to be tested proved to be worthwhile. Similarly, flags to define whether to build a RISC-V or RISC-5 image, as well as a flag to use a different manifest (list of files to build), were useful to quickly build different versions of the operating system. Norebo was also extended to, after installation, verify whether files were correctly installed to the image, which proved itself useful when e.g. accidentally using the wrong version of a manifest.

The end result of making Project Norebo work for RISC-V is that, as mentioned, building RISC-V images becomes faster and easier; all the commands necessary to create a new build are:

```
rm -rf imagebuild; ./build-image.py -r OberonRV/Oberon
```

A disk image along with all compiled programs and symbol tables are deposited in the `imagebuild` folder.

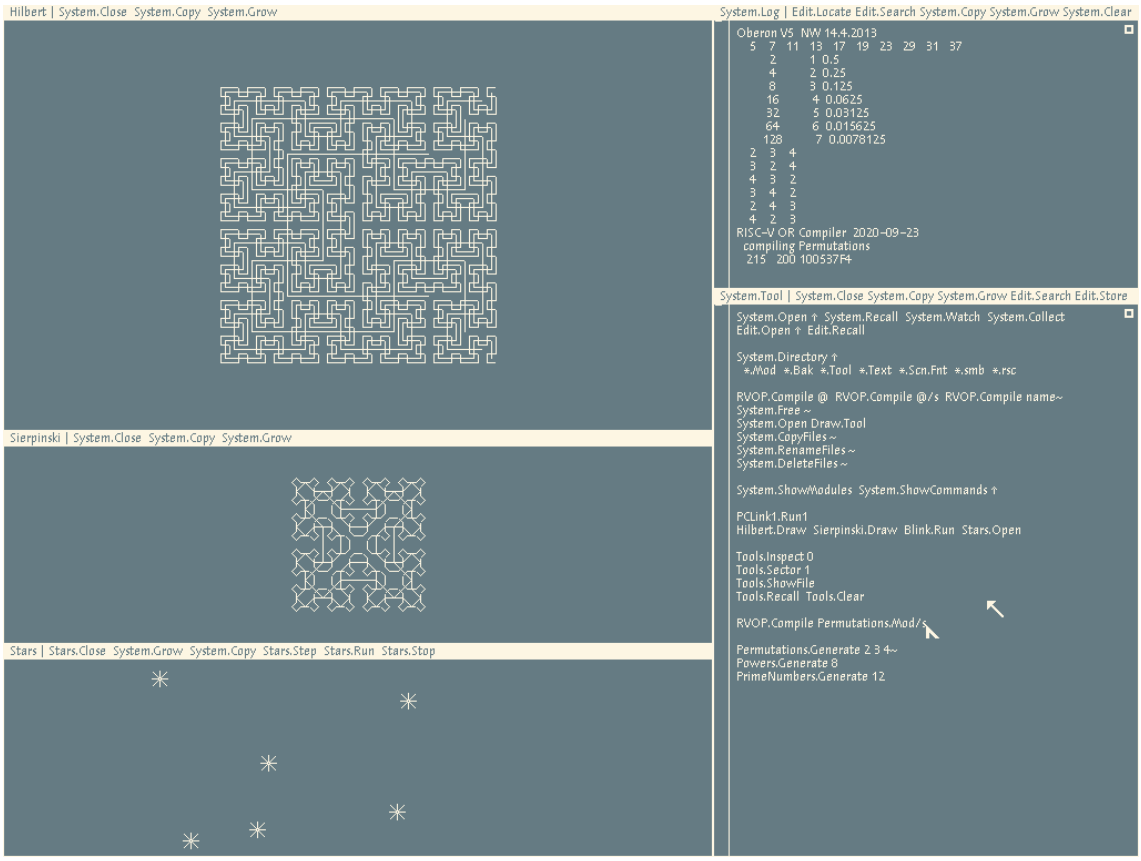


Figure 8: The Oberon system, running in a RISC-V emulator.

7 Results and Discussion

Some interesting results can be gathered from the RISC-V port, when compared to the original. These can give some insight into benefits of RISC-V over the previous ISA used by the Oberon system, RISC-5.

7.1 Working Oberon system

The major result of the project is a version of the Oberon system that runs on RISC-V, and is close to fully functional, excepting bugs discussed in 4.6. This can be seen in figure 8. It can run most programs without issues, and supports the same set of features as the RISC-5 version.

7.2 Quantitative results

As mentioned in section 3.1, the emulator's main focus is speed and ease of use, not accuracy. As such, the data that can be gathered is rather limited, and would require either a different emulator or a hardware platform. However, some data can be gathered with the tools at hand, mostly some notes on speed and size, which will be examined in more detail below.

To gather information on traces of specific parts of programs, "breakpoints" were left at the beginning and end of the area of interest. This is done in the form of writing specific magic numbers to the I/O port that writes all input to LEDs, which can then be identified in logs from the emulator, so that only instructions run between the beginning and end of the area of interest are included.

7.2.1 Instructions run

A metric to consider is the amount of instructions it takes for the system to boot. *Booting* is here considered not to be just the time spent in the bootloader, but as the time spent from turning on the system and entering the bootloader, to entering the main loop in the *Oberon* module. Table 1 shows the difference in instructions performed to do this. As one might expect going from one RISC architecture to another, the difference is not too large, but still notable, with RISC-V taking 8.63% more instructions to fully boot than RISC-5.

Number of instructions run	RISC-5	RISC-V	Increase in RISC-V %
boot	7664855	8326033	8.63%
boot (only linker)	450092	561861	24.83%
boot (only linker and files)	4334435	5146595	18.74%
Hilbert.Draw (with linker and files)	1881046	1831383	-2.64%
Hilbert.Draw (without linker and files)	1610140	1534963	-4.67%
PrimeNumbers.Generate 12 (with linker and files)	84006	92634	10.27%
PrimeNumbers.Generate 12 (without linker and files)	10150	9696	-4.47%

Table 1: Comparison of the amount of instructions it takes to boot the Oberon system on RISC-5 and RISC-V respectively. In addition, some standalone programs are included.

The entire process of booting the Oberon system is shown in figure 9, which shows where the processor is executing instructions over time. The bootloader is excluded from this figure, as it resides in the memory region of $0xFFFFF800 - -0xFFFFFFFF$, and as such would vastly reduce the legibility of the figure. From this graph, one can ascertain which parts are run for the longest period of time during booting: particularly notable is the block that takes up most of the execution time, from time $1.33e6$ to $7.49e6$. This period of time is almost entirely occupied by the linker and loader, as well as calls to the file system. The "spikes" into higher memory regions seen in this period are due to initialization bodies of loaded modules being called, which are noticeably short.

As a very large part of booting is linking and loading the rest of the operating system, the increased complexity of linking seemed like a good reason for the increased time spent in RISC-V. To confirm, a trace taken of *just* the time spent linking was taken.

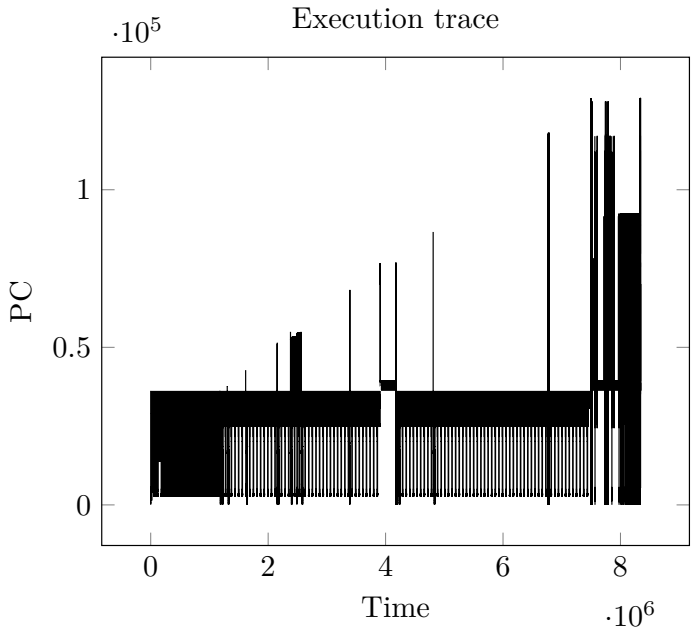


Figure 9: Execution trace of the Oberon system, taken from after the bootloader transferred execution to the operating system to when it finally enters the scheduler loop for the first time. Each timestep is the execution of a single instruction.

As can be seen from the row *boot (only linker)* in table 1, loading and linking the outer core almost takes 25% more instructions to do in RISC-V. The reason for this is likely largely that the linker is more complex for RISC-V, as already discussed in section 5.3. However, while responsible for part of the increased boot time, it is not responsible for all of it. If one includes calls to the file system into this as well, included in row *boot (only linker and files)*, one gets a more complete picture of where the increase in time comes from. Note that this does not include the loading of the files themselves, for which the Kernel is responsible, as that is part of the driver to the SD card. The file system has not seen any significant revision to work in RISC-V, but is rather complex, performing many operations that are less optimal on RISC-V. However, note that the difference in boot time is *less* than the difference in linking, meaning the RISC-V port saves instructions elsewhere.

For more insight, data was gathered on standalone programs as well, namely Hilbert and PrimeNumbers, which respectively draw Hilbert curves and generate prime numbers. These are particularly interesting for this purpose as they can compile without modification for both RISC-5 and RISC-V, due to their lack of low-level system calls, while still interacting with the operating system. This means they are more representative of speed-ups or slow-downs that are not caused by modifications to the program, but rather to the compiler. In addition, to get a fuller picture of the cost of the linker's complexity as well as potentially increased operating system overhead, time spent from when the programs start linking is also included in table 1.

Evidently, both Hilbert and PrimeNumbers[9] spend *less* time executing the core of

[9]Both of these programs can be found in Wirth's Project Oberon repository: https://people.inf.ethz.ch/wirth/ProjectOberon/. Note that PrimeNumbers is found in `SmallPrograms.Mod`.

the program, but *more* time in linking, loading, and operating system overhead.

What, then, causes the programs themselves to spend less instructions running to completion? The compiler does not particularly optimise for using the zero-register (although it quite easily could). This result is somewhat surprising, as it is not immediately obvious what causes the RISC-V program to require running less instructions. Upon closer inspection, the difference is mostly due to the fact that branching in RISC-5 takes two instructions – one to compare, one to branch – while in RISC-V it only takes one, as discussed in closer detail in section 4.4. While other programs may take more instructions in RISC-V, these have many small loops and branches, which favours RISC-V. This also seems to be where RISC-V saves instructions elsewhere in the booting process: many parts of the rest of the booting process, e.g. the Kernel, consist of many such small loops.

7.2.2 Binary size

As the Oberon system should function with limited memory – with the latest revision being created for an FPGA with 1MB memory – keeping modules small is rather important. A comparison of the size of programs compiled for RISC-5 and RISC-V respectively are found in table 2.

Program	RISC-5	RISC-V	Increase in RISC-V %
Hilbert	2397B	2873B	19.86%
PrimeNumbers	995B	1107B	11.26%
Modules	5675B	6587B	16.07%
RVOG	33241B	35753B	7.56%

Table 2: Sizes of different binaries compiled with the RISC-5 and RISC-V compilers respectively. For Modules, the linker for RISC-V was compiled in both cases; i.e., they compiled the same program.

Binaries compiled for RISC-V are generally larger, though not prohibitively so. This is for several reasons: instructions performing on immediates often have to spend multiple instructions rather than one, if the immediate is large enough that it has to be loaded into a register to be operated on; traps require three instructions rather than one, as discussed in section 4.6; etc. Note that despite both Hilbert and PrimeNumbers being larger binaries, they still perform better than in RISC-5 once invoked, as the program does not spend most of its time in code that has expanded to more instructions from RISC-5 to RISC-V.

7.3 Future work

There are still things that can be done for this project, although the main focus was accomplished. For one, the Oberon system has not been tested on hardware, although this should be feasible to accomplish without too much trouble. Additionally, there are still some bugs and inefficiencies left in the compiler, as discussed in

section 4.6, although they are minor enough not to have any impact on the operating system or the compiler. Finally, to fully support all software developed for Oberon, interrupts and REALs need to be implemented, although they are not for the moment. Both should be feasible; REALs especially if one targets hardware that supports floating-point operations. If not, a good solution would be to make the compiler generate system calls to the operating system for floating point emulation.

In addition, the Oberon system has the potential to run on more limited hardware than it is currently targeting. The execution trace of Oberon shows large parts of loaded code being run once, and never again; this can be resolved more optimally by using overlays to transfer these blocks of code out of main memory when they are no longer needed, and load them back in if they are needed once more. This could improve the Oberon system's capability to run on smaller systems.

8 Conclusion

In this project, the Oberon system – meaning the compiler as well as the rest of the operating system – was ported to run on RV32IM hardware, using only machine-mode. As a result of this effort, an emulator that provides an easy way to test the Oberon system was extended to support RISC-V; and a build system, Project Norebo, was also extended to create RISC-V images.

In the process of porting the system, the compiler in particular had to go through some extensive changes to work properly. This was in part due to RISC-V being somewhat more complex to work with in software, due to more complicated instruction encoding; and in part due to some fundamental differences between the two ISAs, as one uses condition codes while the other compares registers upon branch.

Another aspect of RISC-5 that became clear in the process, and made the porting process to RISC-V more complicated, is the fact that it was designed explicitly for the Oberon system. For instance, traps in Oberon only take one instruction to jump to the trap vector table, with the instruction also encoding both the character of the program wherein it trapped, as well as the cause of the trap. This is because the jump instruction in RISC-5 includes several unused bits in the word, specifically to make this so efficient. RISC-V, designed to be as general-purpose as possible, naturally lacks unused bits for this purpose.

However, despite RISC-5 being specifically designed for the Oberon system, the results are still promising. RISC-V does not take much longer to boot, and for some programs, it actually requires less instructions to run. As booting only takes 5% more instructions in RISC-V than RISC-5, the difference would likely be imperceptible.

The port has also put some of Oberon's strengths further into view: for instance, while the compiler required much work to create RISC-V programs rather than RISC-5, the rest of the operating system was quite simple, requiring relatively few changes. Much of this comes down to the simple assumptions the Oberon system makes about its hardware: it requires no memory protection, its traps do not rely

on hardware support, and neither floating-point nor interrupts are required for the system to run.

In conclusion, while RISC-5 does offer some helpful simplicity for the Oberon system, it is far from necessary for the system to run well. The differences between RISC-5 and RISC-V offer some challenges in the effort of porting from one to the other, but none of them are insurmountable. The results in terms of performance in RISC-V are also of comparable order of magnitude.

Bibliography

[1] Michael Franz. 'Oberon - The Overlooked Jewel'. In: *The School of Niklaus Wirth*. 2000, pp. 41–54.

[2] David Patterson and Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017.

[3] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. 2019.

[4] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. 2019.

[5] Niklaus Wirth. 'A Plea for Lean Software'. In: *Computer* 28.2 (1995), pp. 64–68.

[6] Niklaus Wirth. 'Modula-2 and Oberon'. In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. 2007, pp. 1–10.

[7] Niklaus Wirth. *The RISC Architecture*. 2018.

[8] Niklaus Wirth and Jürg Gutknecht. *Project Oberon*. Addison-Wesley Reading, 2013.