Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Using a Modern Language for Bare Metal Programming on RISC-V

by

Yannis Huber

Supervised by

Daniel Gachet
and
Jacques Supcik

May 8, 2020
v1.0.0

Department of Computer Science

School of Engineering and Architecture of Fribourg

Revision History

Version	Description
`v0.1.0`	First draft for review
`v0.2.0`	Second draft for review
`v1.0.0`	Final draft

Abstract

In this project we explore the use of a modern programming language and compiler to produce bare metal programs targeted at RISC-V based embedded systems. To illustrate this, we have made contributions to an existing Go compiler called TinyGo. The goal is to extend the compiler's support for the HiFive1 Rev B RISC-V based board. Features such as I^2C and SPI have been developed or extended as a way to get familiar with the compiler and its associated runtime library. This document also serves as an introduction to the RISC-V instruction set architecture and discusses some interesting aspects of compiler design.

Keywords: RISC-V, compiler, Go, embedded systems, IoT

Contents

1 Introduction

In order to perform arithmetic tasks, early computer systems of the 1950s had to be programmed physically using a determined mechanism. The most common way of programming such a computer was by using punched cards as digital data input. Since each computer design was unique, every piece of code had to be written specifically for the targeted computer and could not be reused on another machine. The incompatibility of the different computers was a huge concern for companies, which knew their customers had little reasons to stay loyal, since the next machine would be incompatible anyway. It is only in the 1960s that IBM developed a reference instruction set architecture (ISA) which allowed the same software to run on multiple machines, each having different performances. An ISA is an abstraction model of a computer and acts as an interface between the hardware and the software. This has been a major breakthrough in computer design and has been followed by almost every other manufacturer. The reference ISA developed by IBM has evolved and is now called the complex instruction set computer (CISC) [5].

Ever since, there have been many new ISAs developed, which can be mostly grouped in two main categories. The first one is the already presented CISC type, and the second one, the reduced instruction set computer (RISC) type. The main difference between these two being the number of CPU cycles it takes to execute an instruction. In fact, a CISC type architecture, has a lot of specialized instructions which may be rarely used in some programs. A RISC type architecture, however, implements fewer but very efficient instructions. There are other RISC architecture specifics, such as fixed-length instructions or load-store memory access which will be further discussed later in this report [6].

The focus of this project is to discover a new ISA, the RISC-V instruction set. The RISC-V instruction set has been originally developed at the Parallel Computing Laboratory at UC Berkeley, California, and is now under the governance of the RISC-V foundation. It is released under an open-source license and therefore can be used by anyone in all types of implementations. As its name implies, the RISC-V instruction set is part of the RISC family, which, due to its fixed-length instructions and simplicity, makes it a perfect candidate for embedded systems. This project aims to contribute to a modern framework which can be used for bare metal programming on RISC-V based boards. The advantage of using a modern and high-level programming language over a more traditional language like C is that the learning curve is less steep. There is also less room for errors induced by the programmer. However the drawback is that it is more difficult to implement a compiler for such a framework because there is a very high level of abstraction between the programmer and the actual hardware. Of course such a framework will never allow the flexibility and performance offered by the C language — or even assembly — and a "real" bare metal program. But for non-critical applications, it can be interesting to have a more accessible approach.

There are multiple high-level languages which have been adapted to run on microcontrollers, the most famous example would be MicroPython[1] which is an adaptation of the Python interpreter that is

[1] https://micropython.org/, visited on 05/06/2020.

capable to run on only 256 Kibit of code space [14]. However, in this project we have chosen another compiler, based on the Go programming language called TinyGo[2]. While still in early development, TinyGo has already plenty of support for different microcontrollers and offers a good starting point to learn more about compilers and the new RISC-V ISA.

Concerning the RISC-V based board, we have chosen two different boards produced by different manufacturers, for comparison. They also have independent hardware specifications which are interesting to compare.

Before diving into the comparisons and choices made for this project, we are going to take a deeper look at the RISC-V ISA and see how it relates to other existing RISC-based architectures.

Structure of this document. The next chapter offers an introduction to the RISC-V instruction set architecture and its specificities. Chapter 3 describes some of the availables RISC-V based development boards and modern frameworks targeting embedded systems. Chapters 4 and 5 explain, by way of examples, the internals of the TinyGo compiler and its runtime. Finally, the last chapter will conclude this project and discuss possible future work.

[2] https://tinygo.org/, visited on 05/06/2020.

2 RISC-V Instruction Set Architecture

As already mentioned in Chapter 1, the RISC-V instruction set architecture has been originally developed at the Parallel Computing Laboratory at UC Berkeley, California, and has since been maintained by the RISC-V foundation. It is a common misconception to think that RISC-V is a CPU implementation while it is "only" an ISA specification. So while the specification is open-source, there can be implementations of the RISC-V specification under closed-source licenses.

RISC-V was designed with a simple fixed base ISA and modular fixed standard extensions to help prevent fragmentation while also supporting customization [9]. The advantage of decoupling the ISA, is that the parts that are now frozen — which means they will not be changed in the future — guarantees developers that the software developed today will run on RISC-V cores forever. Additions to the base ISA can still be made using either one of the standard extensions or a custom one. More details concerning the different extensions are in the next section.

2.1 Specification

Even though we often talk about *the* RISC-V ISA. RISC-V is actually a whole group of related ISAs. The only required base ISA defines integer support and has only a few instructions. Extensions can be added on top of the base integer ISA to add new features according to our needs. Each extension adds a new layer of instructions and hardware requirements to the base ISA as shown on Figure 2.1.

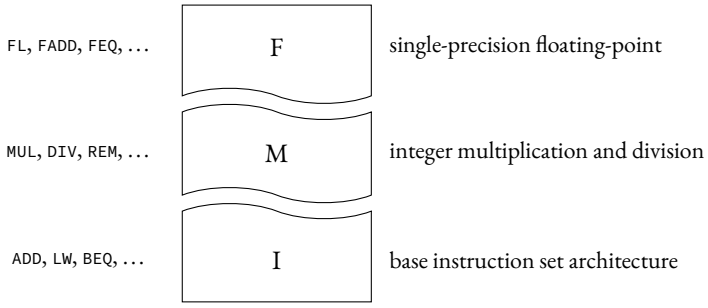


Figure 2.1: Base integer ISA with the extensions M and F and the instructions they add to the base.

2.1.1 Naming Convention

All the extensions are denoted following a specific naming convention. This naming convention is also followed when naming a specific CPU implementation to help identify which extensions are implemented. The rules for naming a RISC-V core are the following:

1. The name should start with `RV` to denote the RISC-V ISA.
2. Followed by the width of the register file which can either be 32, 64 or 128 bits.
3. The last part of the core name are the letters corresponding to the implemented extensions as shown in Table 2.1.

Following these three rules, a 32 bit RISC-V core with integer multiplication and division, atomic instructions and supporting instruction compression, has the denomination `RV32IMAC`. Extensions can be implemented as needed and non-standard ones can be added as long as they respect the naming convention. Having such a convention makes it easy to identify the supported features on a particular core implementation.

In the next section, we are going to look at another central aspect of the RISC-V ISA, namely the registers and their usage.

Extension	**Description**
I	Compulsory integer arithmetic
M	Integer multiplication and division
A	Atomic instructions
F	Single-precision floating-point
D	Double-precision floating-point
G	Shorthand for the above extensions
C	16 bit compressed instructions
X*ext*	Non-standard extension *ext*

Table 2.1: Non-exhaustive RISC-V extensions list. A full and up-to-date version of this list can be found on Wikipedia. [7]

2.1.2 Register Definition

As well as the other ISAs of the RISC family, RISC-V is a load-store architecture. This divides instruction into two categories:

1. memory access (to load and store data from and to the main memory)
2. ALU operations (these are only possible between registers or registers and immediate values).

The RISC-V base ISA defines 32 integer registers and the floating-point extension adds another 32 floating-point registers. There is also a separate `PC` register which holds the current memory address of the program. The base integer registers are shown on Table 2.2. All of them are general-purpose registers except for the first one (`x0`) which is the so-called zero register and, if read, always returns 0. The purpose of the remaining registers is defined by the ABI[1]. The ABI defines eight registers for function arguments `a0`–`a7` and the two first argument registers `a0` an `a1` are also used as return values, if needed.

[1] An application binary interface (ABI) is a common interface between two binary program modules [4]. We can compare it to an API which ensures a common interface for high-level programs, an ABI does the same on the machine code level. For example, the ABI defines the calling conventions and how variables should be passed to functions using the available registers.

Finally, each register can either be caller- or callee-saved, which as the name implies, defines if the register must be saved by the caller — thus the called function can change those registers without having to restore them before returning — or the called function.

Register	ABI Name	Description	Saver
`x0`	`zero`	Hard-wired zero	—
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	—
`x4`	`tp`	Thread pointer	—
`x5`	`t0`	Temporary/alternate link register	Caller
`x6–x7`	`t1–t2`	Temporaries	Caller
`x8`	`s0/fp`	Saved register/frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10–x11`	`a0–a1`	Function arguments/return values	Caller
`x12–x17`	`a2–a7`	Function arguments	Caller
`x18–x27`	`s2–s11`	Saved registers	Callee
`x28–x31`	`t3–t6`	Temporaries	Caller

Table 2.2: Assembler mnemonics for RISC-V integer registers, and their role in the first standard calling convention. [18, p. 137]

2.1.3 Instruction Formats

We mentioned in Chapter 1 that the RISC-V ISA uses fixed-length instructions. In fact, each instruction is 32 bits long, except when using the compressed instructions extension. In that case, some instructions are encoded using only 16 bits. This is similar to ARMs Thumb mode. In this document we are going to focus on the 32 bits base instruction form.

There are six possible types for an instruction. The difference between these types is the format of the instruction, based on the usage of immediate values and the number of registers used for the instruction. These different types are shown on Figure 2.2. The type of instruction is defined by its opcode. The fields `funct7` and `funct3` are, if present, used to further identify the instruction's operation, and their name reflects their size (`funct7` is 7 bits wide and `funct3` is 3 bits wide). The fields `rs1` and `rs2` are used to specify the two source registers on which to perform the operation. `rd` represents the destination register, also only if applicable. Lastly, the `imm` fields are reserved bits to encode immediate values.

R-type for instructions using three registers (`add`, `sub`, `mul`, ...).

I-type for instructions with immediate. Thus, the second source register is not needed anymore. (`ori`, `srli`, ...)

S-type for store instructions. Since data is stored to memory, there is no destination register needed. (`sw`, `sh`, ...)

B-type for branching instructions. The format is similar to S-type instructions except the immediate representing the — `pc` relative — offset from the instruction after the branch, to the target. (`beq`, `blt`, ...)

U-type for instructions with upper immediate. We can use this instruction if we want to store up to a 20 bits long immediate value in the destination register. If we want to store a 32 bits immediate value we need two instructions, first we write the upper 20 bits of a register with an `lui` instruction and secondly we can set the low 12 bits using an `addi` instruction.

J-type for jump instructions. While branching instructions are `pc` relative, jump instructions can jump to any location in memory, thus the 20 bits immediate value represents the destination memory address.

These six types are the compulsory ones which come with the base ISA. Other RISC-V extensions may add further instruction formats. In the next section, we are going to look at the main differences between ARM and RISC-V ISAs.

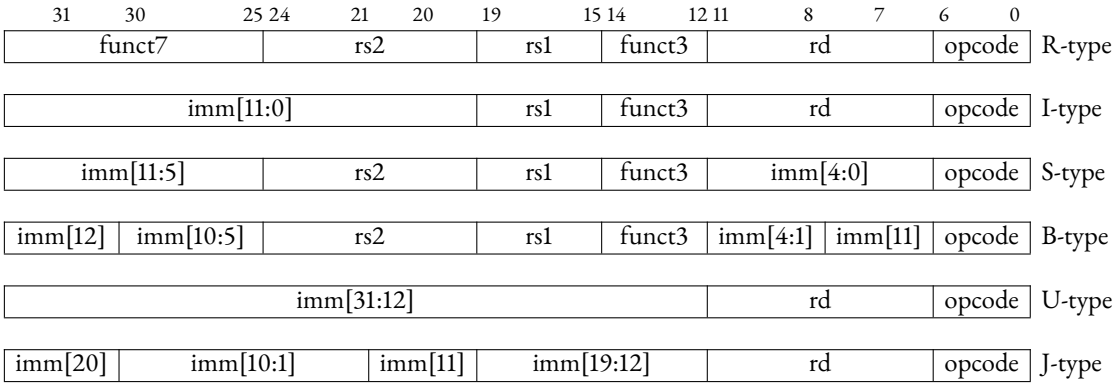


Figure 2.2: RISC-V base instruction formats showing immediate variants. [18, p. 16]

2.2 Comparison with ARM

ARM is the biggest company that sells RISC-based ISAs. Their architectures can nowadays be found in most mobile phones and tablets. Thus it is interesting to try to compare them with RISC-V. Since, like RISC-V, ARM also has multiple ISAs, we are going to compare the RISC-V `RV32I` ISA with the `Cortex-A32 Armv8-A` from ARM.

Both have an in-order CPU execution pipeline, however, ARM has an eight-stage pipeline while RISC-V only has five stages. More stages generally mean less pipelining hazards [3, p. 11–12] but also increases the complexity of the hardware. Another major difference with ARM is that RISC-V does not have processor flags[2] [12, p. 295]. This means that the such bits have to be calculated explicitly and stored in a register. Again, RISC-V has decided to keep the ISA as simple as possible.
There are also fewer instructions in RISC-V with many pseudo instructions. For example, the instruction `MV rd,rs` used to move data between registers is actually just a pseudo-instruction which maps to the equivalent `ADDI rd,rs,0` instruction. This reduces once more the size of the instruction set.

Now that we understand some of the differences between ARM and RISC-V, the next section will look at the advantages of RISC-V over ARM in fields like education or industry.

2.3 RISC-V Educational and Industrial Benefits

RISC-V has multiple advantages becoming the ISA of choice when it comes to learning about computer architectures, compiler design or programming in assembly. Its first asset is to be a reduced instruction set computer architecture which is easier to learn due to its limited number of instructions. Furthermore, with its segmented ISA it is possible to start simple, for example only considering the integer base ISA and its instructions. Once the concepts are learned, more complex aspects can be introduced using for example the floating-point or atomic extensions.

Teachers can also build RISC-V cores or simulators which exactly correspond to their needs and the scope of a particular course. There are also a lot of FPGA RISC-V cores freely available which can be used as learning material and experimented with.

From an industrial point of view, an open-source ISA mainly represents huge savings for a company which does not have to pay the royalties that come with other commercially available chips. It will also allow companies to design their own application-specific chips according to their needs.

These are only a few of the benefits offered by the RISC-V ISA. In the next chapter, we are going to compare some hardware and frameworks that we will use for the rest of the project.

[2] Processor flags include bits such as the carry, zero or sign bit.

3 Hardware and Software Comparison

This section is dedicated to the analysis and comparison phase of this project. We had to choose a RISC-V based board as well as a modern framework capable of producing bare metal programs for the chosen board.

3.1 Board Comparison

There are multiple commercially available RISC-V based boards. For this project, we have taken a deeper look at two of them. The first one is the HiFive1 Rev B produced by SiFive which is available for $59 [11]. The second one is the MAix Bit by Sipeed which comes at a slightly lower price point of $13 [13].

Given the educational purpose of this project, the simplicity of the board will be one of the main criterion for the comparison, alongside with the price, and the embedded peripherals. The following two sections are dedicated to introduce each board and its chip, followed by a thorough explanation of the choice made between the two options.

3.1.1 HiFive1 Rev B

The first board in our comparison is SiFive's HiFive1 Rev B which runs on the Freedom E310 G002 chip. This has been one of the first commercially available RISC-V boards and it has all the features expected on a development board like this, such as GPIO ports, UART, SPI, I^2C and more. It even embeds an ESP32 chip for Wifi and Bluetooth.

The Freedom E310-G002 chip which is also produced by Sifive, contains a 32 bit E31 RISC-V single-core CPU. The chip has 16 KiB of RAM as well as an L1 instruction cache of 16 KiB. An architecture overview of the E310-G002 chip is shown on the Figure 3.1. The E31 CPU supports following RISC-V ISA extensions:

1. Compulsory integer arithmetic (I)
2. Integer Multiplication and division (M)
3. Atomic instructions (A)
4. Compressed instructions (C)

The HiFive1 is, at the time of writing, the most popular RISC-V based board and already partially supported by some compilers as described in Section 3.2.

3.1.2 MAix Bit

The second board in our comparison is the MAix Bit by Sipeed which runs on a Kendryte K210 chip. This board is, compared to the HiFive1 Rev B, more powerful with his dual-core 64 bit CPU, 32 KiB instruction and data cache, and 8 MiB of RAM. The Kendryte K210 chip is optimized for computer vision and artificial intelligence. As we can see on the architecture overview in Figure 3.2, the chip has a dedicated KPU unit to run convolutional neural networks as well as an FFT unit for signal processing using Fast-Fourier Transforms frequently used in computer vision.

The CPU cores in the Kendryte K210 chip support following RISC-V ISA extensions:

1. Compulsory integer arithmetic (I)
2. Integer Multiplication and division (M)
3. Atomic instructions (A)
4. Single-precision floating point (F)
5. Double-precision floating point (D)
6. Compressed instructions (C)

3.1.3 Board Choice

We chose to focus on the HiFive1 Rev B in this project for multiple reasons. The first reason is its simplicity, we do not need AI, digital signal processing capabilities, or even floating point support. Secondly, a dual-core processor would bring unnecessary complexity to the compilator such as cache management and synchronization between the cores. Thus the HiFive1 Rev B is a good choice to get started with the RISC-V ISA and allows us to understand compiler design and keep it as simple as possible.

Furthermore, based on our language and compiler choice described in Section 3.2, the HiFive1 Rev B offers a better starting point due to the fact that it is already partially supported on the chosen compiler. Introducing a whole new board to a unknown compiler would be a very demanding task and would probably not be feasible in the scope of this project.

In the next section, we describe the different available languages/frameworks and their benefits.

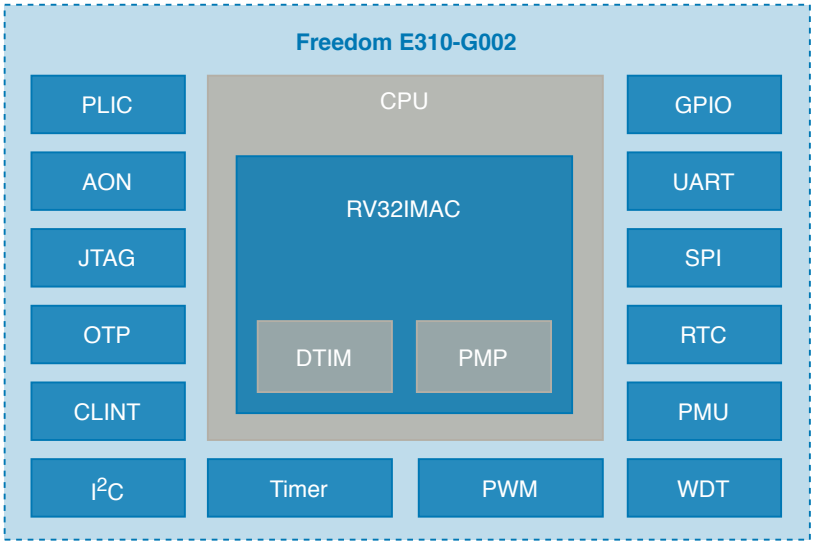


Figure 3.1: Freedom E310-G002 architecture overview.

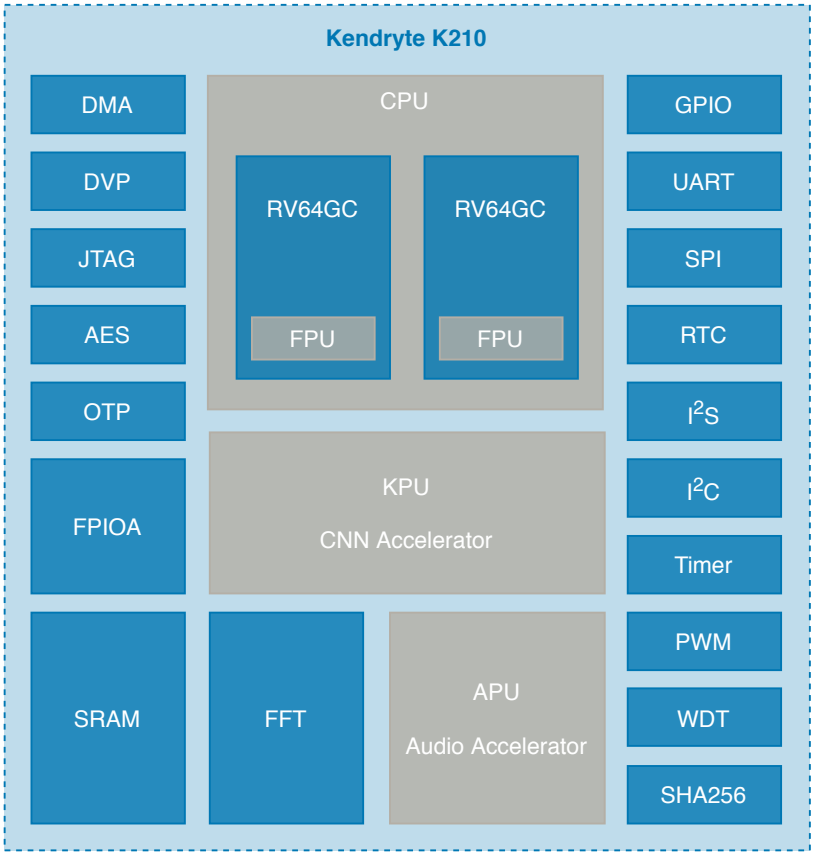


Figure 3.2: Kendryte K210 architecture overview.

3.2 Compiler and Language Comparison

For this project we needed to find a modern language that can be used for bare metal programming on embedded systems. We have focused on two popular frameworks each based on another programming language and with their own compiler. The main criterion to choose the framework is the current support of the RISC-V architecture and the HiFive1 Rev B board.

3.2.1 TinyGo

TinyGo is a Go compiler written in Go and based on LLVM. TinyGo can run programs on microcontrollers and already supports most features of the HiFive1 Rev B board. The project is open-source and available on GitHub[1].

Go is an open-source programming language developed by Google to provide an alternative to C++. Go is a garbage collected and statically typed language. Go is an easy language to learn since the syntax and the language's concepts are designed to be clear and unambiguous.

The TinyGo compiler is a new Go compiler and runtime library focused on producing very small binaries for embedded systems or WebAssembly. To do so, TinyGo uses an LLVM backend which provides a lot of flexibility regarding the supported targets.

3.2.2 Rust

The second language/framework we have analyzed for this project is Rust. Rust is a language targeting embedded systems. It does not have garbage collection or a runtime library and its main focus is execution speed. Rust is also open-source and available on GitHub[2].

RISC-V and more specifically the HiFive1 Rev B is also partially supported on Rust. The disadvantage of Rust however, is that its standard library currently is unusable on bare metal because it is tightly coupled with the operating system. Another, rather subjective, disadvantage of Rust is that it is "harder" to learn, maybe because of its very strict memory safety guarantees.

3.2.3 Framework Choice

We choose to use TinyGo because of the clear design of the Go language. Moreover, the TinyGo community is very active and it is crucial in a project like this, where we start from the very bottom, to be able to ask questions. The popularity of the Go language also makes it a very promising choice. The fact that TinyGo has memory management with garbage collection is also interesting to explore.

In the next chapter, we are going to look at the TinyGo compiler internals and explain how it converts Go code to bare metal RISC-V assembly.

[1] https://github.com/tinygo-org/tinygo, visited on 05/06/2020.

[2] https://github.com/rust-lang/rust, visited on 05/07/2020.

4 The TinyGo Compiler

There are several reasons which pushed the developer of TinyGo to create a new compiler instead of modifying the existing Go compiler to produce binaries for microcontrollers. The main reason is that the standard Go compiler (`gc`) does not — or only partially — support instructions sets like AVR or RISC-V. This could be resolved by using `gccgo` which is an alternative Go compiler implementation with support for more architectures, but even then, there would still be problems which come from the Go runtime itself. For example, a simple "hello world" program using the built-in `println` function produces a binary which is almost 1 MB big. Because the compiler is optimized for speed and not code size or memory consumption, some design choices such as memory allocation on interface conversion are not suited on microcontrollers where code size is primordial. Also the existing Go compilers never have been developed with embedded systems in mind.

However, thanks to the excellent Go standard library and its packages for parsing Go code, coupled with the LLVM optimizer, it is relatively simple to build a new compiler which produces small sized binaries. The TinyGo compiler is released under a BSD 3-clause license just like the Go project itself. The code is available on GitHub[1] and contributions are welcome.

The TinyGo compiler is mostly based on the Go standard library to parse the programs and relays on LLVM for the optimization and translation to machine code for the targeted architecture. TinyGo also implements a new runtime library for compiler intrinsics like memory allocation, scheduling and more.

The compilation of a program is based on multiple steps of transformation which are linked into a pipeline. Every step has a well-defined purpose and generally takes an input and transforms it into a simpler output. The TinyGo pipeline is discussed in more details in the next few sections.

4.1 Compiler Pipeline Overview

As with most compilers, the TinyGo pipeline can be broken down to several phases [15, 17]:

Lexical analysis this is the first phase for every compiler which consists of reading the program file and dividing the text into *tokens*. Every token corresponds to a valid symbole of the programming language.

Syntax analysis this phase takes the tokens produced by the previous lexical analysis and arranges them in an abstract syntax tree (AST) that represents the syntactic structure of the source code.

Type checking this phase analyzes the AST and determines if the semantic of the program is correct. This includes for example identifier resolution[2], type deduction[3] or constant evaluation[4].

[1] `https://github.com/tinygo-org/tinygo`, visited on 05/06/2020.
[2] Identifier resolution determines for every name in the program, the declaration it refers to.
[3] Type deduction is used to deduct the type of an expression or report an error if the expression has no type or is invalid
[4] Constant evaluation simply determines the value of every constant in the code.

static single-assignment (SSA) this phase converts the code to its SSA form. This ensures that every variable of the program is assigned exactly once, and used after its declaration.

Intermediate code generation this phase generates a platform independent LLVM intermediate representation (IR) of the code. This IR can then be compiled by LLVM to actual machine code for the targeted platform.

Optimization this phase optimizes the IR with some TinyGo-specific optimizations to keep the binary size low.

LLVM compilation this is the final phase which compiles the optimized IR to machine code using the LLVM compiler infrastructure.

The TinyGo compiler uses the Go standard library for some of the phases and also implement its own mechanisms to produce an optimized IR of the code which can be fed to LLVM. These different phases are represented on Figure 4.1.

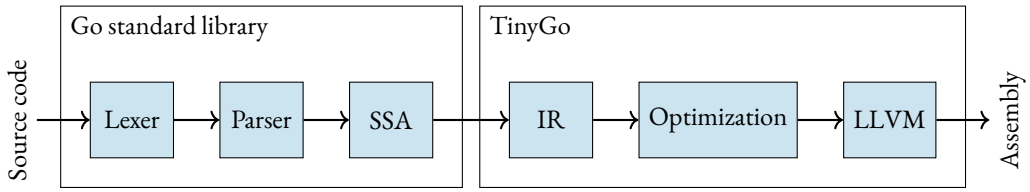


Figure 4.1: The TinyGo compiler pipeline.

For the sake of understanding, let us define a simple code example and analyze it after each one of the phases. We are going to do a very straightforward program which adds two integers together and computes the result, the code is shown in Listing 4.1.

```
1  package main
2  func main() {
3          a := 1
4          b := 2
5          if a == 1 {
6                  b = 3
7          }
8          c := sum(a, b)
9  }
10
11 func sum(x, y int) int {
12         return x + y
13 }
```

Listing 4.1: Simple Go code with a main function and a sum function which adds together two integers.

4.2 Lexical Analysis

The lexical analysis produces three information for each token, its position in the source file, its type and if applicable its literal string. The `go/scanner` package of the standard library is used in TinyGo to do the lexing and produces an output as shown in Listing 4.2. The first column represents the position of the token in the source file, the second column is the token which it represents (for operators, delimiters, and keywords the string is the actual token character sequence), and the last column is the literal string of the token if applicable. The complete list of possible tokens can be found in the `go/token` package[5]. We can notice how every "word" of the Go language vocabulary in our program has been assigned to

```
1:1     package "package"       5:10    INT     "1"             11:1    func    "func"
1:9     IDENT   "main"          5:12    {       ""              11:6    IDENT   "sum"
1:13    ;       "\n"            6:3     IDENT   "b"             11:9    (       ""
2:1     func    "func"          6:5     =       ""              11:10   IDENT   "x"
2:6     IDENT   "main"          6:7     INT     "3"             11:11   ,       ""
2:10    (       ""              6:8     ;       "\n"            11:13   IDENT   "y"
2:11    )       ""              7:2     }       ""              11:15   IDENT   "int"
2:13    {       ""              7:3     ;       "\n"            11:18   )       ""
3:2     IDENT   "a"             8:2     IDENT   "c"             11:20   IDENT   "int"
3:4     :=      ""              8:4     :=      ""              11:24   {       ""
3:7     INT     "1"             8:7     IDENT   "sum"           12:2    return  "return"
3:8     ;       "\n"            8:10    (       ""              12:9    IDENT   "x"
4:2     IDENT   "b"             8:11    IDENT   "a"             12:11   +       ""
4:4     :=      ""              8:12    ,       ""              12:13   IDENT   "y"
4:7     INT     "2"             8:14    IDENT   "b"             12:14   ;       "\n"
4:8     ;       "\n"            8:15    )       ""              13:1    }       ""
5:2     if      "if"            8:16    ;       "\n"            13:2    ;       "\n"
5:5     IDENT   "a"             9:1     }       ""
5:7     ==      ""              9:2     ;       "\n"
```

Listing 4.2: Representation of the code after the lexical analysis phase.

his semantic value, known as token. The scanner also automatically added semicolons at each new line. This is done to respect the EBNF formal grammar definition of the Go language[6].

4.3 Syntax Analysis

The next phase of the pipeline is syntax analysis. For this we need to introduce the concept of an abstract syntax tree (AST). An AST, is a tree representation of the tokens produced in the earlier lexing phase. They are used during the entire compilation process and can be enhanced with further information throughout the different phases. A graphical AST representation for the main function of our program

[5] https://golang.org/pkg/go/token/#Token, visited on 05/06/2020.
[6] https://golang.org/ref/spec, visited on 05/08/2020.

is depicted on Figure 4.2. The complete textual representation of the AST is available in this project's management git repository as `appendices/ast.txt`.

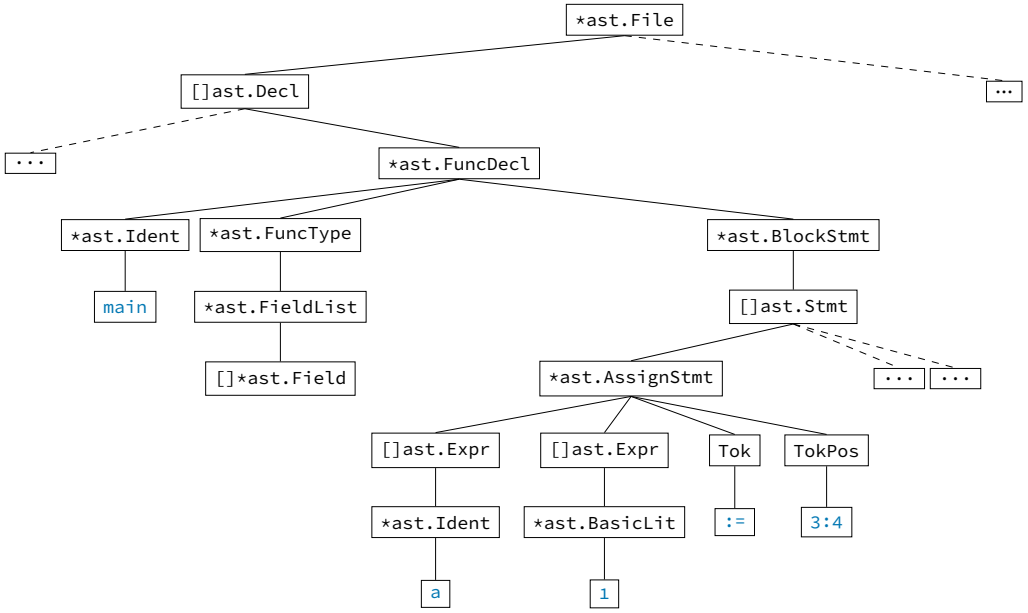


Figure 4.2: AST representation of a variable assignment (`a := 1`) inside the main function. Only the first variable assignment has been kept since the others are almost identical. The nodes depicted in blue represent the tokens from the earlier lexical analysis.

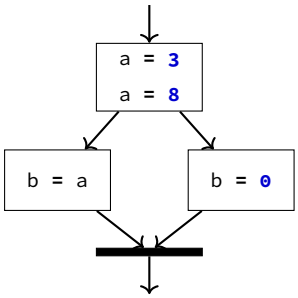
The AST is generated by the `go/ast` package in the Go standard library. With the help of the generated tree, it is now much easier for the compiler to navigate through our code. We will use this AST during the next steps in our compilation process.

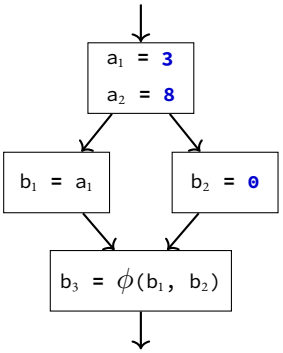
4.4 Transformation to Static Single Assignment Form

The static single-assignment form of a code is an intermediate representation of the code which guarantees that each variable is assigned exactly once and defined before its use. Multiple assignations of the same variable will produce different versions of the variable. When the value of a variable depends on some branching condition and could have multiple values, a special ϕ instruction is inserted which has as many arguments as there are possible branches with a definition for the variable.

The advantage of SSA form is that it intrinsically performs or enables compiler optimizations, such as live-range splitting [2] for register allocation, constant propagation [1], dead code elimination [8] and more. To illustrate this let us look at the code in Figure 4.3a where, for a programmer it is trivial to see that the first assignment is useless because the variable `a` gets overwritten by its second assignment. In SSA form, this becomes clear even for the compiler, since the variable a_1 is never used. We can also see how SSA form affects the control flow graph (CFG) of our program on Figures 4.3b and 4.3c. A new version, depending on the ϕ function, is created for variable `b` which contains the right value depending on the control flow in the past.

```
var a, b int16
a = 3
a = 8
if condition {
        b = a
} else {
        b = 0
}
```





(a) Source pseudo-code (b) Control flow graph (c) control flow graph in SSA-form

Figure 4.3: Pseudo-code for a program with unused assignment for variable `a` and branch-depending assignment for variable `b`, as well as the CFG representation before and after converting to SSA-form.

In the TinyGo compiler, the transformation to static single-assignment form is done using the `go/ssa` package of the Go standard library. The result of this generation for the source code of Listing 4.1 is shown in Listing 4.3. The result has been limited to the `main` function since the `sum` function does not introduce new interesting SSA specific instructions. The full representation can be found in this project's management git repository as `appendices/ssa.txt`.

```
1  func file.go.main:
2  0: entry:
3          t0 = 1:int == 1:int
4          if t0 goto 1 else 2
5  1: if.then:
6          jump 2
7  2: if.done:
8          t1 = phi [0: 2:int, 1: 3:int] #b
9          t2 = sum(1:int, t1)
10         return
```

Listing 4.3: SSA-form of the main function in Listing 4.1 generated by the `go/ssa` package.

The produced SSA form has replaced our variable names and has generated a branch table for our if condition. We can also notice the usage of the ϕ function for the assignment of variable `b`. This SSA is now converted by the TinyGo Compiler to LLVM IR[7]. Since Go SSA is higher level as LLVM IR some constructs such as goroutines or interfaces have to be transformed using a TinyGo specific logic. However, the majority of the transformations come down to trivially lowering the available Go SSA to LLVM IR, whose result is shown in Listing 4.4. In the next phase of the compilation, the LLVM IR will be optimized to reduce the produced binary size.

[7] More information on LLVMs intermediate representation in Section 4.6.

4.5 Constant Initializations and Optimizations

The optimization step is very important to reduce the size of the produced binaries. Normally, Go initializes constant variables at runtime but TinyGo does it — if possible — at compile time. This produces smaller binaries since only the initial data of those globals has to be copied from flash memory (the `.data` section) to RAM, instead of all the instructions needed to initialize the values at runtime. Also, constants are recognized by LLVM and can be used in constant propagation and dead code elimination which can also reduce the final binary size. For example, the IR of Listing 4.4 gets fully optimized out since the result of the `sum` function is never used and thus the complete `main` function contains dead code and is therefore useless.

In addition to the global variables interpretation step introduced by TinyGo, LLVM also has multiple optimization passes which are applied to the IR depending on the chosen optimization level.

4.6 LLVM Compilation

LLVM is a compiler infrastructure which allows developers to write compilers for any arbitrary languages using a reusable toolchain. With LLVM, a developer can focus on the front end of a compiler and leave the back end such as code generation for a specific architecture to the framework. This offers a great advantage because an LLVM-based compiler can automatically support all architectures supported by the framework without having to handle too much platform dependent code generation. There are a lot of existing LLVM-based compilers, the most notorious one being the C/C++ compiler Clang.

The framework uses an intermediate representation of the code which can be optimized using standard compiler optimization tools and lowered to platform dependent machine code. LLVM IR, unlike RISC-V assembly, has infinitely many registers, ϕ functions, and other higher-level constructs.

LLVM compilation is the last step of the compiler pipeline. In this step, the virtual registers used in the IR get eliminated and replaced by physical registers using a register allocation algorithm. It is also in this step that ϕ functions are replaced by "real" machine instructions.
Once the LLVM compilation has finished, the compiler pipeline has come to its end and produced a binary from the input Go code. This binary has been optimized by TinyGo mostly in size in order to be able to run on a microcontroller.

This also closes this chapter on the TinyGo compiler and we can now explore another very important aspect of TinyGo, which is its runtime library.

```
define dso_local void @file.go.main(i8* %context, i8* %parentHandle) unnamed_addr {
entry:
    br i1 true, label %if.then, label %if.done

if.then:                                          ; preds = %entry
    br label %if.done

if.done:                                          ; preds = %if.then, %entry
    %0 = phi i32 [ 2, %entry ], [ 3, %if.then ]
    %1 = call i32 @file.go.sum(i32 1, i32 %0, i8* undef, i8* undef)
    ret void
}
```

Listing 4.4: Non-optimized LLVM intermediate representation.

5 The TinyGo Runtime Library

The TinyGo runtime implements some compiler intrinsics such as scheduling or memory allocation. It also defines all the machine-specific code of the TinyGo compiler. To stay on this project's scope we are only going to focus on the memory management features of the TinyGo runtime as well as the implemented features for the HiFive1 Rev B board. Other features such as interrupt support, function lowering or goroutine support are also implemented in the TinyGo runtime but not relevant for this project.

5.1 Memory Management

Memory management in TinyGo is a crucial part of the runtime library. Go heavily relies on heap allocation and thus needs a good garbage collector. TinyGo tries to reduce heap allocation as much as possible. In the next sections, we are going to explain how heap allocation works and why we have to avoid it. We are also going to talk about the garbage collector implemented in TinyGo.

5.1.1 Heap Allocation

In general heap allocation is used for dynamic data. The advantage of heap allocation is that it allows a programmer to allocate dynamically sized memory regions during runtime. On the other hand, heap allocation is slow. In fact, before allocating memory on the heap we have to find a free space in memory which is big enough for our data and since TinyGo implements a garbage collector we have to keep track of the allocated memory and free it once its use is over. Furthermore, on a computer — and especially on an embedded system — memory is limited. With dynamic allocation it is not possible to know at compile time whether or not our program will eventually run out of memory. To prevent heap allocation, TinyGo uses two main optimizations:

- Escape analysis
- `string` to `[]byte` conversion optimization.

Normally, taking the pointer of a local variable produces heap allocation. However, using escape analysis TinyGo will check if the pointer escapes the scope of the function in which the local value has been declared. If that is not the case, the variable can be stored on the stack. An example of code with heap allocation and another where heap allocation is prevented using escape analysis is shown on Listing 5.1.

Another optimization that TinyGo tries to do is when converting `string` to `[]byte`. In Go, strings are constant and slices are not and therefore a conversion from the first type to the latter produces heap allocation. However, if the slice's underlaying buffer never gets written to — which means it stays constant — then TinyGo avoids heap allocation. An example of this behavior is depicted on Listing 5.2.

```go
var globalByte *byte

func foo() {
    counter := byte(0)
    globalByte = &counter
}
```

```go
func foo() {
        counter := byte(0)
        bar(&counter)
}

func bar(b *byte) {
        print(*b)
}
```

Listing 5.1: The code on the left side produces heap allocation since the pointer to the local variable `counter` escapes the scope of function `foo`. The code on the right-hand side, however, does not produce heap allocation since the pointer to the local variable counter never escapes the function `foo`.

```go
func PrintString(s string) {
        buf := []byte(s)
        buf = append(buf, '\x00')
        for _, c := range buf {
                print(c)
        }
}
```

```go
func PrintString(s string) {
        buf := []byte(s)
        for _, c := range buf {
                print(c)
        }
}
```

Listing 5.2: The code on the left side produces heap allocation since the buffer of slice `buf` is modified. The code on the right-hand side, however, does not produce heap allocation since the buffer of slice `buf` never gets modified.

To allocate heap memory, TinyGo uses a simple fixed-size blocks allocation algorithm. Memory is segmented in multiple equally sized blocks which are allocated and freed as needed. Such a memory management model is often used in embedded systems because it is very fast. The major drawback is that using this kind of algorithm causes a lot of fragmentation of the available memory space. Let us take for example the situation depicted on Figure 5.1, where we want to allocate two blocks of memory (the size of a block is not important). The algorithm will scan the heap for free space and will allocate the two blocks in the first place where there is enough contiguous memory. In this particular example, it would not be possible to allocate four blocks of memory even though there would be enough non-contiguous free memory blocks. This is called fragmentation.

The state of the different blocks as well as the allocation table is explained in the next section.

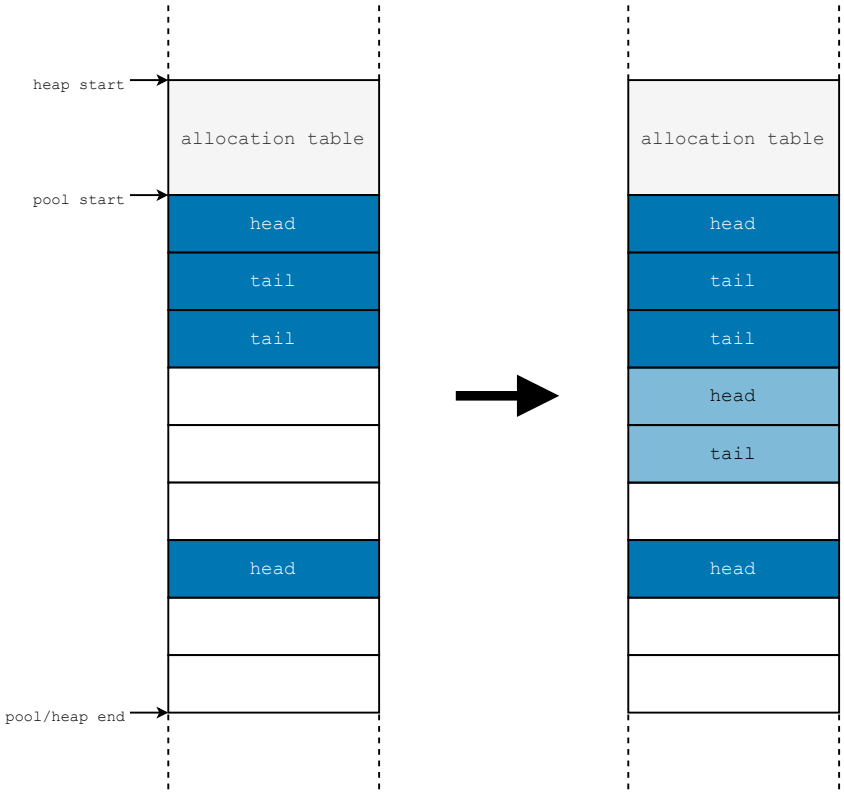


Figure 5.1: Heap allocation of two blocks when there is enough memory.

5.1.2 Garbage Collection

Now that we understand how the heap memory is allocated it is important to understand how the memory gets freed. TinyGo implements a naïve mark-and-sweep [15, p.269–270] garbage collection algorithm. This algorithm is part of the tracing garbage collection algorithm family, which search for alive objects. Each heap block can be in one of the four states listed on Table 5.1. The state for each block is encoded on 2 bits and stored at the beginning of the heap. The heap blocks start after the allocation table (this address is called the pool start). The heap blocks are then managed the following way:

- At initialization every heap block is set to the `free` state.
- When a memory block gets allocated, its state is changed to `head`. If there are multiple blocks allocated, then the blocks following the `head` block are set to the `tail` state.
- When a garbage collector cycle occurs, reachable[1] `head` blocks are set to `mark` state. This is called the marking phase. Unreachable `head` blocks will remain `head` and can be removed during the sweep phase.
- The sweep phase frees all remaining `head` blocks and, if there are some, their corresponding `tail` blocks by setting them to state `free`.
- Finally all the `mark` blocks are changed back to `free` state.

An example garbage collection cycle is shown on Figure 5.2. The most important phase in the garbage collector is the marking of the reachable blocks. To do this, TinyGo does a depth first search on the global variables sections (`.data` and `.bss`) as well as on the stack. During the search, TinyGo sweeps the previously mentioned memory regions and checks if it finds a pointer to the heap. If this is the case, the garbage collector will search the heap blocks referenced by the pointer to check if they also contain pointers to the heap. This situation is shown on Figure 5.3.

During the development of the SSD1351 OLED display driver, we found a bug in the garbage collector that affects only the HiFive1 Rev B board. The problem was that the garbage collector was blocked in an infinite loop as soon as it executed a marking phase. For the garbage collector to be able to scan the current registers for heap pointers, they have to be saved on the stack. This is done thanks to some assembly code which saves the registers to the stack and calls the function that will do the marking pass. The code to do this is shown in Listing 5.3. The problem was that line 9 of the code was missing. When doing a `call` instruction, the `ra` register gets overwritten to the memory location after the `call` instruction. Without restoring the `ra` register to its original value, the `ret` instruction returns inside the same function, provoking an infinite loop. The whole code, as well as the pull request #1071 for this bug can be found on the TinyGo GitHub repository[2].

Bits	State
`0b00`	Free
`0b01`	Head
`0b10`	Tail
`0b11`	Mark

Table 5.1: Heap block states.

[1] A reachable block is a block with a pointer in memory that points to this heap block.

[2] `https://github.com/tinygo-org/tinygo/pull/1071`, visited on 05/06/2020.

```
1  tinygo_scanCurrentStack:
2  addi sp, sp, -64 // Make space on the stack.
3  sw   ra, 60(sp)  // Save return address.
4  //...            // Save all registers (s0-s11).
5
6  mv a0, sp             // Set the stack pointer as function argument.
7  call tinygo_scanstack // Scan the stack.
8
9  lw ra, 60(sp)   // Restore return address.
10 addi sp, sp, 64 // Restore stack state.
11 ret             // Return to the caller.
```

Listing 5.3: Assembly code to save registers to the stack and scan the stack.

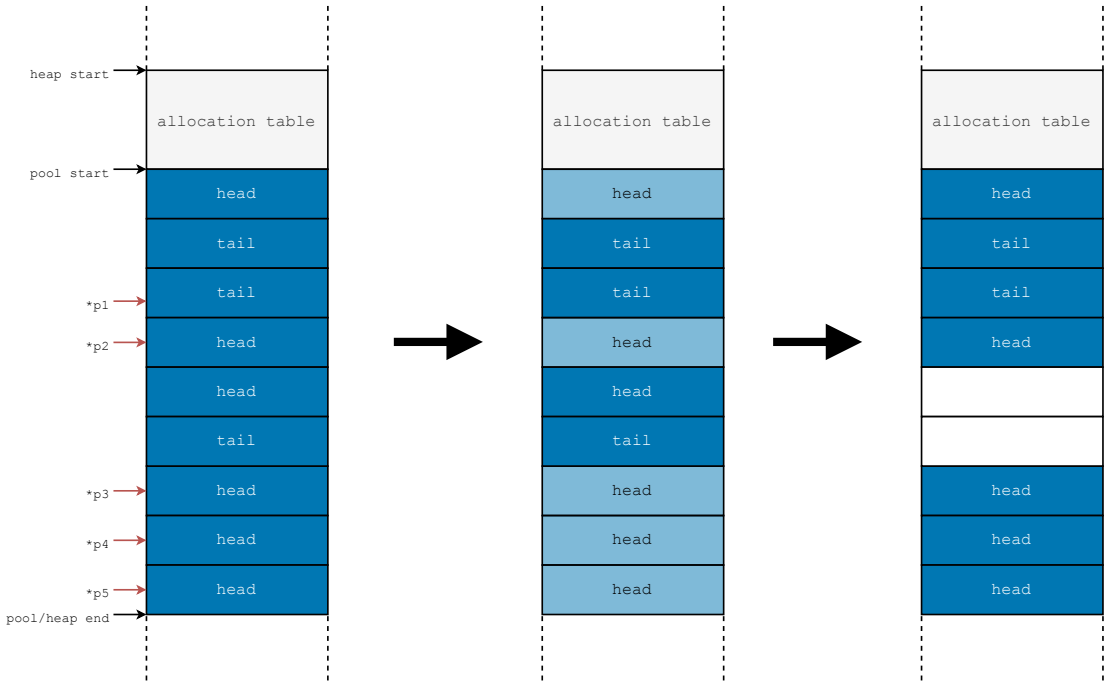


Figure 5.2: Garbage collector cycle when two blocks can be freed. The reachable `head` blocks are marked and the remaining `head` blocks are freed.

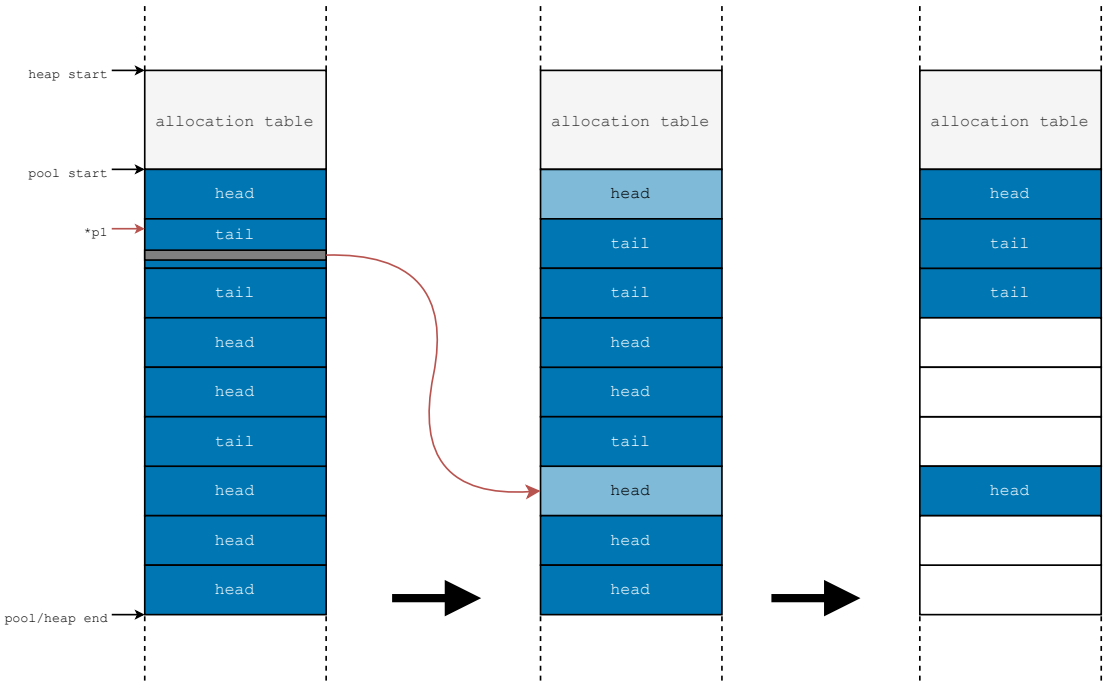


Figure 5.3: Garbage collector depth-first search illustration. There is only one pointer on the stack `*p1` which points to the heap. The pointed heap block is marked and the GC looks for eventual other heap pointers inside the heap block. In this case there is another heap pointer in the heap block which is never referenced elsewhere and would not have been marked without this recursive search.

5.2 Inter-Integrated Circuit (I^2C)

One of the missing features of the HiFive1 Rev B was reading and writing using the I^2C controller. This seemed like a good starting point for contributing to the compiler and learning about the HiFive1 Rev B and its RISC-V architecture.

The I^2C controller embedded in the FE302-G002 chip is open-source and designed by OpenCores. It supports the two I^2C standard speeds of 100 kbps and 400 kbps and the default 7 bit addressing mode. The first step to implement the controller was learning about the protocol. An explanation of the important concepts of the I^2C protocol can be found in the next section.

5.2.1 Protocol Design

The I^2C protocol is a synchronous, multi-master, multi-slave, serial protocol widely used in communication with peripherals. The bus uses a clock line (SCL) and a bi-directional data line (SDA). There are two possible roles for a device in the I^2C protocol:

Master device the master device generates the clock and initiates the connection with slaves.

Slave device the slave device waits to be addressed by a master in order to respond.

The typical block diagram for an I^2C bus is shown on Figure 5.4. Since the bus is multi-master capable, it is possible to have multiple masters.

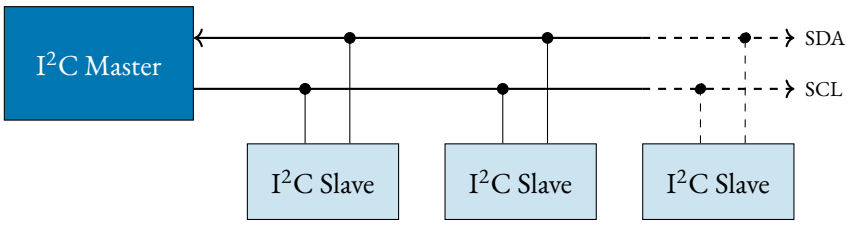


Figure 5.4: Typical I^2C block diagram with one master and 3 slaves. Every device shares the common SDA and SCL lines. Not shown in the diagram are the pullup resistors which have to be taken in consideration to avoid incorrect values or even signal loss.

Slave devices are addressed with a 7 bit address (10 bit addresses are also supported by the protocol but rarely used). Each slave must have a unique address to prevent collisions. An I^2C transaction is always started by a master device and delimited with a start and stop symbol. A representation of an I^2C transaction and its four main parts is shown on figure 5.5.

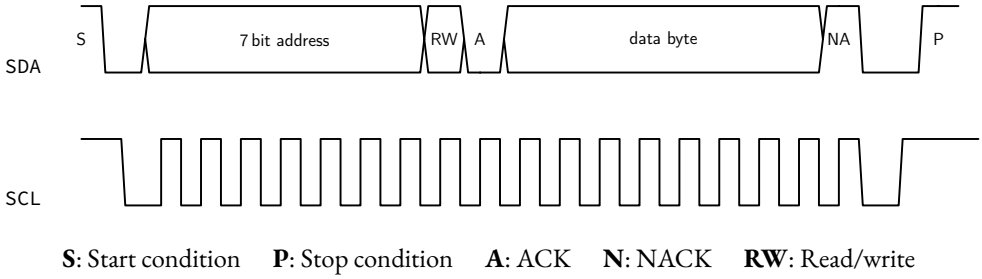
The four main parts of an I^2C transaction are the following [10]:

1. Start condition generation. The master pulls the SDA signal to low while SCL is high to indicate the beginning of the transaction.

2. Slave address transfer. The first byte sent by the master after the start condition is the 7 bit slave address followed by the read/write bit to indicate the data transfer direction (1=Read, 0=Write).

3. Data transfer. The data transfer in the direction indicated by the master occurs on a byte-by-byte basis, with each byte followed by an acknowledgment byte.

4. Stop condition generation. The master indicates the end of the transaction with a low-to-high transition on the SDA signal while the SCL signal is high.



S: Start condition **P**: Stop condition **A**: ACK **N**: NACK **RW**: Read/write

Figure 5.5: I²C transaction diagram.

5.2.2 Controller Core Implementation

As mentioned before, the I²C controller embedded in the FE310-G002 chip is designed by OpenCores. The controller has seven 8 bit registers used for configuration and data transfer. The controller's base address is `0x10016000` and every register is 4 Byte aligned. The complete code relative to the I²C interface can be found in pull request #982 in the TinyGo GitHub repository[3].

TinyGo defines an API for I²C as shown on Listing 5.4. This API has to be respected by each board that uses the I²C interface.

```
1  // All fields are optional and may be left out on a particular platform.
2  type I2CConfig struct {
3          Frequency uint32
4          SCL       Pin
5          SDA       Pin
6  }
7  // Not a real exported type, just here to serve as example.
8  type I2C interface {
9      Configure(I2CConfig)
10     Tx(addr uint16, w, r []byte) error
11 }
```

Listing 5.4: TinyGo machine API for the I²C interface.

The first step for the implementation on the HiFive1 Rev B is to define some board specific pins and variables in the `machine` package of TinyGo. TinyGo automatically generates structures for each peripheral of a specific chip based on its SVD[4] description. For example, our I²C controller's structure

[3] https://github.com/tinygo-org/tinygo/pull/982, visited on 05/06/2020.

[4] An SVD file contains the description of a device based on XML. Described is information such as the name, width and memory location of a peripheral's registers.

is represented as on Listing 5.5. The location of the specific controller is also automatically generated from the SVD file and declared in the `device` package of TinyGo, see Listing 5.6.

```
// Inter-Integrated Circuit Master Interface (FE310-G002 only)
type I2C_Type struct {
        PRER_LO volatile.Register32
        PRER_HI volatile.Register32
        CTR     volatile.Register32
        TXR_RXR volatile.Register32
        CR_SR   volatile.Register32
}
```

Listing 5.5: I^{2}C controller register structure generated from the SVD file in the `device` package.

```
// Peripherals.
var (
    // ...
    I2C0 = (*I2C_Type)(unsafe.Pointer(uintptr(0x10016000))) // Inter-Integrated Circuit
    ↪  Master Interface (FE310-G002 only)
    // ...
)
```

Listing 5.6: I^{2}C controller memory location generated from the SVD file in the `device` package.

Thanks to these definitions above, we have a high-level representation of our I^{2}C controller which we can define and use as shown on Listing 5.7.

```
// I2C controller definition
var I2C0 = I2C{
        Bus: sifive.I2C0,
}
// Sample usage (write enable bit of the control register)
I2C0.Bus.CTR.SetBits(sifive.I2C_CTR_EN)
```

Listing 5.7: Sample usage of the high-level abstraction of the controller.

Now we can start to look at the `Configure` function which has to do multiple things:

- Setup the GPIO pins used for SDA and SCL.
- Setup the I^{2}C controller's frequency.
- Enable the I^{2}C controller.

On the HiFive1 Rev B each GPIO port can have two hardware driven functions (IOF). The IOF can be selected for each pin using the `iof_sel` register. For the I^2C controller, GPIO pin 12 is used for the SDA line and GPIO pin 13 for the SCL line. Both pins must use hardware function `IOF0`.

As already mentioned, the controller supports multiple frequencies. To set the controller's frequency, we have to set two prescaler registers. The prescale value P can be calculated using the following formula, with F_{ctrl} being the controller's core frequency and F_{des} the desired frequency for the SCL signal:

$$P = \frac{F_{ctrl}}{5 \cdot F_{des}} - 1$$

The calculated prescale value must then be set in the corresponding registers. The eight least significant bits in the `PRERlo` register, and the eight most significant bits in the `PRERhi` register.

Lastly, the controller must be enabled to be ready to send and receive data. This is done by setting the enable bit in the control register for the controller. The code for the `Configure` function is shown in Listing 5.8.

```
1  func (i2c I2C) Configure(config I2CConfig) error {
2          // ...
3          var prescaler = i2cClockFrequency/(5*config.Frequency) - 1
4          // disable controller before setting the prescale registers
5          i2c.Bus.CTR.ClearBits(sifive.I2C_CTR_EN)
6          // set prescaler registers
7          i2c.Bus.PRER_LO.Set(uint32(prescaler & 0xff))
8          i2c.Bus.PRER_HI.Set(uint32((prescaler >> 8) & 0xff))
9          // enable controller
10         i2c.Bus.CTR.SetBits(sifive.I2C_CTR_EN)
11         // select IOF0 for both pins
12         config.SDA.Configure(PinConfig{Mode: PinI2C})
13         config.SCL.Configure(PinConfig{Mode: PinI2C})
14         return nil
15 }
```

Listing 5.8: HiFive1 Rev B I^2C controller configuration in TinyGo.

Now that the controller is configured we can define the `Tx` function, used to make one read/write transaction. It is important to note that in I^2C every transaction is initiated by the master. This means that the master also decides how many bytes to read and when he wants to read them. The algorithm for the `Tx` function is shown in Listing 5.9.

All the interactions with the controller are made using the registers. For example, to send the slave address we need four registers, the transmit register `TXR`, the receive register `RXR`, the command register `CR` and the status register `SR`. The transmit and receive register each only support respectively writing or reading. Thus, those two registers can be placed at the same memory location and the distinction between the two registers is made according to the access mode (read or write). This is the same for the command and status registers. This is why in TinyGo we only have the two registers `TXR_RXR` and `CR_SR`

```
r ← number of bytes to read from slave
w ← number of bytes to send slave
send start/repeated start signal
send slave address
wait for ACK from slave
if w ≠ 0 then
    while w > 0 do
        w ← w − 1
        write data byte
        wait for transmission to finish
        wait for ACK from slave
    end while
end if
if r ≠ 0 then
    while r > 0 do
        r ← r − 1
        read data byte
        send ACK to slave
    end while
    send NACK to slave
end if
send stop condition
```

Listing 5.9: Algorithm for the Tx function which does a single read or write I^2C transaction.

which actually map to four registers. A code example to show how the address of the slave is sent on the SDA line is shown on Listing 5.10. The complete commented code of the `Tx` function can be found in Appendix A.

```
1  func (i2c I2C) sendAddress(address uint16, write bool) error {
2          data := (address << 1)
3          if !write {
4                  data |= 1 // set read flag in transmit register
5          }
6          // write address to transmit register
7          i2c.Bus.TXR_RXR.Set(uint32(data))
8          // generate start condition
9          i2c.Bus.CR_SR.Set((sifive.I2C_CR_STA | sifive.I2C_CR_WR))
10         // wait until transmission complete
11         for i2c.Bus.CR_SR.HasBits(sifive.I2C_SR_TIP) {
12         }
13         return nil
14 }
```

Listing 5.10: Sample code to send the slave address.

The `Configure` and `Tx` functions are the only needed to implement full read/write capabilities for the I^2C interface. We can now write TinyGo code which uses the interface to test our implementation. We decided to write a code example using two I^2C peripherals. The first peripheral is the TMP102 digital thermometer and the second one is a SSD1306 display. The code in Listing 5.11 periodically reads the temperature value from the thermometer and displays it on the screen. Notice how this code uses the TinyGo drivers[5] package as well as the TinyFont[6] text library on lines 9–12.

To check that our I^2C implementation works as expected, we made some measures of the I^2C transactions using a Saleae Logic 8 logic analyzer. This allowed us to record the SDA and SCL signal and automatically parse the data that was sent and compare it with the values we wanted to send.

5.2.3 TMP102 Digital Thermometer Driver

The TMP102 driver used in the code example of Listing 5.11 is also a contribution made in the scope of this project to the TinyGo driver package. The full associated pull request #141 can be found in the TinyGo Driver Github repository[7].

The TMP102 temperature sensor is used in the MikroElektronika Thermo 3 Click chip. The sensor uses I^2C to transfer data. The chip's pinout diagram is shown on Figure 5.6. The role of the different pins is the following:

[5] https://github.com/tinygo-org/drivers, visited on 05/06/2020.
[6] https://github.com/tinygo-org/tinyfont, visited on 05/06/2020.
[7] https://github.com/tinygo-org/drivers/pull/141, visited on 05/06/2020.

```go
package main

import (
	"fmt"
	"image/color"
	"machine"
	"time"

	"tinygo.org/x/drivers/ssd1306"
	"tinygo.org/x/drivers/tmp102"
	"tinygo.org/x/tinyfont"
	"tinygo.org/x/tinyfont/freemono"
)

func main() {
	machine.I2C0.Configure(machine.I2CConfig{Frequency: machine.TWI_FREQ_400KHZ,})
	thermo := tmp102.New(machine.I2C0)
	display := ssd1306.NewI2C(machine.I2C0)
	thermo.Configure(tmp102.Config{})
	display.Configure(ssd1306.Config{
		Address: ssd1306.Address_128_32,
		Width:   128,
		Height:  32,
	})
	white := color.RGBA{1, 1, 1, 255}
	for {
		display.ClearBuffer()
		temp, _ := thermo.ReadTemperature()
		stringTemp := fmt.Sprintf("%.2f C", float32(temp)/1000.0)
		tinyfont.WriteLineRotated(&display, &freemono.Bold9pt7b, 15, 20,
		↪  []byte(stringTemp), white, tinyfont.NO_ROTATION)
		display.Display()
		time.Sleep(time.Millisecond * 1000)
	}
}
```

Listing 5.11: Test program for the I^2C interface. Read periodical temperature from the thermometer and prints it to the display.

ALERT this pin is used by the device to trigger an interrupt if the alarm feature of the chip is used and the temperature reached the configured threshold.

SCL this is the I^2C clock input pin.

SDA this is the I^2C data input/output pin.

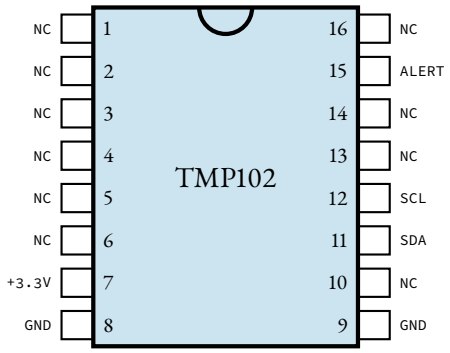


Figure 5.6: TMP102 pinout diagram.

The driver is pretty straightforward and exposes three functions to the user. The first function, `New`, creates a new display device and maps it to the given I^2C controller. The second function `Configure` is used to configure the device which in our case consists of specifying the display's I^2C address. The last function `ReadTemperature` is the one that actually makes the I^2C transaction and converts the received data to a temperature value.

The temperature register on the TMP102 controller is configured as a 12 bit register. Therefore, to read its value the driver has to read two bytes of data from the device. The complete driver code can be found in Appendix B.

5.3 Serial Peripheral Interface (SPI)

The second feature we analyzed is the serial peripheral interface (SPI) in the TinyGo compiler. The interface was indicated to be not fully supported on the HiFive1 Rev B according to the TinyGo documentation but turned out to be already working. Thus we didn't implement the SPI feature itself but wrote a driver for an OLED display.

The HiFive1 Rev B has three available SPI controllers. However, one of those (`QSPI0`) is reserved for flashing the board. All three controllers support Quad-SPI with four data lines. However, we are only going to focus on standard SPI with both MISO and MOSI lines. A short introduction to the SPI protocol is given in the next section.

5.3.1 Protocol Design

The SPI is a synchronous communication interface. It is found in almost every embedded system for communication between peripherals. The bus defines a clock line (SCLK), two unidirectional lines to read/write from/to slaves (MOSI/MISO) and, for each slave, a chip select (CS) line to activate the slave. Like in I^2C, there are two possible modes for an SPI device:

Master device the master device generates the clock and chooses the device with which it wants to communicate.

Slave device the slave device waits to be addressed by the master in order to respond.

The typical block diagram for an SPI bus is shown on Figure 5.7. Unlike with I^2C, slaves do not need unique addressing. The master can select each slave using its dedicated chip select line.

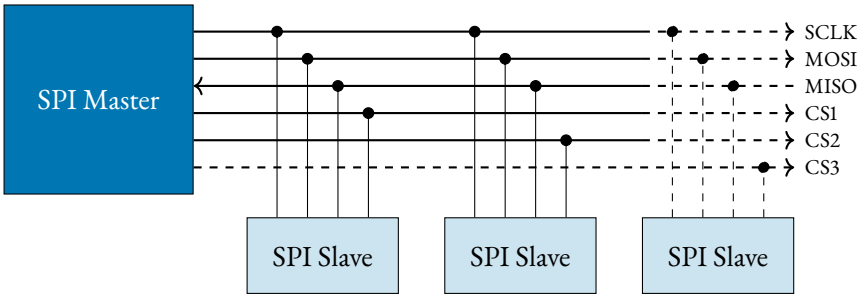


Figure 5.7: Typical SPI block diagram with one master and 3 slaves. Every device shares the common SCLK, MOSI and MISO lines. Each slave also has its own CS line.

An SPI transaction has no fixed length. The data transmission can have an arbitrary number of bits to transmit[8]. The communication between the master and the selected slave must always be full duplex even when only unidirectional data transfer is intended. An example SPI data transfer of 8 bits is shown on Figure 5.8. On this example, the chip select line is active low and data is sampled on the falling edge of the clock signal.

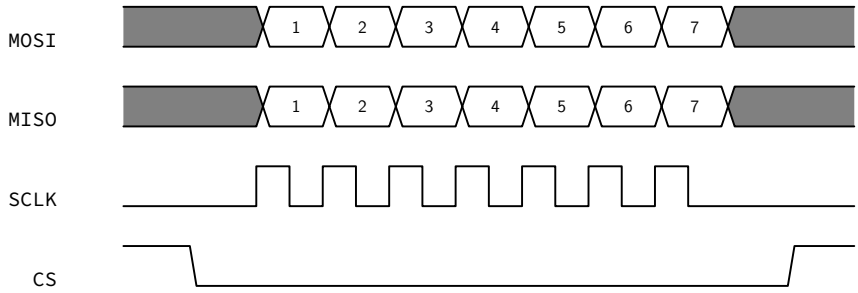


Figure 5.8: 8 bit long SPI data transfer diagram.

This resumes all there is to know about the SPI protocol for this project. The next section explains the driver that has been implemented for the SSD1351 OLED display controller.

[8] The devices will, however, be limited by their shift registers used to receive the data.

5.3.2 SSD1351 OLED Display Driver

The SSD1351 OLED Display driver supports displays of up to 128 × 128 pixels. This controller is used in the MikroElektronika OLED C Click chips. The controller communicates over SPI and has therefore been used to test the SPI interface on the HiFive1 Rev B. The pinout diagram for the SSD1351 controller is shown on Figure 5.9. The role of the different pins is the following:

R/WC this pin is only used for parallel communications which are not discussed here. In serial communication mode, the pin must be pulled low.

RST this pin is used to reset the device when it is pulled to low. During normal operations, the pin should stay high.

CS this is the SPI chip select input pin for this device. It is active low.

SCK this is the SPI clock input pin.

MISO this is the SPI MISO output pin.

MOSI this is the SPI MOSI input pin.

EN this pin activates the onboard step-up converter to provide voltage for the display. This pin should stay high.

D/C this pin is used by the device to determine if the received data is a command (D/C is low) or normal data (D/C is high).

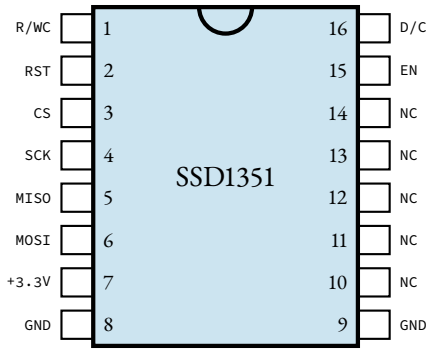


Figure 5.9: SSD1351 pinout diagram.

To configure the controller there are multiple commands defined in the datasheet [16] of the device. We have listed some of the commands used in this driver on Table 5.2. To initialize the device we have to send multiple commands. Each command which is sent can be followed by one or more bytes of configuration data for that command. The initialization code is available on Appendix C but will not be further discussed since it is not very interesting. To send the commands and the data to the device we made a little helper function shown on Listing 5.12 which automatically sets the chip select pin to low and also sets the D/C pin according to whether the data is a command or not.

Our driver allows the user to draw unique pixels, lines or rectangles. Before being able to send the pixel color data we have to configure the devices memory according to the area we want to modify in RAM. This is exactly what the `setWindow` function does. The code for this function can be found on Listing 5.13. The remaining code of the driver, which is elementary, can be found in the pull request #146 in the TinyGo Driver GitHub repository[9].

```
// Tx sends data to the display
func (d *Device) Tx(data []byte, isCommand bool) {
        d.dcPin.Set(!isCommand)
        d.csPin.Low()
        d.bus.Tx(data, nil) // Send data on the SPI bus.
        d.csPin.High()
}
```

Listing 5.12: Function used to send commands or data to the SSD1351 controller.

```
// setWindow prepares the screen memory to be modified at given coordinates
func (d *Device) setWindow(x, y, w, h int16) {
    x += d.columnOffset
    y += d.rowOffset
    d.Command(SET_COLUMN_ADDRESS)
    d.Tx([]byte{uint8(x), uint8(x + w - 1)}, false)
    d.Command(SET_ROW_ADDRESS)
    d.Tx([]byte{uint8(y), uint8(y + h - 1)}, false)
    d.Command(WRITE_RAM)
}
```

Listing 5.13: Function used to set the area in RAM which is going to be written to on the SSD1351 controller.

Lastly, the controller uses the 16 bit RGB565 color representation. Since 16 bits cannot be divided by 3, there is one additional bit for the green component[10]. To convert a `color.RGBA` Go type to a `int16` representing an RGB565 color, we take the upper 5 or 6 bits of the color component and add them together using some shifting operators. The actual function implemented to do this is shown on Listing 5.14.

To finish off the section dedicated to the SSD1351 driver, we also added a code example that shows the driver in use on Listing 5.15. Not every aspect of the driver code has been detailed but, as already mentioned, the remaining code is straightforward and can be found on the TinyGo GitHub repository.

[9] https://github.com/tinygo-org/drivers/pull/146, visited on 05/07/2020.

[10] The human eye is most sensitive to the green color, thus it makes sense to spend the extra bit for the green component. https://www.nde-ed.org/EducationResources/CommunityCollege/PenetrantTest/Introduction/lightresponse.htm, visited on 05/06/2020.

Command	Description	Command code
Set Command Lock	Allows the device to be locked to ignore new commands.	`0xFD`
Set Sleep Mode ON	Turns off the display.	`0xAE`
Set Sleep Mode OFF	Turns on the display.	`0xAF`
Set Front Clock Divider	Sets the oscillator frequency and clock divider.	`0xB3`
Set MUX Ratio	Sets the multiplexer ratio.	`0xCA`
Set Re-map/Color Depth	Sets re-map and dual COM line mode.	`0xA0`
Set Column Address	Sets the start and end column memory positions.	`0x15`
Set Row Address	Sets the start and end row memory positions.	`0x75`
Set Display Start Line	Sets the display start line register.	`0xA1`
Set Display Offset	Sets the mapping of the display's start line.	`0xA2`
Set GPIO	Sets the states of GPIO0 and GPIO1.	`0xB5`
Set Function Selection	Sets the voltage regulator.	`0xAB`
Set Phase Length	Sets the phase of the waveform.	`0xB1`
Set Contrast	Sets the contrast for colors A, B and C.	`0xC1`

Table 5.2: SSD1351 commands.

```
1  // RGBATo565 converts a color.RGBA to uint16 used in the display
2  func RGBATo565(c color.RGBA) uint16 {
3      r, g, b, _ := c.RGBA()
4      return uint16((r & 0xF800) +
5          ((g & 0xFC00) >> 5) +
6          ((b & 0xF800) >> 11))
7  }
```

Listing 5.14: Convert a `color.RGBA` type to a RGB565 color.

```go
package ssd1351

import (
    "machine"

    "image/color"

    "tinygo.org/x/drivers/ssd1351"
)

func main() {
    machine.SPI1.Configure(machine.SPIConfig{
        Frequency: 2000000,
    })
    display := ssd1351.New(machine.SPI1, machine.D18, machine.D17, machine.D16,
    ↪  machine.D4, machine.D19)

    display.Configure(ssd1351.Config{
        Width:        96,
        Height:       96,
        ColumnOffset: 16,
    })

    width, height := display.Size()

    white := color.RGBA{255, 255, 255, 255}
    red := color.RGBA{255, 0, 0, 255}
    blue := color.RGBA{0, 0, 255, 255}
    green := color.RGBA{0, 255, 0, 255}

    display.FillRectangle(0, 0, width, height/4, white)
    display.FillRectangle(0, height/4, width, height/4, red)
    display.FillRectangle(0, height/2, width, height/4, green)
    display.FillRectangle(0, 3*height/4, width, height/4, blue)

    display.Display()
}
```

Listing 5.15: Code example using the SSD1351 OLED display driver.

6 Conclusion

While some people are skeptical about the success or the usefulness of the RISC-V instruction set architecture, there is no doubt that it brings some interesting design ideas. While it is not the only open-source ISA, RISC-V has the advantage of being decoupled in many different extensions, each one focusing on a very specific set of functionalities. The fact that it allows users to develop new extensions is also promising. RISC-V International now counts more than 500 industry and academic members innovating and contributing to the ecosystem. RISC-V is still in the beginning of its development and its features — excepted from the parts of the ISA who are frozen — will continue to evolve, offering diversity and maintaining competition in the world of processors.

In this project we have demonstrated that the use of a modern language to program bare metal applications for RISC-V embedded systems is possible. There are multiple challenges which would not necessarily be present if using a "traditional" language like C, that the TinyGo developers had to solve. It is very interesting to see how a modern compiler framework and other Go tools from the standard library can be used to write a whole new compiler. The contribution to an open-source project is also a very good way to gain experience during which the help of the TinyGo community has been imperative.

▲

Personal conclusion. On a more personal note, I have really enjoyed working on this project. It allowed me to learn some basics about compiler design and apply it to another field of computer science which I am passionate about, namely embedded systems. I wish to thank both of my supervisors, Mr. Gachet and Mr. Supcik, for their trust and the freedom they gave me for this project.

A I²C Tx Function

```
// Tx does a single I2C transaction at the specified address.
// It clocks out the given address, writes the bytes in w, reads back len(r)
// bytes and stores them in r, and generates a stop condition on the bus.
func (i2c I2C) Tx(addr uint16, w, r []byte) error {
        var err error
        if len(w) != 0 {
                // send start/address for write
                i2c.sendAddress(addr, true)

                // ACK received (0: ACK, 1: NACK)
                if i2c.Bus.CR_SR.HasBits(sifive.I2C_SR_RX_ACK) {
                        return errI2CAckExpected
                }

                // write data
                for _, b := range w {
                        err = i2c.writeByte(b)
                        if err != nil {
                                return err
                        }
                }
        }
        if len(r) != 0 {
                // send start/address for read
                i2c.sendAddress(addr, false)

                // ACK received (0: ACK, 1: NACK)
                if i2c.Bus.CR_SR.HasBits(sifive.I2C_SR_RX_ACK) {
                        return errI2CAckExpected
                }

                // read first byte
                r[0] = i2c.readByte()
                for i := 1; i < len(r); i++ {
                        // send an ACK
                        i2c.Bus.CR_SR.Set(^uint32(sifive.I2C_CR_ACK))
```

```
37
38                          // read data and send the ACK
39                          r[i] = i2c.readByte()
40                  }
41
42                  // send NACK to end transmission
43                  i2c.Bus.CR_SR.Set(sifive.I2C_CR_ACK)
44          }
45
46          // generate stop condition
47          i2c.Bus.CR_SR.Set(sifive.I2C_CR_STO)
48          return nil
49  }
50
51  // Writes a single byte to the I2C bus.
52  func (i2c I2C) writeByte(data byte) error {
53          // Send data byte
54          i2c.Bus.TXR_RXR.Set(uint32(data))
55
56          i2c.Bus.CR_SR.Set(sifive.I2C_CR_WR)
57
58          // wait until transmission complete
59          for i2c.Bus.CR_SR.HasBits(sifive.I2C_SR_TIP) {
60          }
61
62          // ACK received (0: ACK, 1: NACK)
63          if i2c.Bus.CR_SR.HasBits(sifive.I2C_SR_RX_ACK) {
64                  return errI2CAckExpected
65          }
66
67          return nil
68  }
69
70  // Reads a single byte from the I2C bus.
71  func (i2c I2C) readByte() byte {
72          i2c.Bus.CR_SR.Set(sifive.I2C_CR_RD)
73
74          // wait until transmission complete
75          for i2c.Bus.CR_SR.HasBits(sifive.I2C_SR_TIP) {
76          }
77
78          return byte(i2c.Bus.TXR_RXR.Get())
79  }
```

```
80
81  // Sends the address and start signal.
82  func (i2c I2C) sendAddress(address uint16, write bool) error {
83          data := (address << 1)
84          if !write {
85                  data |= 1 // set read flag in transmit register
86          }
87
88          // write address to transmit register
89          i2c.Bus.TXR_RXR.Set(uint32(data))
90
91          // generate start condition
92          i2c.Bus.CR_SR.Set((sifive.I2C_CR_STA | sifive.I2C_CR_WR))
93
94          // wait until transmission complete
95          for i2c.Bus.CR_SR.HasBits(sifive.I2C_SR_TIP) {
96          }
97
98          return nil
99  }
```

B TMP102 Driver Code

```go
// Package tmp102 implements a driver for the TMP102 digital temperature sensor.
//
// Datasheet: https://download.mikroe.com/documents/datasheets/tmp102-data-sheet.pdf

package tmp102 // import "tinygo.org/x/drivers/tmp102"

import (
	"machine"
)

// Device holds the already configured I2C bus and the address of the sensor.
type Device struct {
	bus     machine.I2C
	address uint8
}

// Config is the configuration for the TMP102.
type Config struct {
	Address uint8
}

// New creates a new TMP102 connection. The I2C bus must already be configured.
func New(bus machine.I2C) Device {
	return Device{
		bus: bus,
	}
}

// Configure initializes the sensor with the given parameters.
func (d *Device) Configure(cfg Config) {
	if cfg.Address == 0 {
		cfg.Address = Address
	}

	d.address = cfg.Address
}
```

```
37
38  // Reads the temperature from the sensor and returns it in celsius milli degrees
    ↪  (°C/1000).
39  func (d *Device) ReadTemperature() (temperature int32, err error) {
40
41          tmpData := make([]byte, 2)
42
43          err = d.bus.ReadRegister(d.address, RegTemperature, tmpData)
44
45          if err != nil {
46                  return
47          }
48
49          temperatureSum := int32((int16(tmpData[0])<<8 | int16(tmpData[1])) >> 4)
50
51          if (temperatureSum & int32(1<<11)) == int32(1<<11) {
52                  temperatureSum |= int32(0xf800)
53          }
54
55          temperature = temperatureSum * 625
56
57          return temperature / 10, nil
58  }
```

C SSD1351 Driver Configure Function

```
// Configure initializes the display with default configuration
func (d *Device) Configure(cfg Config) {
	if cfg.Width == 0 {
		cfg.Width = 128
	}

	if cfg.Height == 0 {
		cfg.Height = 128
	}

	d.width = cfg.Width
	d.height = cfg.Height
	d.rowOffset = cfg.RowOffset
	d.columnOffset = cfg.ColumnOffset

	d.bufferLength = d.width
	if d.height > d.width {
		d.bufferLength = d.height
	}

	// configure GPIO pins
	d.dcPin.Configure(machine.PinConfig{Mode: machine.PinOutput})
	d.resetPin.Configure(machine.PinConfig{Mode: machine.PinOutput})
	d.csPin.Configure(machine.PinConfig{Mode: machine.PinOutput})
	d.enPin.Configure(machine.PinConfig{Mode: machine.PinOutput})
	d.rwPin.Configure(machine.PinConfig{Mode: machine.PinOutput})

	// reset the device
	d.resetPin.High()
	time.Sleep(100 * time.Millisecond)
	d.resetPin.Low()
	time.Sleep(100 * time.Millisecond)
	d.resetPin.High()
	time.Sleep(200 * time.Millisecond)

	d.rwPin.Low()
```

```
37          d.dcPin.Low()
38          d.enPin.High()
39
40          // Initialization
41          d.Command(SET_COMMAND_LOCK)
42          d.Data(0x12)
43          d.Command(SET_COMMAND_LOCK)
44          d.Data(0xB1)
45          d.Command(SLEEP_MODE_DISPLAY_OFF)
46          d.Command(SET_FRONT_CLOCK_DIV)
47          d.Data(0xF1)
48          d.Command(SET_MUX_RATIO)
49          d.Data(0x7F)
50          d.Command(SET_REMAP_COLORDEPTH)
51          d.Data(0x72)
52          d.Command(SET_COLUMN_ADDRESS)
53          d.Data(0x00)
54          d.Data(0x7F)
55          d.Command(SET_ROW_ADDRESS)
56          d.Data(0x00)
57          d.Data(0x7F)
58          d.Command(SET_DISPLAY_START_LINE)
59          d.Data(0x00)
60          d.Command(SET_DISPLAY_OFFSET)
61          d.Data(0x00)
62          d.Command(SET_GPIO)
63          d.Data(0x00)
64          d.Command(FUNCTION_SELECTION)
65          d.Data(0x01)
66          d.Command(SET_PHASE_PERIOD)
67          d.Data(0x32)
68          d.Command(SET_SEGMENT_LOW_VOLTAGE)
69          d.Data(0xA0)
70          d.Data(0xB5)
71          d.Data(0x55)
72          d.Command(SET_PRECHARGE_VOLTAGE)
73          d.Data(0x17)
74          d.Command(SET_VCOMH_VOLTAGE)
75          d.Data(0x05)
76          d.Command(SET_CONTRAST)
77          d.Data(0xC8)
78          d.Data(0x80)
79          d.Data(0xC8)
```

```
80          d.Command(MASTER_CONTRAST)
81          d.Data(0x0F)
82          d.Command(SET_SECOND_PRECHARGE_PERIOD)
83          d.Data(0x01)
84          d.Command(SET_DISPLAY_MODE_RESET)
85          d.Command(SLEEP_MODE_DISPLAY_ON)
86  }
```

Acronyms

ABI	application binary interface
ALU	arithmetic logic unit
API	application programming interface
AST	abstract syntax tree
CFG	control flow graph
CISC	complex instruction set computer
EBNF	extended Backus–Naur form
GC	garbage collector
I^2C	inter-integrated circuit
IoT	internet of things
IR	intermediate representation
ISA	instruction set architecture
RISC	reduced instruction set computer
SPI	serial peripheral interface
SSA	static single-assignment

Bibliography

1. M. Anastos. *Sparse Conditional Constant Propagation*. URL: https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/sccp/ (visited on 04/26/2020).
2. M. Braun and S. Hack. "Register Spilling and Live-Range Splitting for SSA-Form Programs". In: *Compiler Construction*. Ed. by O. de Moor and M. I. Schwartzbach. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 174–189. ISBN: 978-3-642-00722-4.
3. S. Chattopadhyay. *Embedded System Design*. Prentice-Hall Of India Pvt. Limited, 2010. ISBN: 9788120340244. URL: https://books.google.ch/books?id=0BQ7WkOa_28C.
4. W. contributors. *Application binary interface — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Application_binary_interface (visited on 04/28/2020).
5. W. contributors. *History of general-purpose CPUs — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/History_of_general-purpose_CPUs (visited on 03/23/2020).
6. W. contributors. *Instruction set architecture — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Instruction_set_architecture (visited on 03/26/2020).
7. W. contributors. *RISC-V — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/RISC-V (visited on 04/28/2020).
8. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". *ACM Trans. Program. Lang. Syst.* 13:4, 1991, 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: https://doi.org/10.1145/115372.115320.
9. A. DeLeo. *RISC-V Foundation Announces Ratification of the RISC-V Base ISA and Privileged Architecture Specifications*. 2019. URL: https://riscv.org/2019/07/risc-v-foundation-announces-ratification-of-the-risc-v-base-isa-and-privileged-architecture-specifications/ (visited on 04/27/2020).
10. R. Herveille. *I^2C-Master Core Specification*. 0.9. OpenCores, 2003.
11. *HiFive1 Rev B – Crowd Supply*. URL: https://www.crowdsupply.com/sifive/hifive1-rev-b (visited on 02/28/2020).
12. J. Ledin. *Modern Computer Architecture and Organization: Learn x86, ARM, and RISC-V architectures and the design of smartphones, PCs, and cloud servers*. Packt Publishing, 2020. ISBN: 9781838987107. URL: https://books.google.ch/books?id=eCLhDwAAQBAJ.
13. *MAix Bit for RISC-V AI+IoT – Seed Studio*. URL: https://www.seeedstudio.com/Sipeed-MAix-BiT-for-RISC-V-AI-IoT-p-2872.html (visited on 02/28/2020).
14. *MicroPython official website*. URL: https://micropython.org/ (visited on 03/31/2020).

15. T. A. Mogensen. *Basics of Compiler Design*. Copenhagen, Denmark, 2010.
16. *SSD1351 – Advance Information*. Solomon Systech Limited, 2009. URL: https://download.mikroe.com/documents/datasheets/ssd1351-revision-1.3.pdf.
17. *TinyGo pipeline*. URL: https://tinygo.org/compiler-internals/pipeline (visited on 04/07/2020).
18. A. Waterman and K. Asanović, eds. *The RISC-V Instruction Set Manual*. Vol. 1. 2019. URL: https://riscv.org/specifications/isa-spec-pdf/.

Declaration of originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. I certify that I have not used plagiarism or any other form of fraud. All sources of information used and author quotes have been clearly mentioned.

Yannis Huber