Extension of the Kotlin/Native compiler to support RISC-V targets

Anton Saatze

A thesis presented for the degree of
Master of Science

Computer Science and Mathematics
HM Hochschule München University of Applied Sciences
Germany

Abstract

This thesis describes how the Kotlin/Native compiler can be extended to enable compilation of Kotlin for the RISC-V architecture. For this purpose, changes in the JetBrains/Kotlin repository are presented and dependencies are analyzed. Strategies to resolve the dependencies and upcoming conflicts are discussed and pursued. Since the Kotlin/Native compiler relies on the LLVM toolchain, one focus of the work is the analysis and adaptation of the integration of the LLVM distribution and the necessary changes that go with it. This research was carried out to make the changes easier to implement later and to show the extent of the changes.

Besides that, theoretical principles are presented to provide a better understanding of the required changes. This includes details about the Kotlin/Native compiler and how it works during compilation and an introduction to the LLVM toolchain and the LLVM Intermediate Representation with a focus on upcoming changes. Moreover, a summary of RISC-V features and basics on the RISC-V ISA are explained to illustrate the benefit of using LLVM.

The result of this work is a fork of the Kotlin repository that makes it possible to build the Kotlin/Native compiler so that it can cross-compile Kotlin source code for RISC-V boards. This requires changes in the Kotlin/Native runtime, the use of a pre-built riscv-gnu-toolchain and the use of a more recent LLVM version. Additionally, an attempt to upgrade the LLVM version in the Kotlin project and upcoming problems are described.

Finally, the outcome of this work is compared with research that targets similar objectives to port languages onto the RISC-V architecture.

Contents

1 Introduction

1.1 Context and background

Kotlin replaced Java as the default language in most Android projects years ago. A JetBrains study from 2022 shows that a majority of respondents had used Java as their main programming language before switching to Kotlin. The main development areas include mobile, web backend and framework development. Besides mobile development in Android and Java Virtual Machine-based web backend development, the language can be used for JavaScript and native applications too. [1] [2]

The basis for this are various Kotlin compiler, which aim to compile Kotlin source code for their specific target. The primary targets are JVM, JavaScript and native binaries. The Kotlin/Native compiler is used for the latter. It enables compiling Kotlin source code for platforms without virtual machines, like embedded targets or iOS devices. The number of operating systems is limited according to the support of the archictecture-vendor-system-abi combination. Not every combination of CPU architecture and operating system is supported, according to the JetBrains team that takes care of this. In addition to extensive macOS support, there is also support for Linux, as long as it runs on an x86_64 CPU or an aarch64 CPU. There is no support for Linux on a RISC-V CPU yet. [3] [4]

RISC-V is also an up-and-coming CPU with many advantages. In addition to license freedom, various extensions to the RISC-V Instruction Set Architecture are being developed to improve the performance of embedded systems and special use cases. [5]

1.2 Motivation

Supporting RISC-V as a CPU target would ensure competitiveness with other programming languages. Languages which support native development such as Rust or Go already support the RISC-V hardware and started with experimental support back in 2020. Moreover, LLVM started support for RISC-V in LLVM version `9.0.0`. [6] [7] [8] [9]

Porting Kotlin to RISC-V targets could enable the Kotlin community to get started with native development. Although there are already ports for Arm CPUs, a larger selection of boards could potentially expand and motivate the community. According to the JetBrains study from 2022 only 2% use Kotlin for IoT projects and 5% for systems programming. The porting of Kotlin to RISC-V could also be a sign that native system development using Kotlin/Native will continue to be pursued in the future. [1]

If Kotlin code could also run on more embedded systems, companies focusing on the IoT industry could also benefit. If an embedded system and an associated mobile application are developed in Kotlin, it may be possible to create a common code base so that code does not need to be developed multiple times and bugs can be more easily identified and resolved. By sharing an under-

standing of syntax, architecture and best practices, it allows more developers to understand, maintain and extend the code.

It can be noted that the support of RISC-V targets is already a commonly requested feature in the community. [10]

1.3 Objectives and research questions

All Kotlin compiler are developed by JetBrains under the Apache 2 license (Version 2.0) and all code in the Kotlin repository has their copyright. Nevertheless, it is a public codebase and the community is invited to participate. The repository can be found at `https://github.com/JetBrains/kotlin`. [11]

For further participation, it would be beneficial if there was more documentation on porting the Kotlin/Native compiler and on working with the compiler in general. Therefore, one objective of this thesis is to summarize the experience of adapting the Kotlin/Native compiler to support the RISC-V platform. This includes presenting problems that can arise.

The second objective is to work out changes so that they can be adopted by the JetBrains team. To achieve this objective, it is divided into four sub-goals. The current structure and functionality of the Kotlin/Native Compiler need to be analyzed and presented. This is tackled in Chapter 2. The present limitations and compatibility issues have to be identified, and a resolution strategy has to be devised. This is covered in Chapter 3. The fourth step is to implement a customization of the Kotlin repository using the chosen strategy to extend the functionality of the Kotlin/Native compiler. Implementation details are presented in Chapter 4.

To validate the produced executables and therefore the implemented changes, a test environment is set up in Chapter 5. In order to provide an insight into other current research, some research projects are presented in Chapter 6. Eventually, the introduced changes are evaluated, and this work is summarized in Chapter 7.

All named and referenced files are also listed in Table 6 in Appendix A. Short error messages and code changes as command line invocations are placed within this work. Larger sections are moved to the appendix. For a better understanding of how the Kotlin/Native compiler works, code examples are included in Appendix B. With the same intention, linker and clang invocations that are executed by the compiler are added to Appendix C. Errors that occurred during the implementation are added to Appendix D. All changes that are required to enable RISC-V as a new target in the Kotlin/Native compiler are supported by examples and code changes from Appendix E. Changes that are required to bump the LLVM version within the repository are extracted to Appendix F. Files that were analyzed to examine issues and problems are added to Appendix G.

> Kotlin/Native is a LLVM backend for the Kotlin compiler, Runtime implementation, and native code generation facility using the LLVM toolchain.
>
> Kotlin/Native is primarily designed to allow compilation for platforms where virtual machines are not desirable or possible (such as iOS or embedded targets), or where a developer is willing to produce a reasonably-sized self-contained program without the need to ship an additional execution runtime.

Listing 1: Excerpt from the Kotlin/Native README[12]

Figure 1: Compilation pipeline

2 Fundamentals

In the repository, the Kotlin/Native compiler is official described as quoted in listing 1. The compiler makes use of the LLVM distribution. Several Intermediate Representation will be compared, edited, combined and dependencies resolved during the compilation. As a result, a final Intermediate Representation is created and will be linked with the c-runtime, which is provided by a platform depending cross-compile-toolchain. The process is visualized in Figure 1.

As explained in Chapter 1.3, one of the objectives is to expand the range of potential target platforms to include the RISC-V architecture.

Consequently, this chapter is dedicated to a detailed examination of the special features of the LLVM Intermediate Representation, how the compiler works during compilation and the processes for generating the compiler. In addition, the RISC-V architecture is introduced to provide a base for the following chapters and to illustrate which details are covered by the LLVM toolchain.

Since the Kotlin/Native Intermediate Representation is very similar to the LLVM IR, and the LLVM IR is ultimately also used, Chapter 2.1 introduces concepts of the LLVM compiler infrastructure. This includes the generation of LLVM IR commands, the use of passes to optimize the IR and the change from fixed pointer types to opaque pointer types. Chapter 2.2 describes the individual steps involved in compiling a Kotlin source file in more detail, and thus describe the pipeline from Figure 1 in detail. The RISC-V ISA is introduced in Chapter 2.3. The contents of a cross-compile-toolchain are summarized in Chapter 2.4.

2.1 LLVM Intermediate Representation

LLVM is a Static Single Assignment (SSA) based representation which can be used to translate high-level languages into hardware-specific assembly code and perform various optimizations. These can be both target independent and dependent. LLVM itself aims to be low-level and makes use of local and global identifiers, which are used to define variables, types and functions. Together with symbol table entries, they are defined within a module. [13]

2.1.1 Codegen

Any Language can be designed to map its syntax into the LLVM SSA form. For example, C, C++ and objective-c can be mapped to a LLVM Intermediate Representation through the compiler frontend clang.

This enables the Kotlin/Native compiler to provide C++ code for the native runtime, which can be compiled into a Intermediate Representation and saved as bitcode file. LLVM also offers a number of options for modifying the Intermediate Representation in the form of classes and functions. The LLVM example project shows, among other things, how existing Intermediate Representations can be read and modified. For example, by adding variables or functions. To do this, the code generator creates a LLVM context with which it extends the active module using an IRBuilder.

The options offered for modifying the LLVM IR are provided by the LLVM-C library. This can be integrated by C or C++ programs and thus call the classes and functions presented. The Kotlin/Native compiler also implements a code generator that makes use of the LLVM-C library. This is presented in more detail in Chapter 2.2. [14] [15] [16]

Besides that, LLVM provides further tools, like the llvm-dis tool, which can be used to convert a bitcode file into its human-readable IR. The llvm-readelf and llvm-objdump tools can be used to analyze object files and to obtain low-level information about them. [17] [18] [19]

2.1.2 Pointer types

The pointer type within the LLVM Intermediate Representation has changed with the LLVM version `16`. Previously, pointers had an explicit pointer type. Since version `14`, it has also been possible to use opaque pointers, but this must be taken into account in the Intermediate Representation. An IR that uses opaque pointers must also be modified and read with functions that are intended for opaque pointers. The LLVM project provides a guide on what needs to be taken into account when converting from explicit pointer types to opaque pointer types.

Command line invocation 1.

```
clang -S -emit-llvm filename.c
```

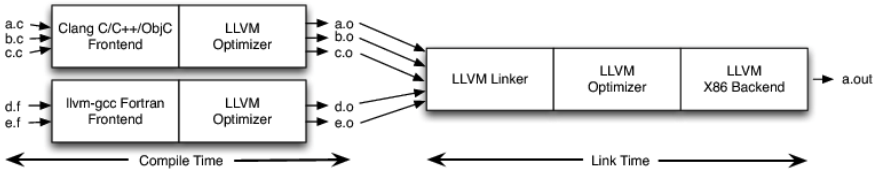

Figure 2: Link time optimization [14]

Comparing Listing 82 and Listing 83 of Appendix G.1 visualizes the change of the LLVM IR. In clang `11.1.0` the pointer type is set to i32, followed by * to indicate that it is a pointer. In clang version `17.0.5` the pointer types are opaque and set to `ptr`. Both listings can be generated by the command line invocation command line invocation 1.

Explicit pointers contain their pointer type. If the pointer type does not match the expected type, bitcasts are required to specify the type. Ultimately, this also results in many redundant no-op bitcasts in the IR. These take up memory space, delay the compile time and complicate the code.

However, as many instructions do not take the pointer type into account and the advantages of opaque pointers outweigh the disadvantages, explicit pointer types are replaced by opaque pointer types.

Opaque pointers all have the same pointer type `ptr`. This can be used to determine the true pointer type using additional function calls. This makes bitcasts redundant and simplifies the implementation of backends. The LLVM-C library evolves, and both variants can be used, with the goal to remove the deprecated fixed pointer type in the long run. [20]

2.1.3 Optimizations and passes

One advantage of the LLVM IR and the LLVM framework is the use of passes. Passes are used to analyze and modify the IR. Existing passes can be used, or own passes can be written. These can be platform-independent. Examples of frequently used optimizations are dead code elimination in relation to various symbols such as global variables, unused arguments, unused functions or basic blocks that cannot be reached. Other examples are loop optimizations, inlining of functions or the simplification of the Control Flow Graph (CFG). To analyze an IR, analysis passes such as dependency analysis or lint checks are used. In addition, some analysis passes are used for debugging purposes. For example, a CFG can be saved as a `dot` file or information on a module, loops or functions can be output. It is up to the user of LLVM or the implemented compiler to decide which passes should be executed and in which order they should be executed. With the help of the PassManager class, it can add and execute the passes. It should be noted that the legacy PassManagerBuilder can no longer be used since LLVM version `17`. This has been considered deprecated since version

13 and is to be replaced by the new pass manager. Instructions on how to use the new Pass Manager are provided in the LLVM documentation. [14] [21] [22] [23] [24] [25] [26]

Another optimization is Link Time Optimization (LTO), which makes it possible to carry out optimizations across aggregated modules. In this way, for example, further dead code eliminations and inline optimizations can be carried out across several files. To make use of the LTO, the libLTO library of LLVM can be used. Figure 2 shows, how LLVM Optimizations can be applied during compile time on single files or modules and how multiple modules can be optimized together during linking time. [14]

2.2 Kotlin/Native compilation

To compile one or multiple Kotlin source files to a native executable application or a dynamic or static library, the Kotlin/Native compiler Konan is used. It is a Java Virtual Machine (JVM) application, which applies several steps to convert the source code into a target specific binary. As in common compilers, the file gets processed by a frontend first. It is represented as an Abstract Syntax Tree (AST). For any Kotlin compiler, this is a Kotlin specific Program Structure Interface (PSI) tree. It will be converted to a Kotlin specific IR afterward. A more detailed view is given in Chapter 2.2.1.

After this, the program will be merged with the native standard library stdlib, which contains all required functions for the interaction with the runtime. The Kotlin/Native runtime is a combination of Kotlin functions and function stubs which are required to invoke C-functions. These can be part of the Kotlin/Native runtime C implementation or native libraries which are added. The function and composition of the Kotlin/Native runtime is presented in more detail in Chapter 2.2.2. The native platform libraries and the C-Interop tool is explained in Chapter 2.2.3.

Eventually, the merged Kotlin specific IR will be optimized and then converted into a LLVM IR. This IR has to be merged with the native Runtime C implementation, which is given as multiple LLVM modules in form of bitcode files. The process will be covered in Chapter 2.2.4. After the final LLVM IR was created, it will be linked by the LLVM linker. This process will be covered in Chapter 2.2.5.

2.2.1 Frontend

The frontend covers the tasks of lexical analysis, syntax analysis and semantic analysis to verify the validity of source files. Kotlin itself is designed to reuse a common codebase for frontend phase tasks for each compiler. Therefore, the frontend makes use of the Program Structure Interface (PSI), which is a layer that "is responsible for parsing files and creating the syntactic and semantic code model"[27]. After the frontend phase run through, the syntax-tree is created as a PSI-tree.

It should be mentioned that the K1 frontend is used by default on the current version and is therefore presented. A newer K2 frontend is in development and was released in Beta at the end of 2023. The essential difference for this work is whether the generation of the IR takes place within the frontend (K2) or between the frontend and backend (K1). In addition, a performance improvement is pursued with the K2 frontend.[28] [29] [30]

Appendix B.2 shows a PSI-tree to the given Kotlin program in Appendix B.1.

2.2.2 Kotlin/Native runtime

The Kotlin/Native runtime consists of two parts. On the one hand, a Runtime implementation in C++, which defines and implements logic of individual functions and defines types and type sizes. All files can be found in the directory |kotlin-native/runtime— and will be compiled into target specific bitcode files. The Runtime implementation covers memory management in form of a custom garbage collection, concurrency, entry point management, error handling, type definitions and interoperability mechanism. The generated bitcode will contain function definitions that can be used by another LLVM module or functions which are added to the LLVM module.

On the other hand, a Kotlin API of the Runtime makes it possible to call these functions. For this, Kotlin's external modifier is used in combination with the `GCUnsafeCall` annotation and the native function name that should be executed. The description of the annotation states that a call to the Kotlin method calls the implementation with the specified C++ function. The source files for the Kotlin/Native runtime can be found in the subdirectory `kotlin-native/runtime` in the Kotlin repository. [31]

Gradle task 1.
:kotlin-native:runtime

The generation of the bitcode files is configured in the gradle task 1, which is executed when the compiler is build. These can be found in the target specific subfolder within the Runtime build directory. During compile time, they are determined and referenced via the Konan config based on the arguments used for the compile time of the Kotlin/Native compiler. This can be recognized by the `KonanConfig.kt` file. The Kotlin/Native stdlib is also added as "resolvedLibrary" by default to the configuration. It is possible to exclude the Kotlin/Native standard library by passing the Konan flag `-nostdlib`. [32]

In summary, this means that the Kotlin referenced stdlib can be used to invoke Runtime functions, because the associated implementation in the form of LLVM bitcode files based on a C++ implementation has been linked to the program. The required sizes for the memory allocation are in turn determined by the C++ implementation of the Runtime and the target-platform oriented compilation.

2.2.3 Platform libraries

Besides the Runtime library, the Kotlin/Native distribution comes with a collection of pre-built libraries tailored for each target platform, which allows users to utilize services native to their operating system. The possible target platforms are `android`, `ios`, `linux`, `mingw`, `osx`, `tvos` and `watchos`. Since the objective of this work is only to use Linux as a platform on RISC-V targets, the consideration of the Linux libraries is sufficient.

To provide the libraries, the C-Interop tool is used. It analyses C headers and generates Kotlin bindings in the form of types, functions, and constants. The header files as well as possible additional compiler and linker arguments, excluded functions and other configuration options must be described in a `.def` file. Depending on the configuration, the library is added as a dynamic or static library and can be used via the generated Kotlin bindings.

For the Linux platform, the five definition files `zlib.def`, `posix.def`, `linux.def`, `iconv.def` and `builtin.def` are provided. The interoperability tool will create a klib file for each, which will be linked and eventually used to access the native library in the compiled library or executable. [33] [34] [35]

Similar to the Kotlin/Native standard library, it can be excluded by using the `-no-default-libs` Konan flag.

2.2.4 Kotlin Intermediate Representation linkage

The by the frontend generated PSI-tree will be transformed into a Kotlin specific IR-tree, which is labeled as IRTree. The phase itself is called PsiToIr. It must be noted, that this IR is not the LLVM IR. Kotlin provides a similar structure as LLVM and generates IRElements like functions, classes, fields, properties and more to represent the Kotlin IRTree accordingly. A LLVM module can be compared with an IRModuleFragment, which composites multiple IRFiles. [28]

After the IRTree got resolved by the PsiToIr phase, the Kotlin backend processes the resulting IRModuleFragment. The "runBackend" implementation shows, that for the IRModuleFragment a NativeGenerationState is created. With that generation state, a LLVMContext is initialized and will be used eventually to transform the IRModuleFragment into LLVM bitcode and to load the depending Runtime bitcode files. According to the implementation, all files of the IRModuleFragment will run through several phases which are likely to the LLVM passes. This is declared as lowering. After all files have been lowered, all dependencies of the IRModuleFragment will be load and lowered as well. The Kotlin API of the Kotlin/Native runtime is one of them. Then the dependencies and the root IRModuleFragment are merged into one final IRModuleFragment. [36][37]

Eventually, the final IRModuleFragment will be compiled into a LLVM IR by executing the *Codegen* phase. A CodeGenerator is going to convert each IRElement within the IRModuleFragment to its respective IR. Each IRElement contains properties, which are required for the code generator to determine the final IR. Native types which are referenced by the Kotlin/Native runtime are

load from its LLVM module using the NativeGenerationState and its LLVM-Context. [38][39][40]

The final merged files will result in a big Kotlin and LLVM IR, hence they are not added to this work. Although the single optimized Kotlin IR of the example program of Appendix B.1 can be found in Appendix B.3.

2.2.5 LLVM Intermediate Representation linkage

The produced LLVM bitcode file will be compiled into an object file through the ObjectFilesPhase. For this, the clang frontend will be used. Then the object file is going to be linked with the C-Runtime, which is provided by the target specific cross-compile-toolchain. The LLVM linker lld is used, but can be exchanged by referencing it in the `konan.properties` file. The finally produced file can be an executable, which can be executed on the target system or a dynamic or shared library which can be linked with other programs. [39]

During the linking phase, the linker is in charge to relocate the addresses of code and data and to correct addresses after multiple files have been linked. Additionally, for RISC-V several relocation types, are designed to resolve the address calculations. Moreover, they are used to optimize the assembled code and instructions can be relaxed by pairing a relocation with the `R_RISCV_RELAX` relocation symbol. This way multiple instructions can be reduced to fewer instructions as long as certain conditions are given. The relocations are specified by the RISC-V ELF specification and are not part of RISC-V itself. RISC-V defines the Instruction Set Architecture (ISA), which is used eventually after the linker produced the final executable machine code.[41] [42]

2.2.6 LLVM-C integration in the JVM based Kotlin/Native compiler

The LLVM-C library is used to generate the LLVM IR and to reference and combine existing modules. As this is not available directly for Java, the JetBrains team implemented their own pipeline to convert the LLVM-C and Clang-C library to a JVM compatible library. Therefore, they are converted to native Kotlin libraries using the C-Interop tool.

As the Kotlin/Native compiler is a JVM application, a bridge must be created between the Java Virtual Machine and the native Kotlin library using Java Native Interfaces (JNI). Java Native Interfaces enables code written in Java, executing within a JVM, to communicate and function alongside applications and libraries developed in different programming languages, including C, C++, and assembly. [43]

Gradle task 2.
:kotlin-native:Interop:genClangInteropStubs

Gradle task 3.
:kotlin-native:Interop:Indexer:updatePrebuilt

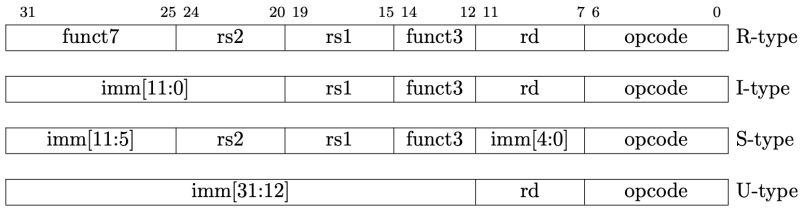

Figure 3: Base instruction formats in RISC-V [44]

To enable clang invocation, a `clang.def` file references the contents of the `clang-c/Indexer.h` and `clang-c/ext.h`. Both files will be part of the include folder, as they are configured in the linker arguments within the gradle task configuration of `kotlin-native/Interop/Indexer/build.gradle.kts`. With the gradle task 2 and gradle task 3 the generated JNI functions will be generated as `clang.kt` and `clangstubs.c` in the
`kotlin-native/Interop/Indexer/prebuild` directory.

The same is implemented for the LLVM-C library. Its configuration can be found under `kotlin-native/backend.native/llvm.def`.

Gradle task 4.
:kotlin-native:backend.native:genLlvmInteropStubs

The gradle task 4 can be used to generate the JNI LLVM stubs and native implementations. This results in a `llvm.kt` file that contains all the necessary operations and constants, as well as type definitions, to interact with the LLVM-C library from the JVM.

In both cases, all function calls are mapped to generated kniBridge functions with an incrementing number. If any of the libraries changes, the definition files must be adapted, and it must be ensured that all functions are fully mapped.

2.3 RISC-V

As mentioned in the introduction, the RISC-V architecture is not yet one of the supported platforms of the Kotlin/Native compiler. Although the feature is requested by the community. Microcontrollers with CPUs based on the x86 and arm instruction sets are already supported.

RISC-V is an Instruction Set Architecture (ISA). Just like other instruction-set architectures, it describes in an abstract way how a CPU executes instructions on the hardware. The specification of an ISA defines the available instructions, the instruction lengths, the registers and the memory as well as exception behavior.

A base integer ISA is defined for RISC-V , which must always be supported by the hardware and is indicated by the letter I. It is based on the width of the registers, the memory and the number of registers. This allows the base ISA

```
auipc ra, 0  # R_RISCV_CALL_PLT (symbol), R_RISCV_RELAX (symbol)
jalr  ra, ra, 0
```

Listing 2: Relaxation of instructions is possible if value of ra is within $\pm 1MiB$[41]

to be divided into RV32I, RV64I and RV128I. The specification defines the OP codes of the standard instructions and the format in which the information on the instructions are interpreted. For the base instruction set, each instruction fits into an assigned format with a length of 4 bytes. The formats differ in the bit length of passable immediate values, amount of usable registers and in the interpretation of the set values. Used formats are shown in Figure 3.

Instructions like the ADD Immediate (ADDI), Set Less Than Immediate (SLTI), AND Immediate (ANDI), OR Immediate (ORI) and XOR Immediate (XORI) make use of the I-type format, to perform their specified arithmetic computation with the sign-extended 12 bit value and the value of the given rs1 register. The result will be stored in the passed rd register.

The U-type format is used for the two very common instructions Load Upper Immediate (LUI) and ADD Upper Immediate to PC (AUIPC). With LUI the upper 20 bits of the passed rd register can be set. With AUIPC Program Counter (PC) relative additions can be calculated within one instruction.

If the arithmetic computation is based on two registers, the R-type format is used. Analogous to Add Immediate, there is also an ADD Instruction (ADD) that adds 2 registers and saves the result in the specified rd register. A subtraction with ADDI is possible if the 12th bit is set to 1 and the immediate is sign extended as a negative number. For the register based subtraction, the SUB instruction is specified, which subtracts the value of rs2 from the value of rs1 and writes the result into rd.

The S-type is specified for store operations where the value of rs2 is stored in memory at the address of the value of register rs1 with an offset of the sign-extended immediate value.

To change the Program Counter (PC) value, the Jump and Link (JAL) and Jump And Link Register (JALR) instructions can be used. The JAL command accepts a sign-extended immediate value of 20 bits, which is shifted left by 1. The shift enables to use the immediate value to be the upper 20 bits of a 21 bit immediate value. This value is used as an offset on the current PC and enables jumps in a range of $\pm 1MegaByte$. The value of the PC before the jump + 4 is written to rd, which enables JALR to jump to the previous PC. JALR commands make use of the I-type format and accept a 12 bit immediate value, which is sign-extended added to the value of rs1. The PC is set to the computed value with setting the least-significant bit to 0. The value of rd is set to PC before the jump + 4. [44]

If a linker is able to use relaxation, some instruction combinations can be marked as relaxable and reduced by it. For example, the linker associates any

R_RISCV_CALL_PLT relocation as a pair of AUIPC and JALR. The AUIPC instruction is marked as R_RISCV_CALL_PLT and R_RISCV_RELAX. The linkers relaxation allows relaxing both instructions into a single JAL instruction, if the given registers are in the range of $\pm 1MiB$. This relaxation is also referenced as Function Call Relaxation. In addition to this, other relaxations are also defined, which differ in terms of relocation symbol combinations and conditions. An example of this is illustrated in Listing 2. [41]

Besides the relaxation of instructions, the linker also is in charge to process R_RISCV_ALIGN relocation symbols. These are added if a location or instruction must be aligned to a given byte's size. If the R_RISCV_ALIGN symbol is processed, no operation (nop) instructions are added as padding in front of the instruction until the required alignment is filled. For the alignment, only alignments in the power of two are possible. [41]

Besides the base instruction set, there are various extensions that can be implemented by hardware manufacturers to develop customized solutions or to provide flexible general purpose solutions. The extensions include instructions for integer multiplication and division (identifier M), atomic read, modify and write instructions (identifier A), floating point support (identifier F) and double precision support (identifier D). The C extension enables compression of some standard 32-bit instructions into a 16-bit representation.

A special characteristic that distinguishes RISC-V from the architectures of other processor manufacturers is its open and license-free access. This means that companies can develop their own processors and optimize them for their own needs and are not dependent on closed source hardware from processor manufacturers such as Intel or Arm. Any software that is written for the given target description can run on the hardware implementation. [44]

Companies such as SiFive have been founded on the basis of the RISC-V ISA. SiFive develops and distributes processors and platforms based on RISC-V . The company offers a range of processor cores that are optimized for a variety of applications, from embedded systems to powerful data center servers. [45]

A compilation of LLVM IR to RV32I and RV64I is supported according the LLVM Project. The range of extensions is limited, and only selected extensions are supported yet. The previous mentioned extensions are fully supported by the latest LLVM compiler. This includes pattern matching to lower instructions by recognized idiomatic patterns. [46]

2.4 Cross-compilation

To develop RISC-V applications on a not RISC-V based host system, a cross-compiler is required. A cross-compiler enables the compilation of source code for a target system with a different architecture to that of the host system. Its toolchain includes a number of files and tools for this purpose. Each cross-compile toolchain is tailored precisely to the combination of host and target platform. Even if hardware manufacturers often provide a toolchain, individual adaptations may be required or the combination of host and target may not be offered.

Part of the toolchain are the libraries for the C library, header files, configurations, tools such as archive utilities, assemblers, compilers, linkers, debuggers and other tools for analyzing the code. The C library is an implementation of the POSIX functions and makes it possible to call various system calls.

The tools also called binutils include one or more frontends like cpp (GNU c++ frontend) and gcc (GNU C frontend). For the RISC-V gnu toolchain these are even split into bare-metal support, where no operating system is installed and systems with Linux with an installed GNU C Library. If the toolchain is targeting a Linux environment, the compiler will contain `-linux-` in its name. Otherwise, the toolchain might be named like the riscv-gnu-toolchain when it is not build for Linux, which contains `-elf-` instead of Linux.

All tools have to be compiled for the host platform, since they cannot be executed otherwise. Using a toolchain with the correct target, but wrong host platform, might result in a partial working solution. The C library and tools like the dynamic linker are target platform dependent and the header files are specified by the kernel version, which makes them still usable, even if the binutils for the host are unusable. [47] [48] [49]

This is the case in Kotlin/Native's cross-platform compilation. The cross-compiler toolchain is mixed with the LLVM toolchain. While the tools of LLVM are used to provide the compiler frontend, the required C library, header files and configurations are used by the cross-compiler. Therefore, there is no problem when a cross-compile-toolchain for the wrong host platform is used.

3 Strategies

The previous Chapter 2 has shown, how the Kotlin/Native compiler can compile source code into executable or linkable assembly code for existing targets. In the next chapters, the three most important dependencies, the LLVM distribution, the cross-compile-toolchain and the codebase of the Kotlin repository are analyzed to determine the necessary changes. These changes will be presented as the initial strategy, designed to fulfill the primary objective, which includes enabling compilation for RISC-V targets.

Chapter 3.1 provides an overview of the dependencies. It presents the versions that are currently in use, discusses potential reasons for modifications, and examines different approaches to modifying or updating the dependencies. It also discusses potential difficulties that may arise when modifying versions.

Chapter 3.2 represents the objective in a technical form and describes how the Kotlin/Native compiler should ultimately be used to create a RISC-V compatible executable. Chapter 3.3 will define the initial strategy to extend the Kotlin/Native compiler. A conflict by the interaction of the linker and the C-library arose during the implementation of the initial strategy, which is presented in Chapter 3.4. This led to a refinement of the strategy and thus to deviated strategies, which are outlined in Chapter 3.5, 3.6 and 3.7.

3.1 Dependency analysis

The Kotlin/Native compiler controls the interaction between the used LLVM distribution, the target platform cross-compiler-toolchain and the source file to be compiled. As described in Chapter 2.2, Konan provides the runtime and platform-dependent libraries and ultimately links these to the target platform. The LLVM distribution and the potentially usable cross-compiler are presented in the following chapters. Expected code changes in the Kotlin repository are also determined.

3.1.1 LLVM distribution

The Kotlin/Native compiler of tag `v2.0.0-Beta2` references a LLVM fork by apple with the version `20200714`. Their fork contains patches that have not yet been upstreamed to the LLVM project, including some special support for Swift. The downloaded distribution shows that it provides clang with version `11.1.0`. Moreover, it contains the include files for the Clang-C and LLVM-C libraries and further tools, libraries and header files.[50]

The linker that will link the final bitcode file with the C-Runtime, which is provided by the cross-compile-toolchain, is also part of the tools and can be referenced as `ld.lld` within the `bin` folder of the distribution's root directory.

If the LLVM version needs to be increased, the LLVM backend can be built using the Kotlin repository script. It is located in the `kotlin-native/tools/llvm_builder` directory.

target	unkown-linux-gnu
gcc	8.3.0
glibc	2.25
kernel	4.9-2

Table 1: Aarch64 cross-compile toolchain properties

target	unkown-linux-gnu
gcc	8.3.0
glibc	2.19
kernel	4.9-2

Table 2: X86_64 cross-compile toolchain properties

target	unkown-linux-gnu
gcc	13.2.0
glibc	2.38
kernel	6.6

Table 3: RISC-V cross-compile toolchain properties

With clang version 9.0.0 RISC-V was described as stable, and according to the official release note, offers full support for RV32I and RV64I. Therefore, it is assumed that the LLVM version does not need to be increased. [9] Extensions such as the vector, Zba, Zbb, Zbc, Zbs, Zfh and Zfhmin extensions are announced as non-experimental in version 14.0.0. If further extensions or bug fixes are required, the version of the LLVM distribution must be upgraded. [51]

Another reason to increase the LLVM distribution would be to use more up-to-date LLVM tools such as the linker or the use of newer LLVM-C functions, to guarantee interoperability with LLVM Intermediate Representation that make use of opaque pointers instead of fixed pointer (see Chapter 2.1.2).

It should be noted that changes in the entire Kotlin repository will be necessary if the version is changed. Changes to the Clang-C API, LLVM-C API or the workflow must be considered according to the functionality of the Kotlin/Native compiler as described in Chapter 2.2. Chapter 4.8 is dedicated to exploring the variety of challenges that will occur when the LLVM version is increased.

3.1.2 RISC-V cross-compile-toolchain

In the context of Kotlin/Native, the GNU Compiler Collection (GCC) is used to generate the target specific Kotlin/Native runtime. GCC as a cross-compile-toolchain provides the finally linked C-Runtime, which is referred as startup code also. It setups the C environment by specifying sections and calling the main function. The C-Runtime is linked by default and has to be disabled actively by linker flags like `-nostartfiles`, `-nostdlib`, `-nolibc` or `-nodefaultlibs`. [52]

For the Arm architecture, JetBrains provides a cross-compiler with the properties shown in Table 1. For intel architectures, a cross-compiler with properties of Table 2 is provided. Each can be build using the Kotlin repository's build script also. It is located in the `kotlin-native/tools/toolchain_builder` directory and makes use of a dockerized environment of crosstool-NG, which is an open source toolchain generator. Their documentation highlights that a toolchain is a sensitive piece of software, and it has to match the host platform, the target platform and its required optimizations, and it should be able to take different versions into account. [53] [54]

To get access to a RISC-V toolchain different approaches can be considered. Manual building provides flexibility in the configuration and asserts that the binutils will work on the host system. Although for a final integration in the JetBrains Repository, the toolchain has to be supplied somehow, therefore manual building is rather an option for experimental purpose.

On the one hand, the builder script could be extended by a configuration for RISC-V . The dockerized version of crosstool-NG offers a way that requires fewer adjustments in the host operating system. However, it is necessary to create a new crosstool configuration and there is a possibility that the entire toolchain builder tool needs to be updated, as no updates have been made for some time according to the Git history. The last time the X86_64 configuration was updated was in December 2020. The last time the Arm configuration was updated was in July 2023.[55] [56]

On the other hand, cloning the riscv-gnu-toolchain repository and initiating the build process manually is an option. It allows defining custom build options and thus influence the resulting cross-compiler toolchain. For this procedure, a setup must be carried out and the required programs, which can be found in the README file, must be installed. Chapter 5.2 shows an initial attempt on a macOS Intel host, which was not successful. [57]

In contrast, a pre-built cross-compile-toolchain can be downloaded from the RISC-V compiler gnu toolchain repository. A potential version can be the tag `2023.12.14`, which is based on the properties shown in Table 3. This can be checked from the commits of the repository and the loaded files. It has been build for Linux as the host's operating system, and its binutils cannot be used on any other operating system. As Chapter 2.4 discussed, the binutils are not used anyway. The downloaded toolchain is used for provision of the dependencies only. [57]

One disadvantage is that this option does not allow any individual customization.

3.1.3 Kotlin compiler codebase

The new target for RISC-V running on Linux has to be added to the compiler codebase. From the analysis of the compiler, it can be interpreted that the files in Table 7 in Appendix E.1 require changes. This can be concluded from the locations used for `LINUX_ARM64` and `ARM64`. Once the `LINUX_RISCV64` target has been added to the list of common targets, a build of the compiler should result

in the required platform libs and Kotlin/Native runtime being made available for RISC-V targets. Since the implementation of the runtime is written in C++ and is provided as a bitcode file compiled by the LLVM compiler, no further change in the Kotlin repository is expected to support the new target. The bitcode file should be loaded automatically by the name of the target.

However, a raise of the version of glibc, the kernel version itself and the GCC compiler will likely result in changes with the current used configuration of platformlibs like the `posix.def` configuration and in changes of the Kotlin/Native runtime. An extensive overview of the required changes is presented in Chapter 4.6.

To support RISC-V as a LLVM target option, it is possible, that the `llvm.def` file has to include additional libraries. In addition, the RISC-V targets must be initialized in the same way as the Arm targets. This is done by a generated Kotlin stub `LLVMKotlinInitializeTargets` within the `LLVM.kt` file.

Moreover, the Configurable options in the `konan.properties` have to be extended for RISC-V .

3.2 Expected result

Command line invocation 2.

```
kotlinc-native filename -target linux_riscv64
```

Command line invocation 3.

```
kotlinc-native filename -Xsave-llvm-ir-after=<PhaseName>
```

Command line invocation 4.

```
kotlinc-native filename -Xprint-bitcode 2> filename.ir
```

Command line invocation 5.

```
kotlinc-native filename -Xprint-ir > program.kexe.kir
```

If the cross-compile toolchain was added successfully and the compiler has been adjusted, it should be possible to compile Kotlin source files with the Kotlin/Native compiler against the RISC-V target. In practice this should be able by running command line invocation 2 which produces the executable `filename.kexe` file.

Furthermore, the existing functionality of the compiler should not be impaired. Accordingly, compilation for Arm, Intel and Apple-specific architectures should continue to be guaranteed.

After the RISC-V specific executable has been build, its execution can be verified by running it in an emulator like QEMU or on Hardware. Running the executable in QEMU will be the focal point of discussion in Chapter 5.

For further research, the Kotlin/Native compiler can also be used to emit the LLVM IR, or Kotlin IR to optimize its bitcode. This can be done by using

command line invocation 3 to emit the LLVM IR code after specific optimization phases.

To generate the LLVM of the final executable, command line invocation 4 can be used. It has to be noted, that the bitcode is printed out on the standard error (stderr) output. Therefore, the stderr output is written to a `filename.ir` file.

To inspect the Kotlin IR after all phases and on all processed files, command line invocation 5 can be used.

3.3 Initial strategy

In order to minimize the effort required for changes, the initial strategy is to leave the LLVM version unchanged. No changes will be necessary in the compiler with regard to the Clang-C and the LLVM-C library.

The in Chapter 3.1.2 presented option to load an already built cross-compile toolchain for RISC-V from the riscv-gnu-toolchain repository with the tag `2023.12.14` will be pursued.

As in Chapter 3.1.3 described, no changes for the Kotlin/Native runtime are expected. All files should be available after the new target was added to the common targets list.

Implementation details for this initial strategy are presented in Chapters 4.1, 4.2, 4.4 and 4.5. This will include the initial bootstrapping of the Kotlin compiler, the addition of the RISC-V target and workarounds to load custom cross-compile-toolchains and dependencies.

3.4 Deviation

The implementation of the initial strategy runs without errors up to the linking phase, where the generated target specific LLVM bitcode and the C-Runtime are linked. The linkers relocation failed, since the relocation type `R_RISCV_ALIGN` is not supported by the linker. The linker with version 11.1.0 does not yet support relaxation for RISC-V . Unlike the usage of the `R_RISCV_RELAX` type, which is ignored by the linker, the usage of the `R_RISCV_ALIGN` type is explicitly marked as throwing an error in the release notes of `lld 11.0.0`. [58] [59]

During the linking phase of the Kotlin/Native compiler, several files are linked. Appendix C.1 shows the linker invocation and the used flags. An analysis of the files shows, that the compiled C-Runtime files make use of the `R_RISCV_ALIGN` type (see Appendix G.2). It is part of the cross-compile-toolchain and can only be exchanged with a different toolchain or modified by rebuilding the cross-compile-toolchain. The usage of `R_RISCV_ALIGN` causes the error, which is shown in Listing 3. Due to this, the strategy needs to be adapted.

3.5 Deviated strategy 1: Use linker with relaxation

One option is to use a linker that supports the relaxation function. To implement this strategy, the initial strategy must be extended by referencing a new linker.

```
ld.lld: error: static-reloc.c:(.text+0x0): relocation
↪  R_RISCV_ALIGN requires unimplemented linker
↪  relaxation; recompile with -mno-relax
```

Listing 3: Error message when using LLD 11.1.0

How this can be achieved is described in Chapter 4.7

A downside of this strategy is that the linker is not provided by JetBrains as a standalone dependency. The linker is part of the downloaded LLVM dependency. This strategy can be used as a provisional solution or as a proof of concept. However, integration in the official JetBrains Kotlin repository is not possible.

3.6 Deviated strategy 2: Update LLVM dependency

The second option is to update the LLVM dependency in the Kotlin codebase. This strategy includes changes that were deliberately avoided in the initial strategy. Chapter 4.8 will go into more detail on the necessary changes with regard to Clang-C and LLVM-C and changes of LLVM general. The fundamental principles for the forthcoming changes are presented in Chapter 2.1. If all changes can be accomplished, a full integration will be possible.

3.7 Other strategies

Another option is to ensure that none of the linked components has been build with the relaxation flag. As the analysis of the C-Runtime object file shows, the pre-built cross-compile-toolchain contains files, which were build with relaxation. The initial strategy could be adapted so that a custom build cross-compile-toolchain without relaxation is used. Moreover, an older version of the toolchain could be used to be better compatible. Chapter 4.5 shows how a downloaded toolchain can be integrated. In the same way, other toolchains could be added.

The outdated LLVM version is identified as the cause of the problem in Chapter 3.4. Therefore, this problem should also be solved and not a manually customized dependency provision should be developed. As downgrading and customization do not offer a long-term maintainable and sustainable solution, these approaches will not be pursued further.

```
+local.properties: kotlin.native.enabled=true
+local.properties: kotlin.build.isObsoleteJdkOverrideEnabled=true
+local.properties: bootstrap.local=false
```

Listing 4: Initial values of local.properties

4 Implementation

The following chapters describe and comment on the implementation of the initial strategy of Chapter 3.3. Detailed steps for building the Kotlin compiler are also described here. This includes the bootstrapping with the added RISC-V target, as well as the error messages to be expected. The chapters are not sorted according to the order in the implementation, but in a way that makes sense in order to present the steps.

First, Chapter 4.1 describes how the Kotlin compiler can be compiled locally and used for further compilations. This process is called bootstrapping. Chapter 4.2 describes how the Linux-RiscV target can be added and which files need to be adapted for this. The described changes can be used as reference for any target that will be added in the future. An extensive overview of the `konan.properties` file, which needs to be updated also, is presented in Chapter 4.3. Exceptions must be added to the code to load custom dependencies that are not available on the JetBrains web server. These are described in subsection 4.4. All exceptions need to be resolved in the final integration in the official Kotlin repository, and the custom dependencies need to be provided as official dependencies by the JetBrains organization. Moreover, the new cross-compile-toolchain for RISC-V needs to be load and added, which is described in Chapter 4.5 Further unexpected changes in the Native Runtime and in the generation of platform functions are listed in Chapter 4.6.

After implementing the presented changes, the Kotlin/Native compiler is able to create a binary for the source code, but not to link it with the Runtime (see Chapter 3.4). To accomplish the objective with strategy 1 (see Chapter 3.5), Chapter 4.7 will present how to add a new linker with relaxation support. To accomplish the objective with strategy 2 (see Chapter 3.6), chapters within Chapter 4.8 will present how the current used LLVM distribution can be increased and which challenges will occur.

4.1 Bootstrap the Kotlin compiler

The instructions for building, as described in the README document, are of significant importance and utility. Nevertheless, in the absence of explicit guidance concerning the sequence in which the commands should be executed, the process remains a time-intensive endeavor to attain working outcomes.

As the README depicts all public artifacts can be installed into a local Maven repository. This will be necessary in the following steps if compiler changes are made that are required in the next build process.

```
./gradlew -Dhttp.socketTimeout=60000
↪  -Dhttp.connectionTimeout=60000
./gradlew install
sed -i -e '/<components>/,/<\/components>/d'
↪  gradle/verification-metadata.xml
./gradlew -i --write-verification-metadata sha256,md5
↪  -Pkotlin.native.enabled=true resolveDependencies
./gradlew publish
```

Listing 5: Command to build and publish artifacts to maven local

```
-local.properties: bootstrap.local=false
+local.properties: bootstrap.local=true
```

Listing 6: Changes in local.properties

Firstly, it must be ensured that the current `local.properties` file is configured accordingly the `/kotlin-native/README.md`. Before the compiler is build the first time, the `local.properties` file should contain the entries as shown in Listing 4. [12]

By invoking the commands in Listing 5 the artifacts will be build and published to the local maven repository. In subsequent builds these will be used, if the version matches the required version and bootstrapping is enabled. This version can be set in the `gradle.properties` file (compare Listing 7).

After the artifacts were published locally, bootstrapping has to be enabled in the `local.properties` file and the new bootstrapping version can be set in the `gradle.properties`. Listing 6 and 7 describe the required changes. With that, the next compilation process will make use of the earlier published artifacts.

Gradle task 5.
dist

For Kotlin/Native, the resulting compiler toolchain can be published with the gradle task 5. The toolchain will be placed in the default directory `/kotlin-native/dist`. All pre-built files as the platform libraries, the pre compiled Kotlin/Native Runtime bitcode files and the JVM based Kotlin/Native compiler Konan is within that directory. When the Kotlin/Native compiler is rebuilt at any time, it will resolve its dependencies from that directory also. If dependencies cannot be resolved, the dist directory needs to be checked, whether the dependencies are existing.

4.2 Add RISC-V as a new Konan target

The in Chapter 4.1 build compiler does not provide the `linux_riscv64` target yet. Therefore, it cannot be used by the command line compiler or by gradle

```
-bootstrap.kotlin.default.version=2.0.0-dev-9013
+bootstrap.kotlin.default.version=2.0.255-SNAPSHOT
```

Listing 7: Changes in gradle.properties

```
'when' expression must be exhaustive, add necessary 'RISCV64'
↪  branch or 'else' branch instead
```

Listing 8: Error message in gradle build script

tasks within the project. It has to be added to the Kotlin repository code base. For this, the `linux_arm64` target is used as reference. By searching for occurrences of Linux and arm64 combinations, the listed files in Table 7 (Appendix E.1) can be found in the repository. In each file, the according combination of Linux and riscv64 will be added as definition or as used reference. In Table 7 for each file hints are given for the implementation, which need to be considered when the change is applied.

Since the in `Architecture.kt` defined enum values are used from the Compiler build process itself, they can only be referenced if they were published as artifacts. Therefore, not all files can be edited at once. First, the new Architecture entry RSICV64 has to be added to the enum. Then the bootstrapping process, that is described in subsection 4.1 has to be applied. With that, the new enum entry can and must be referenced in the repository itself. For example, it is used in `kotlin-native/runtime/build.gradle.kts` gradle file. In the file, an exhaustive when statement is used, which will break the next build process if the new enum entry is not added to the when statement (compare Listing 8).

It should be noted, that depending on the properties of the new target, the use of the reference is determined in some places in the code. In `KonanTargetExtensions.kt`, a distinction is made between targets with unaligned access and aligned access. It is also determined whether the target can support 64 BitAtomics or not. Internally, this defines whether the gcc memory model aware atomic operations can be used. Which is the case, hence it is a 64 Bit architecture. This would not be the case if the RISC-V 32 bit architecture is targeted.

As RISC-V must be made accessible for future build processes in the LLVM-C library, RISC-V must also be added as a LLVM target. This is done in the `LLVMKotlinInitializeTargets` function of the `CAPIExtensions.cpp` file. The LLVM targets are initialized with each compilation so that the target triplet specified in `konan.properties` can be resolved. Appendix E.2 shows the extension of CAPIExtensions.cpp.

It is also necessary to link the required LLVM RISC-V libraries in the LLVM-C library. These must be added to the C-Interop `llvm.def` configuration for this purpose. Appendix E.3 shows the extension of the `llvm.def`

file.

Just as the RISC-V target was added, further targets can also be added. The referencing problem described should be taken into account. The properties of the new target should be known in advance in order to reference the new target correctly in the repository.

4.3 Add RISC-V configuration to Konan.properties

Besides adding the RISC-V target to the codebase, the configuration for RISC-V builds has to be updated. Appendix E.4 shows all properties that have to be added to the `konan.properties` file. It is part of the finally provided compiler within the `dist` folder and if changes are applied, gradle task 5 needs to be rerun to apply the changes in the used compiler.

Almost all properties belong to a combination of the host architecture and the target architecture. The entries for the `linux_arm64` can be used as reference to create the `linux_riscv64` entries.

In the properties file, paths for toolchains can be supplied. This is important to supply the cross-compile-toolchain which is added in Chapter 4.5.

For testing purposes, emulator dependencies and executables can be supplied. The executable is used to set up a TestRunner to verify the correctness of the compiler. Since an emulator is similar to another target, its dependencies must be specified and are different from the cross-compile information specified in the other flags.

For each target architecture a target triple has to be supplied, which is used eventually passed to clang with the `-target` flag. It should match the target triple that is given for the cross-compile toolchain. For RISC-V , this will be set to `riscv64-unknown-linux-gnu`.

The linker can also be tuned for different use cases. Flags that should be added every time can be supplied with the `linkerKonanFlags` property. It is notable that the flags set here are only an extension. In addition to these flags, the compiler sets further flags at compile time, which depend on several factors, such as whether the target is a library or an executable or whether it is being built for an Apple or a native target.

For the target sysroot configuration, the sysroot path of the cross-compiler can be set. It is passed with the `--sysroot` flag to clang during the build process.

To create a lot of flexibility, the libGcc path can also be adjusted. As for other targets, the path of the gcc within the cross-compile-toolchain is used here.

A targetCPU and additional flags can be set to align the compilation to the target in the best possible way. This is mainly used so that LLVM generates the correct platform-dependent instructions. Since RISC-V differs from other ISAs, an approach could be pursued here that behaves independently of the CPU and is only configured via the features used. The possible options for these flags are dependent on the LLVM versions. The LLVM static compiler (`llc`) can be used to print available target CPUs and target features. During this work, it turned

```
    Hard-float 'd' ABI can't be used for a target that doesn't
    ↪  support the D instruction set extension (ignoring
    ↪  target-abi)
```

Listing 9: Error message using generic-rv64 CPU

```
ld.lld: error:
↪  $TMP_PATH/konan_temp9778693601135814674/program.kexe.o:
↪  cannot link object files with different floating-point ABI
↪  from
↪  $HOME/.konan/dependencies/riscv64-glibc-ubuntu-20.04-gcc-
↪  nightly-2023.12.14-nightly/sysroot/usr/lib/crt1.o
```

Listing 10: Error message when using wrong floating-point ABI

```
+ toolchainDependency.linux_riscv64 =
↪  riscv64-glibc-ubuntu-20.04-gcc-nightly-2023.12.14-nightly
```

Listing 11: Set toolchain property for linux_riscv64 in konan.properties

out, that the available sifive-u54 CPU can be used to generate RISC-V targeting bitcode. The CPU generic-rv64 was available also, but cannot be used, since it is not supporting Hard-float ABI. The error in Listing 9 shows that. This is a requirement that comes by the used toolchain.

Similar to the linker flags also additional clang flags for different purposes can be used. As the toolchain is making use of ABI lp64d, it has to be passed to the clang compiler by the `-target-abi` flag.

Otherwise, an error shown in Listing 10 will be shown, which indicates that the provided C-Runtime is using a different floating-point ABI than the prepared bitcode file by the Kotlin/Native compiler.

The relocation mode can also be set and will be passed to the LLVM target when it gets initialized. This setting is adopted by the other targets and set to pic (Position-independent Code).

Not only linker flags can be supplied, but the used linker also. This is mandatory for the approach that is pursued in Chapter 4.7. The configuration used for the other targets points to the LLVM provided lld tool.

The dynamic linker must target the linker that is supplied in the cross-compile-toolchain.

All target specific ABI libraries and crt files are supplied by the cross-compile-toolchain too. This setting is therefore adopted by the other targets.

4.4 Enable loading custom dependencies

For the next step, the reference for the cross-compile-toolchain of the RISC-V target has to be added (compare Listing 11). It will be loaded as a dependency when the compiler is build. The DependencyDownloader is a class, that takes care of downloading all dependencies on the first run of the compiler. The default URL that is used for this is `https://download.jetbrains.com/Kotlin/Native`. When the RISC-V cross-compile-toolchain is loaded, it is loaded from the URL `https://download.jetbrains.com/Kotlin/Native/riscv64-glibc-ubuntu-20.04-gcc-nightly-2023.12.14-nightly.tar.gz`.

So far this is not supported by JetBrains, so the file cannot be loaded, and the compiler cannot be built. Therefore, a modification of the DependencyDownloader is necessary so that it does not crash if the file cannot be loaded. Once this is done, the compiler-build no longer crashes and the file can be stored manually in the dependency folder. Appendix E.5 shows the workaround implementation. [60]

This workaround will be required also if further new targets are added and the cross-compile-toolchain is not yet available.

4.5 Add new cross-compile-toolchain

As the initial strategy presented in Chapter 3.3 states, the cross-compile-toolchain should be downloaded from the riscv-gnu-toolchain repository with tag `2023.12.14`. [57] After the zipped archive has been downloaded, it must be converted into a bzip2 compressed file and saved in the directory `$KONAN_HOME/dependencies/cache`. It will be load from this directory when the compiler is build and then extracted in the `$KONAN_HOME/dependencies` directory. The name of the zip file and the referenced toolchain dependency in the `konan.properties` file has to be identical. As Chapter 4.3 states, during compilation, subdirectories will be referenced and used.

This procedure can also be applied to different toolchains for their platforms.

4.6 Runtime and platform changes

When the Runtime is build, it makes use of the include directory of LLVM `11.1.0`. Eventually the files are compiled into bitcode using the standards of GCC provided by the cross-compile-toolchain. Appendix C.2 shows one of the clang invocations that is used to generate one bitcode file of the Kotlin/Native runtime. The call shows that the gcc-toolchain of the cross-compiler is used. According to Chapter 3.1.2, it is known that the gcc-toolchain with version `13.2.0` is used for RISC-V and version `8.3.0` for Arm and Intel. The version difference leads to a problem at compile time. In the gcc-toolchain the `std:optional` has been introduced and needs to be included explicitly. The error in Appendix D.1 is shown. The problem can be resolved ab adding `#include <optional>` to the concerned `ThreadSuspension.cpp` file. The same needs to be done in `SafePoint.cpp`. [61] [62] [63]

Moreover, `std::iterator` got deprecated and with the higher gcc-toolchain version the compiler is stricter about it. [64] The error in Appendix D.2 is

shown. It can be resolved applying the changes of Appendix E.6. The change has to be applied for `checked.h` and `unchecked.h`.

When building, the compiler also prompts to change the usage of `size_t` as `std::size_t` in `common.h`.

Header reference	Deprecated since	Solution
_G_config.h	glibc 2.27	Removed
libio.h	glibc 2.27	Replaced with stdio.h
ustat.h	glibc 2.28	Removed
sys/ustat.h	glibc 2.28	Removed
sys/stropts.h	glibc 2.30	Removed
sys/sysctl.h	glibc 2.30	Removed
sys/ultrasound.h	glibc 2.26	Removed
sys/vtimes.h	glibc 2.33	Removed
sys/ioctl.h		Added

Table 4: Overview of deprecated glibc header files [65] [66] [67] [68] [69]

In addition to the jump of the GCC toolchain version, the change of the glibc version from `2.19` to `2.38` (see Table 3) also leads to errors that affect the complete provision of all required components. The def files mentioned in Chapter 2.2.3, which are required for C-Interop, configure the generation of required platform libraries. In the configuration, the Linux header files that are to be taken into account are specified. This also includes the entries from Table 4. It shows the version with which the respective files were marked as deprecated. The entries in the `linux.def` file must be adjusted accordingly. Otherwise, compilation is not possible.

A similar issue occur for entries in the posix library. In version `2.26` `xlocale.h` got deprecated, which is referenced in the `linux/posix.def` file. For RISC-V this has to be replaced with `locale.h`. [65]

Changing any Linux targeting C-Interop configuration file will affect the compilation of existing targets with older gcc and glibc versions. Accordingly, a new directory (`linux_for_riscv`) is created for the platformlibs task for RISC-V . This must be taken into account for compilation, which is why the change in Figure in Appendix E.7 can be used to load the new def files for RISC-V from the new created directory. This is a provisional solution that should be revised for integration into the JetBrains repository.

With these changes the Linux platform libraries and the Runtime can be compiled completely including all existing targets and the new added RISC-V target. As in the bootstrapping phase (see Chapter 4.1) the new build compiler needs to be provided with the gradle task 5.

The presented changes show, that the version of the compiled toolchain and its dependencies are important and indicate the upcoming changes. The changes presented might differ if a different toolchain with different versions is targeted. For the targeted toolchain, all changes enable the compilation up to the linker phase. The error described in Chapter 3.4 occurs at compile time. As described,

```
CMake Error at CMakeLists.txt:136 (MESSAGE):
  libcxx isn't a known project:
  bolt;clang;clang-tools-extra;compiler-rt;
  ↪  cross-project-tests;libc;
  ↪  libclc;lld;lldb;mlir;openmp;polly;pstl;tapi;flang.
  Did you mean to enable it as a Runtime in LLVM_ENABLE_RUNTIMES?
```

Listing 12: Error when using package.py

```
-        projects = ["clang", "lld", "libcxx", "libcxxabi",
↪  "compiler-rt"]
-        runtimes = None
+        projects = ["clang", "lld", "compiler-rt"]
+        runtimes = ["libcxx", "libcxxabi"]
```

Listing 13: Changes in package.py

the relaxation and align relocation symbols of the C-Runtime cannot be handled by the linker.

4.7 Strategy 1: Use linker with relaxation

As strategy 1 states, a linker with a relaxation function should be used. Therefore, a newer linker with a higher LLVM version can be targeted through the `linker.macos_x64-linux_arm64` attribute in the `konan.properties`. To obtain the new linker, the LLVM distribution can be built by the JetBrains build script mentioned in Chapter 3.1.1. The script is located in the `/kotlin-native/tools/llvm_builder` directory.

Command line invocation 6.

```
python3 package.py --branch stable/20230725
```

The `package.py` script can be executed with the command line invocation 6. Using `--branch stable/20230725` references a newer forked version of LLVM on apple's LLVM-project repository. [70]

According to the git history, the file was last modified 3 years ago. This indicates that changes may be necessary. Listing 12 shows an error, which indicates that the Runtime and Project settings are not correct in the build script. According to the LLVM documentation, the `libcxx` and `libcxxabi` arguments belong to a Runtime and not to a project. [71] Listing 13 shows the required change.

For a uniform structure, the resulting directory is also moved to the `$HOME/.konan/dependencies` directory. This enables easier referencing in the `konan.properties` file. With this build, the `clang++` and the linker `ld.lld`

```
-linker.macos_x64-linux_arm64 =
↪  $targetToolchain.macos_x64-linux_arm64/bin/ld.lld
+linker.macos_x64-linux_arm64 =
↪  $targetToolchain.macos_x64-linux_arm64/../
↪  apple-llvm-20230725-macos-x64-1/bin/ld.lld
```

Listing 14: Changing LLVM linker dependency in konan.properties

version is targeting version `17.0.5`. Finally, the new build linker can be used by referencing it in the `konan.properties` file (see Listing 14).

With these changes, it is possible to compile a Kotlin source file into a executable binary targeting RISC-V . Chapter 5.3 shows that the file has the correct file type and that it can be executed in a RISC-V emulation.

4.8 Strategy 2: Update LLVM dependency

Although the approach of strategy 1 works, its major flaw is, that it cannot be integrated in the final project. To also provide changes that can be adopted, strategy 2 (Chapter 3.6) is presented. The following chapters will show its implementation attempt in detail. To pursue strategy 2, the version of the LLVM distribution is updated. The manually built LLVM distribution presented in Chapter 4.7 is used here also. This means that the LLVM version used internally is increased from version `11.1.0` to `17.0.5`. Therefore, all release notes and changes to LLVM distributions 12, 13, 14, 15, 16 and 17 must be taken into account. Since the tools provided by LLVM are used or integrated in many places, many changes can be expected. The usage of the LLVM distribution is described in Chapter 2. As a consequence, this chapter presents changes in the repository that are necessary due to the version change. However, this collection is not complete and not all problems that arise can be solved. Therefore, this strategy does not fulfill the goal of completely extending the Kotlin/Native compiler.

In summary, the new LLVM version must be referenced, problems with modified and removed files such as the legacy pass manager and the use of fixed pointer types must be resolved, type casts must be added, the macOS version being used must match the macOS version of the LLVM libraries being used, ZSTD must continue to be available as an include, signal handling between JVM and native execution must be guaranteed and configurations must be updated.

4.8.1 Reference a new LLVM distribution

After the LLVM distribution has been build according to Chapter 4.7, not only the linker has to be referenced in the `konan.properties`, but the entire LLVM distribution. Listing 15 shows that the new LLVM distribution can be set as the used LLVM toolchain for the host directly. After this change, the compiler tools with version `17.0.5` will be used.

```
-llvmHome.macos_x64 = $llvm.macos_x64.dev
+llvmHome.macos_x64 =
↪  $REPO_ROOT/kotlin-native/tools/llvm_builder/build
```

Listing 15: Use custom LLVM distribution by konan.properties

4.8.2 Resolve deprecated and moved files

Any breaking change between the previous version `11.1.0` and the new version, like the renaming, move or deletion of files, functions or constants will break the compilation process or the build of the compiler itself.

For example, the Interprocedural optimization (`IPO.h`) has been removed. Its reference has to be removed in the `llvm.def` file. The same applies for the `Initialization.h` file. [72]

In a similar way, the Clang-C API has changed and in commit `29e0435` targeting Clang release `16.0.0` some functions and data structures were extracted from `clang-c/Index.h` into `clang-c/CXFile.h`, `clang-c/CXDiagnostic.h` and `clang-c/CXSourceLocation.h`. These need to be referenced in the `clang.def` file as well, to ensure all files can be linked correctly. [73]

The path of `llvm/ADT/Triple.h` was changed to `llvm/TargetParser/Triple.h`, therefore the `CAPIExtensions.cpp` and `CoverageMappingC.cpp` file has to be updated. It has to be noted, that `CoverageMappingC.cpp` has been removed in the meantime in commit `4f77434` and the discovered change relies on an earlier version of the JetBrains/Kotlin repository, based on version `2.0.0-Beta-2`. [74]

In `kotlin-native/libclangext/src/main/cpp/ClangExt.cpp` the `getNullability` function of `QualType` is used, which has been changed and the `AstContext` does not need to be provided anymore, and the return type changed from `llvm::Optinal` to the `std::optional`. By this, the syntax of the `hasValue` function has changed also. These changes can be read by the git history and were introduced in LLVM version `16.0.0`. [75][76]

In `kotlin-native/runtime/src/alloc/legacy/cpp/ObjectAlloc.hpp` the struct makes use of an implicitly-declared copy constructor, which results in an exception since `C++17`. Therefore, it has to be removed. [77]

All required changes of `llvm.def` can be seen in Appendix F.1 and `clang.def` in Appendix F.2.

4.8.3 Resolve stricter type handling

In `Logging.hpp` the cast of the passed tag, which is an enum class of type int32_t cannot be casted implicitly for the String formatting. This has been possible before. To resolve errors during the build, the cast has to be done explicitly.

Gradle task 6.

:kotlin-native:Interop:Indexer:updatePrebuilt

The in general stricter type handling leads also to the fact, that the `createExpression` function that is used in `kotlin-native/llvmDebugInfoC/src/main/cpp/DebugInfoC.cpp` has to be updated to an uint64_t instead of an int64_t, since the unsigned type is expected and not casted implicitly. In the same manner, the generated file `clangstubs.c` passes a jlong type to a function argument of type `void *`, which is marked as an error, shown in Appendix D.3. In general, the new targeted GNU Compiler Collection got more strict about type parameters and enforces explicit casts. Instead of fixing this problem in the generated `clangstubs.c` file only, the generation file should be fixed also. This generation is done in the `MappingBridgeGeneratorImpl.kt`. Appendix F.3 shows the manually added cast for `memcpy` invocations. To regenerate the `clangstubs.c` the gradle task 6 can be used.

4.8.4 Increase targeted macOS version

Further warnings and errors avoid the completion of the gradle task. The LLVM libraries that are generated in Chapter 4.7 are build with a different target macOS version than the current linked version. This warning is issued by the linker. The libraries are build locally and in this case version `14.0` is used. Appendix D.4 show few of many warning statements. The targeted macOS version can be specified in the `konan.properties` file and is set to version `11.0`. Setting the `minVersion.macos` to `14.0` resolves the warnings.

4.8.5 Update linkage of ZSTD library

The linker command fails also, because several symbols cannot be found for the host architecture. Appendix D.5 shows the linker error. To ensure the ZSTD library is linked correctly the `build.gradle.kts` file has to be updated. The path and library can be passed as linker argument as shown in Appendix F.4.

In LLVM release 15 the place for the ZSTD compression files has changed and ZSTD has to be enabled by the builder of the LLVM toolchain manually. This can be recognized from the commit. Only if `LLVM_ENABLE_ZSTD` is defined correctly, the ZSTD library will be included into `Compression.cpp` and therefore compiled into the `Compression.cpp.o`. This step was not done for the custom build LLVM distribution and therefore the ZSTD library is not part of the build files. The linked library libLLVMSupport cannot find the symbols that are defined in the moved ZSTD code. The fix to link ZSTD is a temporary fix, but for a final integration or the attempt to bump LLVM in Kotlin, the LLVM distribution should be configured correctly. [78] [79]

The entry in the release notes of LLVM 15 is missing, but a similar change for zlib is part of the release notes. The first time that the flag is mentioned is in the release notes `15.0.3`, where bugfixes based on the change are added. [80]

It needs also to be linked in the `llvm.def` file (see Appendix F.1).

> The problem is that those are incompatible with JVM, leading to JVM crashes when LLVM is used directly in a process with JVM (like it is when running the Kotlin/Native compiler or cinterop).

Listing 16: Issue description of a known problem handling with errors [81]

4.8.6 Avoid crashes in signal handling between the compiler's Java Runtime and native code execution

After the previous issues are resolved, another error occurs, which is shown in Appendix D.6. The affected function `initSignalChaining` belongs to the file `src/nativeInteropStubs/cpp/signalChaining.cpp`. According to a JetBrains youtrack issue, this code is used to set up handling for LLVM and clang signal handlers. [81]

The encountered SIGBUS signal is generated, because an invalid pointer is dereferenced. This might be the case because the clang.kt file has not been regenerated yet, and some functions still point to an older version of LLVM and therefore point to not existing functions or constants. [82] One way to avoid this is to skip the initialization process in the `signalChaining.cpp` entirely. This will not solve the origin of the crash, but will enable further research on further changes that will be required.

Another reason that the signal chaining will not work is that the LLVM project refactored the signal handling process for LLVM release 16. The defined values for the signals have been moved from `signal.h` to `libc/include/llvm-libc-macros/linux/signal-macros.h`. [83] [84] Further analysis to resolve the problem remains open.

4.8.7 Update build settings

When building the entire project again with the new LLVM version also errors occur regarding the passed arguments. Previously, the macOS version has been passed by the `__ENVIRONMENT_OS_VERSION_MIN_REQUIRED__` argument to clang. This is specified in the `native/utils/src/org/jetbrains/kotlin/konan/target/ClangArgs.kt` and is just a temporary workaround. In the new build, this leads to a redefinition of the flag, since the existing linked files already define the version. The workaround can be removed now. The error is shown in Appendix D.7.

Gradle task 7.
:native:kotlin-native-utils:publish

To enable the change, the gradle task 7 has to be used. The build files are not automatically recreated with every build, even if changes have been made.

> Using the legacy pass manager for the optimization pipeline is deprecated and will be removed after LLVM 14. In the meantime, only minimal effort will be made to maintain the legacy pass manager for the optimization pipeline.

Listing 17: Deprecation of legacy pass manager in LLVM release notes 13 [25]

> LLVM now uses opaque pointers. This means that different pointer types like i8*, i32* or void()** are now represented as a single ptr type. See the linked document for migration instructions.

Listing 18: Announcement of deprecation of opaque pointers in LLVM release notes 15 [86]

4.8.8 Inspect changes on the LLVM pass manager

Since the legacy pass manager functions got deprecated and removed during the releases (see listing 17), many functions are not existing anymore and therefore the according stub in the `llvm.kt` file has been removed. Appendix D.8 shows the error log and the missing stub functions. This affects mostly the `OptimizationPipeline.kt` file. For further development, the changes were ignored and the calls to the LLVM-C library were commented out at this point. Also, the `createThreadSanitizerLegacyPassPass` invocation that is called in `CAPIExtensions.cpp` has to be removed.

To fully support LLVM 17, all new pass manager bindings must be used and the old legacy pass manager must be removed. [85][22]

4.8.9 Inspect changes on the switch from fixed to opaque pointers

With LLVM version 15 typed pointers are no longer supported (see listing 18). Due to the change from typed pointers to opaque pointers, the code generation needs to be adapted. The LLVM documentation specifies which functions are replaced (see listing 19). The handling of the return values must also be taken into account. [20]

Besides the renaming of the used functions, also the way how the return values are handled, stored and compared has to be updated. The necessary changes are beyond the scope of this work, so they will not be made and remain open.

4.8.10 Inspect further changes

Besides the mentioned changes, further changes can be identified, but are not deeply analyzed. On the one hand, the compiler claims about implicit declarations for the windows target build. The error is shown in Listing 20 and the temporary path for the file generation is substituted. It is possible to resolve

```
LLVMBuildLoad -> LLVMBuildLoad2
LLVMBuildCall -> LLVMBuildCall2
LLVMBuildInvoke -> LLVMBuildInvoke2
LLVMBuildGEP -> LLVMBuildGEP2
LLVMBuildInBoundsGEP -> LLVMBuildInBoundsGEP2
LLVMBuildStructGEP -> LLVMBuildStructGEP2
LLVMBuildPtrDiff -> LLVMBuildPtrDiff2
LLVMConstGEP -> LLVMConstGEP2
LLVMConstInBoundsGEP -> LLVMConstInBoundsGEP2
LLVMAddAlias -> LLVMAddAlias2

Additionally, it will no longer be possible to call
↪  LLVMGetElementType() on a pointer type.
```

Listing 19: Required frontend C API changes [20]

```
Exception in thread "main" java.lang.Error:
↪  $TMP_PATH/4102919711648418251.c:43:12: error: call to
↪  undeclared function 'htons'; ISO C99 and later do not
↪  support implicit function declarations
↪  [-Wimplicit-function-declaration]
```

Listing 20: Error during gradle task:kotlin-native:platformLibs:compileKonan-Mingw_x64-posixMingw_x64

```
LLVM is now built with C++17 by default. This means C++17
↪  can be used in the code base.
```

Listing 21: C++17 is used for LLVM distribution [87]

these by reordering the code snippet of `kotlin-native/platformLibs/src/platform/mingw/posix.def`. The required `include <winsock2.h>` has to be at the very top of the added code snippet within the `posix.def` file.

On the other hand, the compiler complains about missing symbols in the built files for the Runtime module. Appendix D.9 shows an excerpt of the error message. An analysis of the Kotlin source code and the bitcode files shows that this is true. The `struct.TypeInfo` should be part of the Runtime module, which is true. Disassembling the `runtime.bc` bitcode file shows that the `struct.TypeInfo` is defined. Although, not the `runtime.bc` but the `compiler_interface.bc` bitcode file is load and passed in the Kotlin Runtime initialization. The change seems not related to the bump of LLVM, but the manual rebuild of the Kotlin/Native compiler produces the error. To disassemble the bitcode file, the `llvm-dis` tool is used.

Moreover, the release notes of LLVM version 16 emphasize that `c++17` will be used from this version onwards (see listing 21). The Kotlin/Native compiler project refers to lower versions at many configurations, as in CMakeLists and gradle task configurations. Adapting these to `c++17` and taking its release notes into account is another open task.

All challenges represent extensive tasks and can mostly only be solved through wide-ranging code changes. Added to this is the analysis and research into the causes and required changes. With the remaining open tasks, the attempt to bump LLVM in the Kotlin repository is discontinued.

5 RISC-V emulation and execution

To assert that the build binary with the implemented extension of Chapter 4.7 can be executed on RISC-V targets, its file format can be checked first. Hence, it matches the expected ELF file format, the executable should be executed on a RISC-V target. As it is very common in development, an emulator can be used if no hardware is available. One solution to emulate the target system is QEMU.

QEMU can be installed on many operating systems. For macOS on Intel, the tool can be installed with homebrew. Other than on Linux machines, only the full system emulation is available by the program `qemu-system-riscv64`. The kernel and the Linux image has to be supplied to run the executable. To get the kernel image and the required Linux image with several system functions, the getting-started guide of RISC-V can be followed. [88] [89]

To build the Linux image, it has to be compiled by using a cross-compiler, which has to be installed on the host machine. With the build image and the kernel, the emulator should be bootable, and the file can be executed on the emulator. To install a RISC-V cross-compiler using Homebrew, the `riscv-software-src/riscv` tap has to be added to the Homebrew environment. After this has been done, the cask riscv-tools can be installed. [90]

Chapter 5.1 will present an error during the build of the Linux image, which leads to the circumstances that QEMU cannot be started on the macOS Intel host.

An attempt to solve this by building the cross-compile-toolchain for the macOS Intel host targeting RISC-V was made, but resulted in errors as well. This is described in Chapter 5.2.

As a workaround, Chapter 5.3 will describe how a dockerized environment of David Burela is used to launch a running RISC-V emulation. On this running RISC-V environment, the file can be executed, and it can be verified whether the compilation was successful or not.

5.1 Error during building the Linux image

To build the Linux image for QEMU, the RISC-V getting-started documentation instructs to clone the Linux repository, check out the required version and build the kernel. Appendix D.11 show the attempt that was made, which resulted in errors.

The major reason for the failure during the build is that `elf.h` and `asm/types.h` cannot be found. One reason for this could be, that the guide has to be adjusted, since the cross-compiler for RISC-V can be installed only for bare metal targets. Instead of using `riscv64-unknown-linux-gnu`, the bare metal compiler `riscv64-unknown-elf` is passed. An attempt to create the Linux specific compiler `riscv64-unknown-linux-gnu` is discussed in Chapter 5.2.

It turns out, the Linux image cannot be build and provided for the macOS Intel host with the riscv-tools provided by homebrew.

```
file program.kexe
program.kexe: ELF 64-bit LSB executable, UCB \riscv, RVC,
↪  double-float ABI, version 1 (SYSV), dynamically linked,
↪  interpreter /lib/ld-linux-riscv64-lp64d.so.1, for GNU/Linux
↪  4.15.0, BuildID[xxHash]=50d302fe8651e550, not stripped
```

Listing 22: Inspect file type of compiled program

```
docker pull davidburela/riscv-emulator
docker run -d --publish 127.0.0.1:2222:2222/tcp
↪  davidburela/riscv-emulator
```

Listing 23: Start the RISC-V docker container of David Burela

5.2 Error during building the cross-compile-toolchain

The RISC-V cross-compiler that can be installed by homebrew results in a selection of tools, that only target bare metal targets, but not a target running Linux. To exclude the possibility that the used compiler is the problem, the correct Linux specific RISC-V compiler has to be built manually on the macOS Intel host.

Therefore, the riscv-gnu-toolchain repository is cloned and the instructions in the README are followed. Eventually this results in an error as well, since the build clashes with the current C environment. The error is shown in Appendix D.10.

Most errors have their origin in referenced files that are located in the Xcode directory. The entire environment is configured by the Xcode command line tools, a macOS developer toolset, which is tight to the running macOS version. It is also required by the Kotlin Project to enable the build for all macOS and iOS targets. There is a chance that an inappropriate version is being used for one of the programs provided by Xcode command line tools and there is a version conflict.

Without a manually built compiler, further attempts to build the Linux image for QEMU cannot be made.

5.3 Set up a running RISC-V system within docker

Since the approach to run QEMU with a custom build Linux kernel does not work without further time invest, an alternative solution can be considered. David Burela maintains a free and open GitHub project, in which a docker image is presented to run a RISC-V QEMU instance. Within a Debian Linux, the QEMU emulator is hosted and a ssh connection is enabled. [91]

The running instance can be used to verify that the compiled executable can be run within a RISC-V environment. Listing 22 ensures that the file format of

```
scp -P 2222 program.kexe root@localhost:.
```

Listing 24: Copy program.kexe to RISC-V test environment

```
ssh root@localhost -p 2222
root@localhost's password:
Linux debian 6.1.0-7-riscv64 #1 SMP Debian 6.1.20-1 (2023-03-19)
↪  riscv64

Last login: Thu May  2 10:08:34 2024 from 172.17.0.1
root@debian:~# ls
program.kexe
root@debian:~# ./program.kexe
Hello, Kotlin/Native!
```

Listing 25: Run program.kexe in RISC-V test environment

the executable is correct. Listing 23 shows how the docker image can be pulled and run. Listing 24 can be used to push the compiled executable to the target. Listing 25 presents the execution of the file.

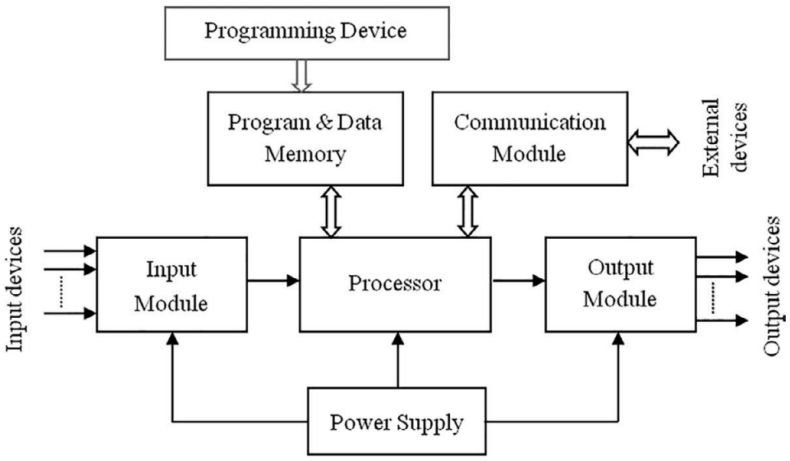

Figure 4: Block diagram of a PLC [92]

6 Related work

The focus of this thesis is the extension of the Kotlin/Native compiler, which utilizes the properties of LLVM. The following chapters present further research work, which on the one hand illustrates the advantages of LLVM as an intermediate representation and on the other hand presents further possible problems that can be solved by using LLVM.

Therefore, Chapter 6.1 presents how Programmable Logic Controller (PLC) can be programmed to run on RISC-V processors. This process involves translating a descriptive language called Instruction List, which is designed for engineers, into C code, which in turn can be converted into assembly code using the RISC-V toolchain. This process will be optimized in further research work so that other descriptive languages for PLCs' can be translated directly into a LLVM IR. The reference to this work is clarified again at the end of Chapter 6.1.

In order to present problems that do not need to be considered in this thesis, but which play a role in porting to new platforms, Chapter 6.2 presents the porting of a JIT compiler to RISC-V , which has not yet been designed for the LLVM toolchain. If the Kotlin/Native compiler were to implement its own IR for native development, the findings from Chapter 6.2 would have to be taken into account.

6.1 PLC implementation on RISC-V

In the industry, Programmable Logic Controller (PLC) are used to process digital and analog signals. Input devices like switches, buttons, sensors and more are used to provide inputs to a program which is running on a processor which

eventually returns its results through output devices like relays, LEDs or any kind of signal. Figure 4 illustrates this mechanism. The program itself is executed on the CPU, which interacts with the input and output modules. It turns out that a PLC is more than just a CPU. The combination of the various inputs and outputs can be described graphically and textually. The International Electrotechnical Commission (IEC) defined five programming language standards which are grouped as the IEC 61131-3 standard that are used to describe a PLC. [92]

6.1.1 Transpilation of Instruction List into C code to run a PLC on a RISC-V CPU

Back in 2019 Hossameldin Eassa, Ihab Adly and Hanady H. Issa proposed a "RISC-V based implementation of Programmable Logic Controller on FPGA for Industry 4.0"[93]. In their paper, a Field Programmable Gate Array (FPGA) is used to implement the instruction set of RISC-V to be used for PLCs. The PLC can be described by the Instruction List (IL), which is one of five languages which are defined in the IEC standard. To make this work, the team developed a special compiler that can compile the IL into a C program, which then is compiled using the RISC-V toolchain to binary code. To initialize the content of a memory block of an FPGA, a Memory Initialization File (MIF) file has to be load to the FPGA. This MIF file is created by the before created binary code. The compiler also generates the Hardware Description Language (HDL) files to represent the processor and the required peripherals. Both will be downloaded as a SRAM Object File to the FPGA.

By providing the HDL, the necessary hardware blocks to complete the IL are declared, along with the signals and input and output pins. This way hardware components like I/O peripherals, timers, counters, Pulse Width Modulations (PWM) and PID can be used by the CPU as coprocessor.

The main motivation of combining the HDL and the IL description on the RISC-V CPU is to reduce clock cycles by the hardwired components and to enable parallel execution of independent instructions. [93] [94]

6.1.2 Implementation of a LLVM frontend for the IEC 61131-3 standard

In supplement, the paper "IEC 61131-3 frontend for the LLVM Compiler Family" [95] presents an approach that does not support IL, but does support the four other languages, Structured Text (ST) and Sequential Function Chart (SFC), Ladder Diagram (LD) and Function Block Diagram (FBD) to be compiled to a LLVM IR and therefore potentially to RISC-V ISA based machine code. However, this is not the primary goal of the work. Rather, the aim is to provide a front end for LLVM that can convert the IEC 61131-3 languages directly into an LLVM IR and can therefore be made available for all LLVM-supported architectures. In addition, the optimization passes provided by LLVM can be used. [95]

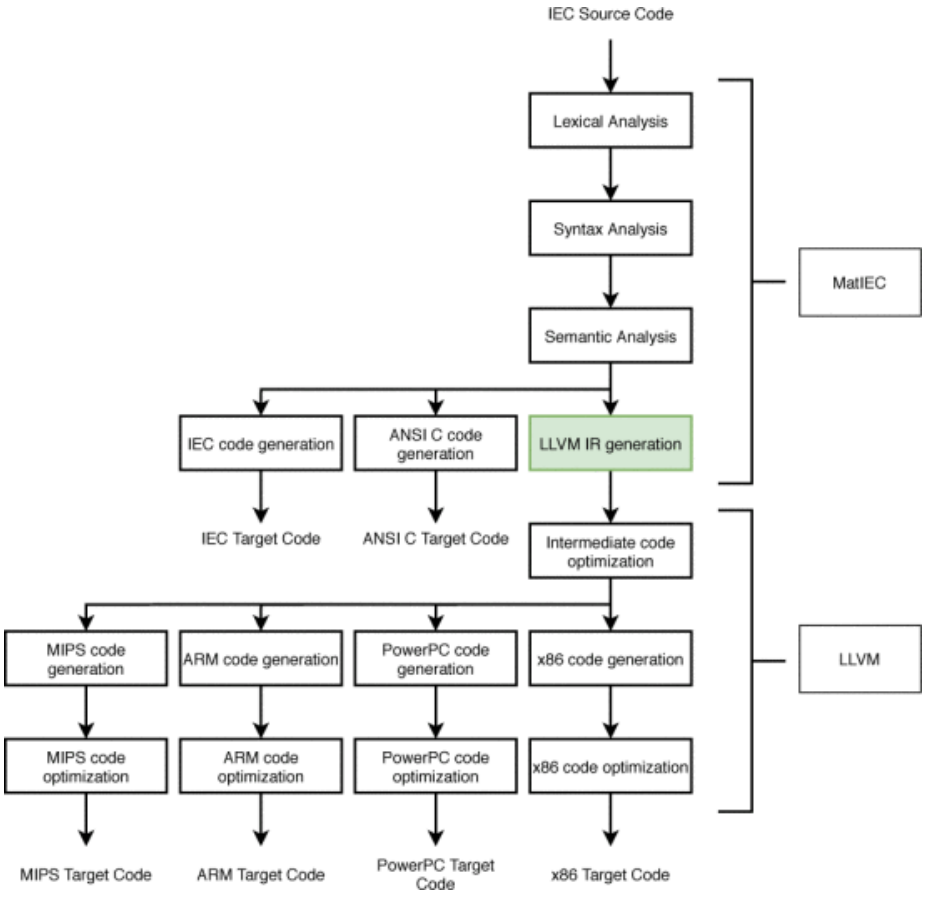

Figure 5: Matiec LLVM integration [95]

The authors Tiago Catalão and Mário de Sousa adapted the work of Edouard Tisserant, Laurent Bessard and de Sousa himself "An Open Source IEC 61131-3 Integrated Development Environment"[96], where a IEC 61131-3 compiler named matiec is used to transform the IEC 61131-3 program into an ANSI C code which is then compiled into C code, which is eventually used to generate the machine code. [96]

To reduce the overhead that was created by this pipeline, Catalão and de Sousa adopted the maetiec lexical, syntax and semantic analysers and implemented a LLVM IR generator.

Figure 5 visualizes the resulting possibilities of the approach.

6.1.3 Discussion and reference

As Chapter 6.1.1 and the paper of Hossameldin Eassa et al. shows, the RISC-V ISA implemented by a FPGA can be a good solution for prototyping PLCs.

The implementation shows that the C language is one crucial intermediate step to convert the descriptive Instruction List into machine code. By using the RISC-V toolchain to compile the C-Code into a machine code, the LLVM IR is indirectly utilized.

Although the IEC 61131-3 languages fulfill a completely different purpose than Kotlin/Native, they have the commonality of compiling the language directly for hardware ISAs and both rely on the LLVM IR as an intermediate step to support a wide range of ISAs. In an even more direct way, the work "IEC 61131-3 frontend for the LLVM Compiler Family" aims to translate the IEC languages into a LLVM IR and is very similar to the effort presented in this work. Instead of the IEC 61131-3 standard, Kotlin is converted to an LLVM IR. The Kotlin/Native compiler thus represents a LLVM frontend for Kotlin.

6.2 Porting the Pharo JIT compiler to RISC-V

Quentin Ducasse, Guillermo Polito, Pablo Tesone, Pascal Cotret and Loïc Lagadec present an approach that is intended to solve various challenges that arise when porting the Cogit JIT compiler to a RISC-V architecture. In their work "Porting a JIT compiler to RISC-V: Challenges and Opportunities"[97] they present problems which, due to historically induced wrong decisions, mean that RISC-V can only be ported with major changes to the design of the Cogit IR. This in turn affects the existing porting for Arm and Intel. The main reason for this is the heavy reliance on the x86/x64 ISA from Intel during the IR design phase. The differences between the RISC-V ISA and the Arm and Intel ISAs as well as the flexibility of the RISC-V ISA has posed major challenges for the extension of the JIT compiler Cogit that never had to be solved before.

Chapter 6.2.1 summarize the interaction of the Pharo language, the Pharao VM, the Cogit JIT compiler and the CogRTL IR. In Chapter 6.2.2 the main causes of conflicts between x86 and RISC-V presented in the paper are shown. Eventually, the correlation between the paper and the present work is discussed in Chapter 6.2.3.

6.2.1 Pharo language

To execute a program that is written in the pharo language, its VM is used. It scans the program's byte code, parses it by its Just in Time (JIT) compiler Cogit into a Cogit Intermediate Representation named CogRTL and generates machine code by the IR. This is a linear and non-optimizing process, which already hints that it is tightly coupled to the machine code and its stated, that the finally generated machine code mostly has a one-to-one mapping to the JIT IR. The JIT compilation process is shown in Figure 6. [97]

6.2.2 Conflicts between x86 and RISC-V

Cogit was originally developed for the x86 architecture and extended to ARM and further ISAs. As a result, the concept of conditional registers was adopted

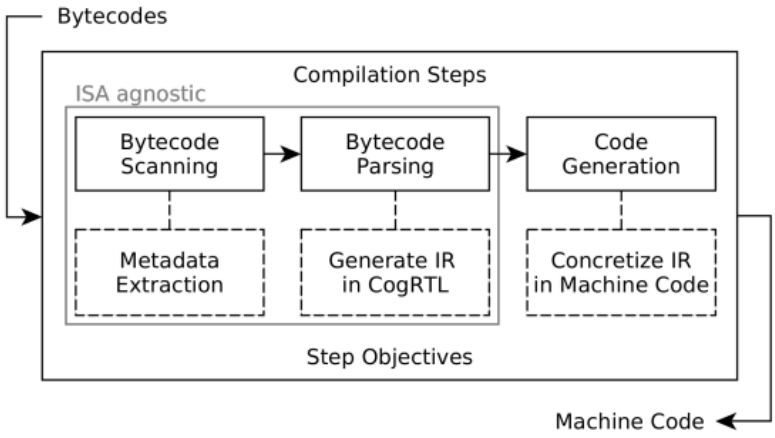

Figure 6: Cogit compilation phases [97]

```
# CogRTL instructions
cogit CmpR: ClassReg R: TempReg
cogit JumpNonZero: (Label 2).

# ARMv8 output
cmp r1, r22 # x1 and x22 are ARMv8 class/temp registers
b.ne 48     # label 2 has offset 48
```

Figure 7: Translation of a CogRTL conditional jump to an Armv8 machine code[97]

for the intermediate representation, among other things. Some IR instructions therefore use implicit flag codes from previous instructions. According to the Arm reference manual, the Current Program Status Register (CPSR) is present as one of the general system control registers. In the CPSR flags like the negative condition flag, carry condition flag, overflow condition flag or zero condition flag are set, when an instruction has been executed. The latter is set to 1 if the instruction `CMP R0, R1` was executed and the registers `R0` and `R1` are equal. Sequentially, a `beq label` instruction will jump to the label if the Zero results flag in the CPSR has been set. Similarly, Intel provides a EFLAGS register, which contains status and system flags. [98] [99]

Due to the x86 ISA aligned design of the CogRTL IR some IR instructions like the conditional jump `CmpCqR` instruction will assume the Zero results flag has been set implicitly and makes use of it. In RISC-V this has to be checked manually by adding further instructions and checking the result register. Figure 7 shows the translation of the CogRTL instruction to Armv8 machine code.

In contrast, RISC-V does not define flag registers but compares the values of two registers directly. The translation of the two commands CmpR and

```
bne x<class>, x<temp>, 48
```

Listing 26: RISC-V machine code representation for the commands of Figure 7[97]

```
CmpRR x<class>, x<temp>
JumpZero label
```

Listing 27: Example of initial Cogit IR

```
BrEqualRR   ; next instruction is JumpZero
x<temp>     ; newBranchLeft := operands at: 1
x<class>    ; newBranchRight := operands at: 0
NewLabel    ; CmpRR instruction is converted to a Label
```

Listing 28: Example of concreting step

```
BrEqualRR x<temp> x<class>
NewLabel
```

Listing 29: Example of concreted Cogit IR

JumpNonZero must be combined into one command for porting to RISC-V . Listing 26 represents the machine code for RISC-V .

As a solution for this, the team proposes different approaches. On the one hand, scratch registers can take over the role of flag registers. Accordingly, individual IR instructions would have to be mapped to several instructions. This applies above all to arithmetic and logical operations.

On the other hand, a refinement of the compilation process is proposed. The IR for architectures that do not have conditional codes needs to be concreted further. If the current instruction is followed by a conditional instruction both, the current and the next instruction will be used eventually to determine how the IR should be mapped. The next opcode determines the new opcode. The operands of the current instructions as well as the operands of the next opcode will be combined and used for the new instruction.

As an illustrative example, the instructions from Listing 27 are mapped into the Cogit IR instructions from Listing 29. To do this, Listing 28 shows the calculation based on the change presented by the team. Listing 30 and 31 show the final mapping to the machine codes.

Furthermore, the paper emphasizes that not all instructions are available as they are in x86 or Arm. As an example, the `rol` and `addOverflow` instructions are named. A few might be supported by RISC-V extensions, but in cases when

```
cmp x<temp>, x<class>
b.eq NewLabel
```

Listing 30: Cogit IR to Armv8

```
beq x<temp>, x<class>, NewLabel
```

Listing 31: Cogit IR to Armv8

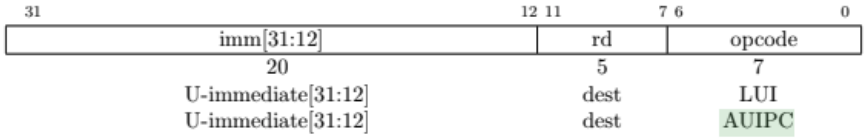

Figure 8: AUIPC description in the U-Type format [44]

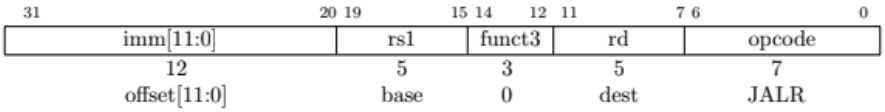

Figure 9: JALR description in the I-Type format [44]

the hardware does not support the extension, the required instruction must be combined by simple base instructions. This needs to be considered too when the machine code is generated. In these cases, a one-to-one mapping is not possible, and it is to be expected that a single IR instruction will be split into multiple machine code instructions.

It also highlights differences in the way RISC-V handles immediate values and how the sign extension is applied. Although it is not presented how Arm handles, it is emphasized that the handling of values larger than the offset size of a command in RISC-V must be taken into account when porting. In case of jumps which can be invoked by the Cogit IR `call` instruction, the team proposes an implementation that can be used to mitigate wrong address interpretation by the fact that RISC-V makes use of sign extension. As in Chapter 2.3 presented, AUIPC and JALR can be used to jump anywhere in a 32-bit pc-relative address range. AUIPC can be used with a 20 bit immediate value, which will write the value as the upper 20 bits into the given rd register. The lowest 12 bits are set to 0. The format for the AUIPC instruction is presented in Figure 8. To set the lowest 12 bits, the JALR instruction will be used. The format for the JALR instruction is presented in Figure 9. [44]

Since the sign extension is applied indirectly, the 11th bit has to be verified

before AUIPC is called. If the 11th bit of the offset is set, 0x800 has to be added to the offset before the upper 20 bits are passed. In that case, the lower 12 bits will be sign extended as negative value, which will result in a subtraction on the Program Counter (PC). [97]

Description	Decimal	Binary	Comment
Example 1			
Offset	1717983028	01100110011001100101_**0**11100110100	11th bit is 0
Offset upper 20 bits	1717981184	01100110011001100101_000000000000	
Offset lower 12 bits	1844	00000000000000000000_011100110100	
AUIPC	1717981184	01100110011001100101_000000000000	0x800 is not added
JALR	1844	00000000000000000000_011100110100	positive sign extended
PC	1717983028	01100110011001100110_011001100100	1717981184 + 1844
Example 2			
Offset	1717985076	01100110011001100101_**1**11100110100	11th bit is 1
Offset upper 20 bits	1717981184	01100110011001100101_000000000000	
Offset lower 12 bits	3892	00000000000000000000_111100110100	
AUIPC	1717985280	01100110011001100110_000000000000	0x800 added
JALR	-204	11111111111111111111_111100110100	negative sign extended
PC	1717985076	01100110011001100101_111100110100	1717985280 + (-204)

Table 5: Example to visualize the proposed solution

Table 5 shows the calculation using two examples. In the first example, `call 1717983028` will be called to jump to the address 1717983028. When the RISC-V Cogit compiler extension which is described in the paper is used, the IR instruction will be mapped to AUIPC and JALR and the passed immediate values will not be changed, since the 11th bit is not set to 1. The table shows the values for the offset as decimal and binary value. The binary value is separated with a _ to separate the 32 bit value into a chunk of 20 and 12 bits.

In the second example, `call 1717985076` will be called to jump to the address 1717985076. In that case, the 11th bit is set to 1, which results in 0x800 being added to the offset before the 20 highest bits of the offset are passed to the AUIPC command. The 12 lowest bits of the offset are interpreted as a negative value by the JALR instruction and will be subtracted from the value that was set by the AUIPC instruction.

Eventually, a similar issue is given for the Load Immediate (LI) command, due to the limited number of bits for immediate values. To load a 64 bit or 32 bit immediate value into a register, multiple machine code instructions need to be used. To set the first 20 bits of the register, the LUI instruction can be used (see Figure 8). To set the other bits, a combination of ADD Immediate and Shift Logical Left Immediate (SLLI) have to be used. In this case, the sign extension has to be considered also. The team emphasizes, that a single load can cause up to 8 instructions each time an immediate value is load into a register.

6.2.3 Discussion and reference

All presented challenges lead the team of Qentin Ducasse Et al. to question the x86 aligned Cogit IR. For the future, they propose to abstract the JIT IR and to shift the machine code generation into different compiler backends. With that, the backend will handle the mapping of IR instructions for branches and handling immediate values and register usage correctly.

The approach presented is similar to that of the LLVM compiler, which already implements the proposed efforts to port the Cogit compiler. With that, the challenges have already been solved for Kotlin/Native too, since the LLVM compiler is used internally. Unlike the work presented in the paper, the IR does not need to be adjusted in this work. Neither the Kotlin IR nor the PSI of the frontend nor the LLVM IR itself require any changes for porting Kotlin/Native to RISC-V .

The challenges presented in combination with the comparatively high level changes presented in this work highlight the advantages of the LLVM IR and show how significant the design decisions can be. It can be concluded that Kotlin/Native is also well suited for further porting in the future, assuming the target architectures are supported by the LLVM toolchain.

Although, it should be noted, that the Kotlin/Native compiler is not a JIT compiler but an Ahead Of Time (AOT) compiler. The executable program is compiled completely before it is executed, which already resolves issues presented in the paper. The compilation time might be less critical, and therefore it

is possible to run more passes. The presented Cogit compiler is described as linear and not making use of optimization passes. In contrast, the Kotlin/Native compiler uses several passes to lower the Intermediate Representation and to make use of optimizations like function inlining and link time optimizations.

In general, when porting to a new target architecture, the current functionality and, if applicable, the existing IR should be taken into account. Particularly in the present case of JIT compilation, the translation of individual instructions to multiple instructions and the use of special registers is an important aspect. In contrast to this work, the team's work shows what challenges can arise at a lower level.

7 Summary

Finally, the outcome of this work, the different strategies and the problems will be summarized and commented in Chapter 7.1. Further tasks and open work is outlined in Chapter 7.2. The importance of this work will be expressed in Chapter 7.3. At the beginning of the work, Kotlin version `1.9.21` was released. At the end of the work, version `2.0.0` is the current version. Work on the Kotlin repository that was done in parallel to this work during the same period is commented in Chapter 7.4.

7.1 Conclusion

Foremost, it can be said, that the provided work could partially achieve the objectives of Chapter 1.3. Due to the possibilities that the Kotlin/Native compiler offers to customize compilations, as for example to use a custom linker, it is possible to extend the JetBrains Kotlin codebase so that a compilation for RISC-V targets is possible. In addition, all changes are manageable and can be adopted for other targets. However, direct integration is currently not possible, as dependencies such as the cross-compile toolchain must be provided by JetBrains itself. In addition, the linker is determined by the LLVM distribution and is not a separate dependent component that can be load.

Furthermore, the work shows which additional steps would be necessary to update the LLVM distribution. The changes that are presented in Chapter 4.8 give a broad insight about the scope and the impact of increasing the LLVM version and changing it in general. It turns out that there are more changes than originally expected.

Sergey Bogolepov, one of Kotlin/Natives compiler engineers stated in a thread that: "Updating LLVM is a pretty big task, and it is mostly about technical debt rather than immediate user-visible improvements of Kotlin/Native, so it is always a trade-off. We might start updating LLVM more often, but no promises." [100]

This confirms the magnitude of the challenge of this work and also shows that Kotlin/Native will be promoted in the future.

Not all problems that occurred are resolved completely, nor a complete strategy how all problems can be solved is presented. A more detailed examination of this matter is considered to deviate too far from the core objective and is therefore not undertaken. It is also important to consider the extent to which the changes remain clear. Many sub-steps are only made possible by temporary modifications, so it must also be considered whether the temporary modifications are functional or whether they may cause further problems.

Chapter 5 has shown, that the build of the riscv-gnu-toolchain on the macOS Intel host also turned out to be a complicated task. It can be said, that this is possible on a Linux target, which is not described further here as it does not contribute to the result of the work.

It has to be said also, that the project of David Burela is not the only option to set up a test environment. However, it offers a quick solution for the

circumstances and is sufficient for a first attempt. The Linux image could also be build within a docker image or a virtual machine and then be used directly.

7.2 Outlook

During the work on this thesis, some compromises had to be made. The following open tasks can be pursued by the community or JetBrains in the future.

As Chapter 4.8 has shown, the signal handling will change with the new LLVM version and needs to be updated. The configurations are not updated to make use of `c++17`, but this will be recommended when using a higher LLVM version. The current usage of the LLVM pass manager has to be replaced with the new LLVM pass manager. Invocations of methods and handling the return values will change. The same applies to the changes to make use of opaque pointers in the LLVM-C library. Besides these necessary changes, it is also beneficial to update the configurations for the `toolchain_builder`, and to add a configuration for RISC-V to provide a reusable and project bound mechanism to provide the toolchains. In addition, the changes presented are only implemented targeting an Intel-based macOS as host. All changes must also be tested for the other host systems, Macos Arm, Linux and Windows.

Besides the open tasks of this work, further research and reviews can be done. Based on this work, it needs to be investigated whether the test suite of the Kotlin/Native compiler can also be executed on real RISC-V targets. Moreover, it will be a great addition if the Kotlin/Native tests could also be run on RISC-V emulators so that they can be integrated into the CI/CD. Since the compiler was tested on the basis of the K1 compiler frontend, it should be asserted that the Kotlin/Native, targeting RISC-V boards, has no issues with the new K2 compiler frontend.

The compilation for native targets is currently still in the starting blocks for Kotlin/Native. It became apparent throughout the work that it would be helpful if it is possible to configure the native target development further in the future. A simple configuration of the native targets is not given and for a change of the target CPU the whole Kotlin/Native compiler has to be rebuilt. This is the only way to make essential components such as the Kotlin/Native Runtime available for the target architecture. This limitation poses a particular challenge for the development of different targets with different ISAs. To be competitive with other programming languages, it would be helpful to work on a solution that improves the selection of targets and implements a flexible injection strategy to provide native functions.

Once the changes have been made for RISC-V , other platforms besides Linux can also be considered. It has to be named also, that only Linux on RISC-V CPUs are enabled, but not the Android NDK. Enabling the Android NDK for RISC-V CPUs is going to change the future, since Android OEMs can make use of new processors and the Android OS can be spread even further.

Another improvement that can be made is to analyze and to enhance Kotlin/Native's optimization and compilation process. As described in Chapter 2.2, some modules are lowered and merged in the compiler backend. The

resulting LLVM module is quite large, and further optimization possibilities are conceivable. These could be investigated and implemented. This would result in smaller libraries or executables.

7.3 Implications and relevance of the thesis for science and practice

As a part of this, this work represents a further step in the development of embedded systems using the Kotlin programming language. It also represents the first step in the interaction between RISC-V and Kotlin. Eventually, in a long term perspective, this will provide developers with a wider range of boards in the future. It also allows developers to take full advantage of Kotlin's syntax and features. It may result in fewer errors during development and makes it easier to read and maintain the code.

In addition, this work can be used as a reference to make changes to the Kotlin compiler in the future. The code examples and the explanations of the Kotlin domain-specific classes and concepts make it easier to get started and solve potential questions. With regard to the general porting of a language for RISC-V targets, it can be seen that this leads to fewer challenges when using the LLVM toolchain.

7.4 Editorial

During this work, several new versions of the Kotlin repository were released. At the beginning of this work, `v1.9.21` was the latest version. In the meantime, the research results were transferred to `v2.0.0-Beta2` and finally to `v2.0.0-Beta5`. Accordingly, there is a possibility that the specified code changes may differ from the latest Kotlin version.

Gradle task 8.
llvmInteropStubsClasses

For example, the directory and therefore the module of the LLVM interop generation has changed. The gradle task to generate the LLVM interop stubs has changed to gradle task 8. This is part of commit `bce550d` in March 2024. [101]

In addition, the implementation of a LLVM bump was initiated in April 2024 with commit `87c043e` on branch `ssa/update-llvm`. Accordingly, similar changes described in Chapter 4.8 are currently being implemented by the JetBrains compiler team.[102]

Acronyms

ADD ADD Instruction. 11

ADDI ADD Immediate. 11, 47

ANDI AND Immediate. 11

AOT Ahead Of Time. 47

AST Abstract Syntax Tree. 6

AUIPC ADD Upper Immediate to PC. 11, 12, 44, 45, 47

CFG Control Flow Graph. 5

CPSR Current Program Status Register. 42

CPU Central Processing Unit. 1, 10, 23, 24, 39, 50

FBD Function Block Diagram. 39

FPGA Field Programmable Gate Array. 39, 40

IEC International Electrotechnical Commission. 39

IL Instruction List. 39

IR Intermediate Representation. 3–8, 38, 41–44, 47, 48

ISA Instruction Set Architecture. 1, 3, 9, 10, 12, 23, 39–42, 50, II

JAL Jump and Link. 11, 12

JALR Jump And Link Register. 11, 44, 47

JIT Just in Time. 41, 47, 48

JNI Java Native Interfaces. 9, 10

JVM Java Virtual Machine. 1, 6, 9, 10, 21, 28, 31

Kotlin IR Kotlin Intermediate Representation. 9, 17, 18, 47

LD Ladder Diagram. 39

LI Load Immediate. 47

LLVM IR LLVM Intermediate Representation. 3–6, 8, 9, 12, 15, 18, 38–41, 47, II

LTO Link Time Optimization. 6

LUI Load Upper Immediate. 11, 47

MIF Memory Initialization File. 39

ORI OR Immediate. 11

PC Program Counter. 11, 45

PLC Programmable Logic Controller. 38–40

PSI Program Structure Interface. 6–8, 47

SFC Sequential Function Chart. 39

SLLI Shift Logical Left Immediate. 47

SLTI Set Less Than Immediate. 11

SSA Static Single Assignment. 4

ST Structured Text. 39

XORI XOR Immediate. 11

Glossary

Clang-C Clang C API. 9, 14, 15, 18, 19, 29

LLVM-C LLVM C API. 4, 5, 9, 10, 14, 15, 18, 19, 22, 32, 50

RISC-V An open source and free Instruction Set Architecture. 1–3, 8–10, 12–18, 20, 22–26, 28, 35, 36, 38–44, 47, 49–51, II

C++ Programming language. 4, 7, 9, 17

GCC GNU Compiler Collection. 15, 17

Konan The name of the Kotlin/Native compiler. 6–8, 14

LLVM LLVM. 1–10, 12–15, 17–20, 22–25, 27–32, 34, 38, 39, 41, 47, 49–51, II, III

A File references

Filename	Path	Comment
filename.kexe		Final build native executable
filename.ir		Generated
package.py	kotlin-native/tools/llvm_builder/package.py	v.2.0.0-Beta2
OptimizationPipeline.kt	kotlin-native/backend.native/compiler/ir/backend.native/src/org/jetbrains/kotlin/backend/konan/OptimizationPipeline.kt	v.2.0.0-Beta2
KonanConfig.kt	kotlin-native/backend.native/compiler/ir/backend.native/src/org/jetbrains/kotlin/backend/konan/KonanConfig.kt	v.2.0.0-Beta2
local.properties	local.properties	v.2.0.0-Beta2
gradle.properties	gradle.properties	v.2.0.0-Beta2
Architecture.kt	native/utils/src/org/jetbrains/kotlin/konan/target/Architecture.kt	v.2.0.0-Beta2
zlib.def	kotlin-native/platformLibs/src/platform/linux	v.2.0.0-Beta2
posix.def	kotlin-native/platformLibs/src/platform/linux	v.2.0.0-Beta2
linux.def	kotlin-native/platformLibs/src/platform/linux	v.2.0.0-Beta2
iconv.def	kotlin-native/platformLibs/src/platform/linux	v.2.0.0-Beta2
builtin.def	kotlin-native/platformLibs/src/platform/linux	v.2.0.0-Beta2
konan.properties	kotlin-native/konan/konan.properties	v.2.0.0-Beta2

Continued on next page

Table 6 – continued from previous page

Filename	Path	Comment
clang.kt	kotlin-native/Interop/Indexer/prebuilt/nativeInteropStubs/kotlin/clang/clang.kt	Generated
clangstubs.c	kotlin-native/Interop/Indexer/prebuilt/nativeInteropStubs/c/clangstubs.c	Generated
llvm.kt	kotlin-native/llvmInterop/build/nativeInteropStubs/llvm/kotlin/llvm/llvm.kt	Generated
llvm.def	kotlin-native/llvmInterop/llvm.def	v.2.0.0-Beta5
llvm.def	kotlin-native/backend.native/llvm.def	v.2.0.0-Beta2
KonanTarget-Extensions.kt	native/utils/src/org/jetbrains/kotlin/konan/target/KonanTargetExtenstions.kt	v.2.0.0-Beta2 (Removed on Beta5)
CAPIExtensions.cpp	kotlin-native/libllvmext/src/main/cpp/CAPIExtensions.cpp	v.2.0.0-Beta2
ThreadSuspension.cpp	kotlin-native/runtime/src/mm/cpp/ThreadSuspension.cpp	v.2.0.0-Beta2
SafePoint.cpp	kotlin-native/runtime/src/mm/cpp/SafePoint.cpp	v.2.0.0-Beta2
checked.h	kotlin-native/runtime/src/main/cpp/utf8/checked.h	v.2.0.0-Beta2
unchecked.h	kotlin-native/runtime/src/main/cpp/utf8/unchecked.h	v.2.0.0-Beta2
common.h	kotlin-native/runtime/src/main/cpp/Common.h	v.2.0.0-Beta2
clang.def	kotlin-native/Interop/Indexer/clang.def	v.2.0.0-Beta2

Continued on next page

Table 6 – continued from previous page

Filename	Path	Comment
CoverageMappingC.cpp	`kotlin-native/libllvmext/src/main/cpp/CoverageMappingC.cpp`	v.2.0.0-Beta2 (Removed on Beta5)
Logging.hpp	`kotlin-native/runtime/src/main/cpp/Logging.hpp`	v.2.0.0-Beta2
MappingBridge-GeneratorImpl.kt	`kotlin-native/Interop/StubGenerator/src/org/jetbrains/kotlin/native/interop/gen/MappingBridgeGeneratorImpl.kt`	v.2.0.0-Beta2
build.gradle.kts	`kotlin-native/Interop/Indexer/build.gradle.kts`	Builds C-Interop of Clang
build.gradle.kts	`kotlin-native/llvmInterop/build.gradle.kts`	Builds C-Interop of LLVM
signal.h		Part of the C-library
signal-macros.h		Part of the C-library
xlocale.h		Part of the C-library
locale.h		Part of the C-library
IPO.h		Part of the C-library
Initialization.h		Part of the C-library
runtime.bc	`kotlin-native/runtime/build/bitcode/main/linux\_arm64/runtime.bc`	Generated for each target
compiler_interface.bc	`kotlin-native/runtime/build/bitcode/main/linux\_arm64/compiler\_interface.bc`	Generated for each target
ld.lld		Part of LLVM
elf.h		Part of LLVM
Compression.cpp		Part of LLVM
Compression.cpp.o		Part of LLVM

Continued on next page

Table 6 – continued from previous page

Filename	Path	Comment

Table 6: Used and referenced files

B Code sample

B.1 Kotlin/Native sample code

```
package thesis.saatze

import printer.Printer

fun main() {
    Printer().also {
        it.sayHello()
    }
}
```

Listing 32: Example Kotlin program

B.2 Kotlin/Native sample PSI tree

```
KtFile: Main.kt(0,112)
  PACKAGE_DIRECTIVE(0,21)
  PsiElement(package)('package')(0,7)
  DOT_QUALIFIED_EXPRESSION(8,21)
    REFERENCE_EXPRESSION(8,14)
    PsiElement(IDENTIFIER)('thesis')(8,14)
    PsiElement(DOT)('.')(14,15)
    REFERENCE_EXPRESSION(15,21)
    PsiElement(IDENTIFIER)('saatze')(15,21)
  IMPORT_LIST(23,45)
  IMPORT_DIRECTIVE(23,45)
    PsiElement(import)('import')(23,29)
    DOT_QUALIFIED_EXPRESSION(30,45)
    REFERENCE_EXPRESSION(30,37)
      PsiElement(IDENTIFIER)('printer')(30,37)
    PsiElement(DOT)('.')(37,38)
    REFERENCE_EXPRESSION(38,45)
      PsiElement(IDENTIFIER)('Printer')(38,45)
```

Listing 33: Program Structure Interface tree representation, 1/2

```
FUN(47,110)
PsiElement(fun)('fun')(47,50)
PsiElement(IDENTIFIER)('main')(51,55)
VALUE_PARAMETER_LIST(55,57)
  PsiElement(LPAR)('(')(55,56)
  PsiElement(RPAR)(')')(56,57)
BLOCK(58,110)
  PsiElement(LBRACE)('{')(58,59)
  DOT_QUALIFIED_EXPRESSION(64,108)
  CALL_EXPRESSION(64,73)
    REFERENCE_EXPRESSION(64,71)
    PsiElement(IDENTIFIER)('Printer')(64,71)
    VALUE_ARGUMENT_LIST(71,73)
    PsiElement(LPAR)('(')(71,72)
    PsiElement(RPAR)(')')(72,73)
  PsiElement(DOT)('.')(73,74)
  CALL_EXPRESSION(74,108)
    REFERENCE_EXPRESSION(74,78)
    PsiElement(IDENTIFIER)('also')(74,78)
    LAMBDA_ARGUMENT(79,108)
    LAMBDA_EXPRESSION(79,108)
      FUNCTION_LITERAL(79,108)
      PsiElement(LBRACE)('{')(79,80)
      BLOCK(89,102)
        DOT_QUALIFIED_EXPRESSION(89,102)
        REFERENCE_EXPRESSION(89,91)
          PsiElement(IDENTIFIER)('it')(89,91)
        PsiElement(DOT)('.')(91,92)
        CALL_EXPRESSION(92,102)
          REFERENCE_EXPRESSION(92,100)
          PsiElement(IDENTIFIER)('sayHello')(92,100)
          VALUE_ARGUMENT_LIST(100,102)
          PsiElement(LPAR)('(')(100,101)
          PsiElement(RPAR)(')')(101,102)
      PsiElement(RBRACE)('}')(107,108)
  PsiElement(RBRACE)('}')(109,110)
```

Listing 34: Program Structure Interface tree representation, 2/2

B.3 Kotlin IR sample

```
// --- IR for Main.kt after Redundant coercions cleaning
FILE fqName:<root> fileName:$REPO_ROOT/
↪  kotlin-sample-code/src/nativeMain/kotlin/Main.kt
  FUN name:main visibility:public modality:FINAL <> ()
  ↪  returnType:kotlin.Unit
  BLOCK_BODY
    CALL 'public final fun println (message: kotlin.Any?):
    ↪  kotlin.Unit declared in kotlin.io' type=kotlin.Unit
    ↪  origin=null
    message: CONST String type=kotlin.String value="Hello,
    ↪  Kotlin/Native!"
    RETURN type=kotlin.Nothing from='public final fun main ():
    ↪  kotlin.Unit declared in <root>'
    CALL 'internal final fun theUnitInstance (): kotlin.Unit
    ↪  [external] declared in kotlin.native.internal'
    ↪  type=kotlin.Unit origin=null
  FUN ENTRY_POINT name:Konan_start visibility:private
  ↪  modality:FINAL <> (args:kotlin.Array<kotlin.String>)
  ↪  returnType:kotlin.Int
  annotations:
    ExportForCppRuntime(name = "Konan_start")
  VALUE_PARAMETER ENTRY_POINT name:args index:0
  ↪  type:kotlin.Array<kotlin.String>
```

Listing 35: Kotlin IR of Main.kt, 1/2

```
BLOCK_BODY
  TRY type=kotlin.Nothing
  try: BLOCK type=kotlin.Nothing origin=null
    CALL 'public final fun main (): kotlin.Unit declared in
    ↪  <root>' type=kotlin.Unit origin=null
    RETURN type=kotlin.Nothing from='private final fun
    ↪  Konan_start (args: kotlin.Array<kotlin.String>):
    ↪  kotlin.Int declared in <root>'
    CONST Int type=kotlin.Int value=0
  CATCH parameter=val e: kotlin.Throwable [val] declared in
  ↪  <root>.Konan_start
    VAR CATCH_PARAMETER name:e type:kotlin.Throwable [val]
    BLOCK type=kotlin.Nothing origin=null
    CALL 'public final fun processUnhandledException
    ↪  (throwable: kotlin.Throwable): kotlin.Unit [external]
    ↪  declared in kotlin.native' type=kotlin.Unit origin=null
      throwable: GET_VAR 'val e: kotlin.Throwable [val]
      ↪  declared in <root>.Konan_start' type=kotlin.Throwable
      ↪  origin=null
    BLOCK type=kotlin.Nothing origin=null
      CALL 'public final fun terminateWithUnhandledException
      ↪  (throwable: kotlin.Throwable): kotlin.Nothing
      ↪  [external] declared in kotlin.native'
      ↪  type=kotlin.Nothing origin=null
      throwable: GET_VAR 'val e: kotlin.Throwable [val]
      ↪  declared in <root>.Konan_start' type=kotlin.Throwable
      ↪  origin=null
      CALL 'internal final fun ThrowKotlinNothingValueException
      ↪  (): kotlin.Nothing declared in
      ↪  kotlin.native.internal' type=kotlin.Nothing
      ↪  origin=null
```

Listing 36: Kotlin IR of Main.kt, 2/2

C Command line execution

C.1 Linker invocation

```
$KONAN_HOME/dependencies/apple-llvm-20200714-macos-x64-essentials/
↪ bin/ld.lld
↪ --sysroot=$KONAN_HOME/dependencies/riscv64-glibc-ubuntu-20.04-gcc-
↪ nightly -2023.12.14-nightly/sysroot -export-dynamic -z relro
↪ --build-id --eh-frame-hdr -dynamic-linker
↪ /lib/ld-linux-riscv64-lp64d.so.1 --verbose -o
↪ $REPO_ROOT/program.kexe
↪ $KONAN_HOME/dependencies/riscv64-glibc-ubuntu-20.04-gcc-
↪ nightly -2023.12.14-nightly/sysroot/usr/lib/crt1.o
↪ $KONAN_HOME/dependencies/riscv64-glibc-ubuntu-20.04-gcc-
↪ nightly -2023.12.14-nightly/sysroot/usr/lib/crti.o
↪ $KONAN_HOME/dependencies/riscv64-glibc-ubuntu-20.04-gcc-
↪ nightly -2023.12.14-nightly/sysroot/../lib/gcc/
↪ riscv64-unknown-linux-gnu/13.2.0/crtbegin.o
↪ -L$KONAN_HOME/dependencies/ riscv64-glibc-ubuntu-20.04-gcc-
↪ nightly -2023.12.14-nightly/sysroot/../lib/gcc/
↪ riscv64-unknown-linux-gnu/13.2.0 --hash-style=gnu
↪ -L$KONAN_HOME/dependencies/riscv64-glibc-ubuntu-20.04-gcc-nightly
↪ -2023.12.14-nightly/sysroot/lib -L$KONAN_HOME/dependencies/
↪ riscv64-glibc-ubuntu-20.04-gcc- nightly
↪ -2023.12.14-nightly/sysroot/usr/lib -S
↪ $TMP_PATH/konan_temp10134404706552790 59/program.kexe.o
↪ -lstdc++ -Bdynamic -ldl -lm -lpthread --defsym
↪ __cxa_demangle=Konan_cxa_demangle -lgcc --as-needed -lgcc_s
↪ --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed
↪ $KONAN_HOME/dependencies/riscv64-glibc-ubuntu-20.04-gcc-
↪ nightly -2023.12.14-nightly/sysroot/../lib/gcc/
↪ riscv64-unknown-linux-gnu/13.2.0/crtend.o
↪ $KONAN_HOME/dependencies/ riscv64-glibc-ubuntu-20.04-gcc-
↪ nightly -2023.12.14-nightly/sysroot/usr/lib/crtn.o
```

Listing 37: Linker invocation during linking phase

C.2 Clang invocation for ThreadSuspension.cpp

```
-c
-emit-llvm
-I$REPO_ROOT/kotlin-native/runtime/src/mm/cpp
-I$REPO_ROOT/kotlin-native/runtime/src/alloc/common/cpp
-I$REPO_ROOT/kotlin-native/runtime/src/gcScheduler/common/cpp
-I$REPO_ROOT/kotlin-native/runtime/src/gc/common/cpp
-I$REPO_ROOT/kotlin-native/runtime/src/externalCallsChecker/common/cpp
-I$REPO_ROOT/kotlin-native/runtime/src/objcExport/cpp
-I$REPO_ROOT/kotlin-native/runtime/src/main/cpp
-std=c++17
-Werror
-O2
-fno-aligned-allocation
-Wall
-Wextra
-Wno-unused-parameter
mm/cpp/ThreadSuspension.cpp
-o
$REPO_ROOT/kotlin-native/runtime/build/
↪  bitcode/main/linux_riscv64/mm/ThreadSuspension.bc
-B$HOME/.konan/dependencies/apple-llvm-20200714-macos-x64-1/bin
-fno-stack-protector
--gcc-toolchain=$HOME/.konan/dependencies/
↪  riscv64-glibc-ubuntu-20.04-gcc-nightly-2023.12.14-nightly
-target
riscv64-unknown-linux-gnu
--sysroot=$HOME/.konan/dependencies/
↪  riscv64-glibc-ubuntu-20.04-gcc-nightly-2023.12.14-nightly/sysroot
-fPIC
-DKONAN_RISCV64=1
-DKONAN_LINUX=1
-DKONAN_FORBID_BUILTIN_MUL_OVERFLOW=1
-DUSE_ELF_SYMBOLS=1
-DELFSIZE=64
-DUSE_GCC_UNWIND=1
```

Listing 38: Clang invocation for ThreadSuspension.cpp

D Error messages

D.1 Error message on clang compilation of ThreadSuspension.cpp

```
mm/cpp/ThreadSuspension.cpp:20:41: error: no template named
↪  'optional' in namespace 'std'
[[clang::no_destroy]] thread_local
↪  std::optional<mm::SafePointActivator> gSafePointActivator =
↪  std::nullopt;
                                      ~~~~~~
mm/cpp/ThreadSuspension.cpp:20:101: error: no member named
↪  'nullopt' in namespace 'std'
[[clang::no_destroy]] thread_local
↪  std::optional<mm::SafePointActivator> gSafePointActivator =
↪  std::nullopt;


mm/cpp/ThreadSuspension.cpp:86:47: error: no member named
↪  'nullopt' in namespace 'std'
    RuntimeAssert(gSafePointActivator == std::nullopt, "Current
    ↪  thread already suspended threads.");
                                              ~~~~~~
$REPO_ROOT/kotlin-native/runtime/src/main/cpp/KAssert.h:52:23:
↪  note: expanded from macro 'RuntimeAssert'
                if (!(condition)) { \
                      ^~~~~~~~~
mm/cpp/ThreadSuspension.cpp:86:47: error: no member named
↪  'nullopt' in namespace 'std'
    RuntimeAssert(gSafePointActivator == std::nullopt, "Current
    ↪  thread already suspended threads.");
                                              ~~~~~~
$REPO_ROOT/kotlin-native/runtime/src/main/cpp/KAssert.h:57:23:
↪  note: expanded from macro 'RuntimeAssert'
                if (!(condition)) { \
                      ^~~~~~~~~
mm/cpp/ThreadSuspension.cpp:110:47: error: no member named
↪  'nullopt' in namespace 'std'
    RuntimeAssert(gSafePointActivator != std::nullopt, "Current
    ↪  thread must have suspended threads");
                                              ~~~~~~
```

Listing 39: Error Message when command of Listing 38 is called, 1/2

```
$REPO_ROOT/kotlin-native/runtime/src/main/cpp/KAssert.h:52:23:
↪  note: expanded from macro 'RuntimeAssert'
                if (!(condition)) { \
                      ^~~~~~~~~
mm/cpp/ThreadSuspension.cpp:110:47: error: no member named
↪  'nullopt' in namespace 'std'
    RuntimeAssert(gSafePointActivator != std::nullopt, "Current
    ↪  thread must have suspended threads");
                                              ~~~~~~
$REPO_ROOT/kotlin-native/runtime/src/main/cpp/KAssert.h:57:23:
↪  note: expanded from macro 'RuntimeAssert'
                if (!(condition)) { \
                      ^~~~~~~~~
mm/cpp/ThreadSuspension.cpp:111:32: error: no member named
↪  'nullopt' in namespace 'std'
    gSafePointActivator = std::nullopt;
                          ~~~~~~
mm/cpp/SafePoint.cpp:69:10:
error
: no member named 'optional' in namespace 'std'
    std::optional<SafePointSignpostInterval> signpost;
    ~~~~~~
mm/cpp/SafePoint.cpp:69:19: error: 'SafePointSignpostInterval'
↪  does not refer to a value
    std::optional<SafePointSignpostInterval> signpost;
                  ^
mm/cpp/SafePoint.cpp:58:7: note: declared here
class SafePointSignpostInterval : private Pinned {
      ^
mm/cpp/SafePoint.cpp:69:46:
error: use of undeclared identifier 'signpost'
    std::optional<SafePointSignpostInterval> signpost;
                                             ^
mm/cpp/SafePoint.cpp:71:9:
error: use of undeclared identifier 'signpost'
        signpost.emplace(threadData);
        ^
```

Listing 40: Error Message when command of Listing 38 is called, 2/2

D.2 Error message during compiler build

```
$REPO_ROOT/kotlin-native/runtime/src/main/cpp/utf8/checked.h:282:34:
↪  error: 'iterator<std::bidirectional_iterator_tag, unsigned
↪  int>' is deprecated [-Werror,-Wdeprecated-declarations]
    class iterator : public std::iterator
    ↪  <std::bidirectional_iterator_tag, uint32_t> {
                                    ^
$HOME/.konan/dependencies/
↪  riscv64-glibc-ubuntu-20.04-gcc-nightly-2023.12.14-nightly/lib/gcc/
↪  riscv64-unknown-linux-gnu/13.2.0/../../../../
↪  riscv64-unknown-linux-gnu/include/c++/13.2.0/
↪  bits/stl_iterator_base_types.h:127:12: note:
↪  'iterator<std::bidirectional_iterator_tag, unsigned int>' has
↪  been explicitly marked deprecated here
    struct _GLIBCXX17_DEPRECATED iterator
           ^
$HOME/.konan/dependencies/
↪  riscv64-glibc-ubuntu-20.04-gcc-nightly-2023.12.14-nightly/lib/gcc/
↪  riscv64-unknown-linux-gnu/13.2.0/../../../../
↪  riscv64-unknown-linux-gnu/include/c++/13.2.0/
↪  riscv64-unknown-linux-gnu/bits/c++config.h:121:34: note:
↪  expanded from macro '_GLIBCXX17_DEPRECATED'
# define _GLIBCXX17_DEPRECATED [[__deprecated__]]
                                    ^
```

Listing 41: Error Message when std:iterator is used

D.3 Incompatible integer to pointer conversion

```
    $REPO_ROOT/kotlin-native/Interop/Indexer/prebuilt/
    ↪  nativeInteropStubs/c/clangstubs.c:151:12: error:
    ↪  incompatible integer to pointer conversion passing
    ↪  'jlong' (aka 'long') to parameter of type 'void *'
    ↪  [-Wint-conversion]
  151 |     memcpy(p1, &kniStructResult,
   ↪  sizeof(kniStructResult));
      |            ^~
$MACOS_SDK/MacOSX14.4.sdk/usr/include/secure/_string.h:63:27:
↪  note: expanded from macro 'memcpy'
   63 |                 __builtin___memcpy_chk (dest,
    ↪  __VA_ARGS__, __darwin_obsz0 (dest))
      |                                         ^~~~
```

Listing 42: Error message when jlong is cast automatically

D.4 Different macOS target version of LLVM libraries and target compilation

```
ld: warning: object file
↪  ($REPO_ROOT/kotlin-native/tools/llvm_builder/build/
↪  lib/libclangAST.a[2](APValue.cpp.o)) was built for newer
↪  'macOS' version (14.0) than being linked (11.0)
ld: warning: object file
↪  ($REPO_ROOT/kotlin-native/tools/llvm_builder/build/
↪  lib/libclangBasic.a[2](Attributes.cpp.o)) was built for newer
↪  'macOS' version (14.0) than being linked (11.0)
ld: warning: object file
↪  ($REPO_ROOT/kotlin-native/tools/llvm_builder/build/
↪  lib/libLLVMMC.a[56](MCTargetOptions.cpp.o)) was built for
↪  newer 'macOS' version (14.0) than being linked (11.0)
ld: warning: object file
↪  ($REPO_ROOT/kotlin-native/tools/llvm_builder/build/
↪  lib/libLLVMTransformUtils.a[6](BasicBlockUtils.cpp.o)) was
↪  built for newer 'macOS' version (14.0) than being linked
↪  (11.0)
ld: warning: object file
↪  ($REPO_ROOT/kotlin-native/tools/llvm_builder/build/
↪  lib/libLLVMCore.a[2](AbstractCallSite.cpp.o)) was built for
↪  newer 'macOS' version (14.0) than being linked (11.0)
```

Listing 43: Libraries and currently build target differ in macOS version

D.5 ZSTD library cannot be found

```
      Undefined symbols for architecture x86_64:
"_ZSTD_compress", referenced from:
    __ZN4llvm11compression8compressENS0_6ParamsENS_
    ↪  8ArrayRefIhEERNS_15SmallVectorImplIhEE in
    ↪  libLLVMSupport.a[37](Compression.cpp.o)
    __ZN4llvm11compression4zstd8compressENS_8ArrayRefIhEERNS_
    ↪  15SmallVectorImplIhEEi in
    ↪  libLLVMSupport.a[37](Compression.cpp.o)
"_ZSTD_compressBound", referenced from:

    ↪  __ZN4llvm11compression8compressENS0_6ParamsENS_8ArrayRefIhEERNS_
    ↪  15SmallVectorImplIhEE in
    ↪  libLLVMSupport.a[37](Compression.cpp.o)
    __ZN4llvm11compression4zstd8compressENS_8ArrayRefIhEERNS_
    ↪  15SmallVectorImplIhEEi in
    ↪  libLLVMSupport.a[37](Compression.cpp.o)
"_ZSTD_decompress", referenced from:

    ↪  __ZN4llvm11compression10decompressENS_20DebugCompressionTypeENS_
    ↪  8ArrayRefIhEEPhm in
    ↪  libLLVMSupport.a[37](Compression.cpp.o)

    ↪  __ZN4llvm11compression4zstd10decompressENS_8ArrayRefIhEEPhRm
    ↪  in libLLVMSupport.a[37](Compression.cpp.o)

    ↪  __ZN4llvm11compression4zstd10decompressENS_8ArrayRefIhEERNS_
    ↪  15SmallVectorImplIhEEm in
    ↪  libLLVMSupport.a[37](Compression.cpp.o)
"_ZSTD_getErrorName", referenced from:

    ↪  __ZN4llvm11compression10decompressENS_20DebugCompressionTypeENS_
    ↪  8ArrayRefIhEEPhm in
    ↪  libLLVMSupport.a[37](Compression.cpp.o)

    ↪  __ZN4llvm11compression4zstd10decompressENS_8ArrayRefIhEEPhRm
    ↪  in libLLVMSupport.a[37](Compression.cpp.o)

    ↪  __ZN4llvm11compression4zstd10decompressENS_8ArrayRefIhEERNS_
    ↪  15SmallVectorImplIhEEm in
    ↪  libLLVMSupport.a[37](Compression.cpp.o)
```

Listing 44: ZSTD symbols cannot be found, 1/2

```
  "_ZSTD_isError", referenced from:

      ↪  __ZN4llvm11compression8compressENS0_6ParamsENS_8ArrayRefIhEERNS_
      ↪  15SmallVectorImplIhEE in
      ↪  libLLVMSupport.a[37](Compression.cpp.o)
     __ZN4llvm11compression4zstd8compressENS_8ArrayRefIhEERNS_
      ↪  15SmallVectorImplIhEEi in
      ↪  libLLVMSupport.a[37](Compression.cpp.o)

      ↪  __ZN4llvm11compression10decompressENS_20DebugCompressionTypeENS_
      ↪  8ArrayRefIhEEPhm in
      ↪  libLLVMSupport.a[37](Compression.cpp.o)

      ↪  __ZN4llvm11compression4zstd10decompressENS_8ArrayRefIhEEPhRm
      ↪  in libLLVMSupport.a[37](Compression.cpp.o)

      ↪  __ZN4llvm11compression4zstd10decompressENS_8ArrayRefIhEERNS_
      ↪  15SmallVectorImplIhEEm in
      ↪  libLLVMSupport.a[37](Compression.cpp.o)
ld: symbol(s) not found for architecture x86_64
```

Listing 45: ZSTD symbols cannot be found, 2/2

D.6 Java Runtime can not handle signal

```
#
# A fatal error has been detected by the Java Runtime
↪  Environment:
#
#  SIGBUS (0xa) at pc=0x00000001414b47e2, pid=6325, tid=8707
#
# JRE version: OpenJDK Runtime Environment Temurin-17.0.7+7
↪  (17.0.7+7) (build 17.0.7+7)
# Java VM: OpenJDK 64-Bit Server VM Temurin-17.0.7+7 (17.0.7+7,
↪  mixed mode, sharing, tiered, compressed oops, compressed
↪  class ptrs, g1 gc, bsd-amd64)
# Problematic frame:
# C  [libclangstubs.dylib+0x97e2]  initSignalChaining()+0x162
#
# No core dump will be written. Core dumps have been disabled. To
↪  enable core dumping, try "ulimit -c unlimited" before
↪  starting Java again
#
# An error report file with more information is saved as:
# $REPO_ROOT/kotlin-native/Interop/Indexer/hs_err_pid6325.log
#
# If you would like to submit a bug report, please visit:
#   https://github.com/adoptium/adoptium-support/issues
# The crash happened outside the Java Virtual Machine in native
↪  code.
# See problematic frame for where to report the bug.
#
```

Listing 46: Error when using the updated LLVM 17 distribution

D.7 Environment OS version flag has already been defined

```
> Task :kotlin-native:common:clangFrontendFilesMainMacos_x64
↪  FAILED
In file included from <built-in>:469:
<command line>:1:9: error:
↪  '__ENVIRONMENT_OS_VERSION_MIN_REQUIRED__' macro redefined
↪  [-Werror,-Wmacro-redefined]
    1 | #define __ENVIRONMENT_OS_VERSION_MIN_REQUIRED__
    ↪  __ENVIRONMENT_MAC_OS_X_VERSION_MIN_REQUIRED__
      |         ^
<built-in>:460:9: note: previous definition is here
  460 | #define __ENVIRONMENT_OS_VERSION_MIN_REQUIRED__ 140000
      |         ^
1 error generated.
```

Listing 47: Error when redefining built-in flags

D.8 LLVM legacy pass manager functions were removed

```
Task :kotlin-native:backend.native:compileKotlin FAILED
CompilerOutput.kt:157:31 Unresolved reference
↪  'LLVMGetProgramAddressSpace'.
OptimizationPipeline.kt:177:102 Unresolved reference
↪  'LLVMPassManagerBuilderRef'.
OptimizationPipeline.kt:201:27 Unresolved reference
↪  'LLVMPassManagerBuilderCreate'.
OptimizationPipeline.kt:204:13 Unresolved reference
↪  'LLVMPassManagerBuilderSetOptLevel'.
OptimizationPipeline.kt:205:13 Unresolved reference
↪  'LLVMPassManagerBuilderSetSizeLevel'.
OptimizationPipeline.kt:207:17 Unresolved reference
↪  'LLVMPassManagerBuilderUseInlinerWithThreshold'.
OptimizationPipeline.kt:236:13 Unresolved reference
↪  'LLVMPassManagerBuilderDispose'.
OptimizationPipeline.kt:258:32 Unresolved reference
↪  'LLVMGetGlobalPassRegistry'.
OptimizationPipeline.kt:260:13 Unresolved reference
↪  'LLVMInitializeCore'.
OptimizationPipeline.kt:261:13 Unresolved reference
↪  'LLVMInitializeTransformUtils'.
OptimizationPipeline.kt:262:13 Unresolved reference
↪  'LLVMInitializeScalarOpts'.
OptimizationPipeline.kt:263:13 Unresolved reference
↪  'LLVMInitializeVectorization'.
OptimizationPipeline.kt:264:13 Unresolved reference
↪  'LLVMInitializeInstCombine'.
```

Listing 48: Reduced error message for missing LLVM legacy pass manager functions, 1/3

```
OptimizationPipeline.kt:265:13 Unresolved reference
↪  'LLVMInitializeIPO'.
OptimizationPipeline.kt:266:13 Unresolved reference
↪  'LLVMInitializeInstrumentation'.
OptimizationPipeline.kt:267:13 Unresolved reference
↪  'LLVMInitializeAnalysis'.
OptimizationPipeline.kt:268:13 Unresolved reference
↪  'LLVMInitializeIPA'.
OptimizationPipeline.kt:269:13 Unresolved reference
↪  'LLVMInitializeCodeGen'.
OptimizationPipeline.kt:270:13 Unresolved reference
↪  'LLVMInitializeTarget'.
OptimizationPipeline.kt:271:13 Unresolved reference
↪  'LLVMInitializeObjCARCOpts'.
OptimizationPipeline.kt:290:102 Unresolved reference
↪  'LLVMPassManagerBuilderRef'.
OptimizationPipeline.kt:312:102 Unresolved reference
↪  'LLVMPassManagerBuilderRef'.
OptimizationPipeline.kt:313:9 Unresolved reference
↪  'LLVMPassManagerBuilderPopulateModulePassManager'.
OptimizationPipeline.kt:314:9 Unresolved reference
↪  'LLVMPassManagerBuilderPopulateFunctionPassManager'.
OptimizationPipeline.kt:322:102 Unresolved reference
↪  'LLVMPassManagerBuilderRef'.
OptimizationPipeline.kt:324:13 Unresolved reference
↪  'LLVMAddInternalizePass'.
OptimizationPipeline.kt:328:13 Unresolved reference
↪  'LLVMAddGlobalDCEPass'.
OptimizationPipeline.kt:332:9 Unresolved reference
↪  'LLVMPassManagerBuilderPopulateLTOPassManager'.
OptimizationPipeline.kt:340:102 Unresolved reference
↪  'LLVMPassManagerBuilderRef'.
```

Listing 49: Reduced error message for missing LLVM legacy pass manager functions, 2/3

```
llvm/LlvmFunctionPrototype.kt:68:27 Unresolved reference
↪  'LLVMAttributeFunctionIndex'.
llvm/LlvmFunctionPrototype.kt:68:27 For-loop range must have an
↪  'iterator()' method.
llvm/LlvmFunctionPrototype.kt:81:52 Unresolved reference
↪  'LLVMAttributeFunctionIndex'.
llvm/LlvmFunctionPrototype.kt:89:51 Unresolved reference
↪  'LLVMAttributeFunctionIndex'.
llvm/LlvmFunctionPrototype.kt:133:61 Unresolved reference
↪  'LLVMAttributeFunctionIndex'.
llvm/LlvmFunctionPrototype.kt:134:61 Unresolved reference
↪  'LLVMAttributeReturnIndex'.
llvm/LlvmFunctionPrototype.kt:142:64 Unresolved reference
↪  'LLVMAttributeFunctionIndex'.
llvm/LlvmFunctionPrototype.kt:143:64 Unresolved reference
↪  'LLVMAttributeReturnIndex'.
llvm/LlvmUtils.kt:314:59 Unresolved reference
↪  'LLVMAttributeFunctionIndex'.
llvm/LlvmUtils.kt:352:39 Unresolved reference
↪  'LLVMAttributeFunctionIndex'.
llvm/objc/ObjCCodeGenerator.kt:74:17 No value passed for
↪  parameter 'CanThrow'.

FAILURE: Build failed with an exception.
```

Listing 50: Reduced error message for missing LLVM legacy pass manager functions, 3/3

D.9 Missing symbols in Runtime module

```
e: java.lang.IllegalStateException: struct.TypeInfo is not found
↪  in the Runtime module.
    at org.jetbrains.kotlin.backend.konan. llvm.Runtime
    ↪  .getStructType(Runtime.kt:24)
    at org.jetbrains.kotlin.backend.konan. llvm.Runtime
    ↪  .<init>(Runtime.kt:26)
    at org.jetbrains.kotlin.backend.konan. NativeGenerationState
    ↪  .runtimeDelegate$lambda$1(NativeGenerationState.kt:93)
    at kotlin.SynchronizedLazyImpl.getValue(LazyJVM.kt:74)
    at org.jetbrains.kotlin.backend.konan. NativeGenerationState
    ↪  .getRuntime(NativeGenerationState.kt:98)
    at org.jetbrains.kotlin.backend.konan.
    ↪  llvm.CodegenLlvmHelpers .getRuntime(ContextUtils.kt:408)
    at org.jetbrains.kotlin.backend.konan.
    ↪  llvm.CodegenLlvmHelpers .<init>(ContextUtils.kt:411)
    at org.jetbrains.kotlin.backend.konan. NativeGenerationState
    ↪  .llvmDelegate$lambda$2(NativeGenerationState.kt:94)
    at kotlin.SynchronizedLazyImpl.getValue(LazyJVM.kt:74)
    at org.jetbrains.kotlin.backend.konan. NativeGenerationState
    ↪  .getLlvm(NativeGenerationState.kt:99)
    at org.jetbrains.kotlin.backend.konan.
    ↪  lower.CacheInfoBuilder$build$1$1
    ↪  .visitClass(CacheInfoBuilder.kt:47)
    at org.jetbrains.kotlin.ir
    ↪  .visitors.IrElementVisitorVoid$DefaultImpls
    ↪  .visitClass(IrElementVisitorVoid.kt:44)
    at org.jetbrains.kotlin.backend.konan.
    ↪  lower.CacheInfoBuilder$build$1$1
    ↪  .visitClass(CacheInfoBuilder.kt:36)
    at org.jetbrains.kotlin.backend.konan.
    ↪  lower.CacheInfoBuilder$build$1$1
    ↪  .visitClass(CacheInfoBuilder.kt:36)
    at org.jetbrains.kotlin.ir.declarations.IrClass
    ↪  .accept(IrClass.kt:72)
    at org.jetbrains.kotlin.ir.declarations.IrFile
    ↪  .acceptChildren(IrFile.kt:34)
    at org.jetbrains.kotlin.ir.visitors.IrVisitorsKt
    ↪  .acceptChildrenVoid(IrVisitors.kt:15)
    at org.jetbrains.kotlin.backend.konan. lower.CacheInfoBuilder
    ↪  .build(CacheInfoBuilder.kt:36)
```

Listing 51: Compiler complains about missing struct in pre-built Runtime bitcode

D.10 Failure during build of cross-compiler-toolchain for RISC-V

```
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:550:5: error:
↪  '__abi_tag__' attribute only applies to structs, variables,
↪  functions, and namespaces
    _LIBCPP_INLINE_VISIBILITY
    ^
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__config:891:37: note:
↪  expanded from macro '_LIBCPP_INLINE_VISIBILITY'
#  define _LIBCPP_INLINE_VISIBILITY _LIBCPP_HIDE_FROM_ABI
                                     ^
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__config:870:26: note:
↪  expanded from macro '_LIBCPP_HIDE_FROM_ABI'

          ↪  __attribute__((__abi_tag__(_LIBCPP_TOSTRING(_LIBCPP_ODR_SIGNATURE))))
                           ^
In file included from
↪  ../../../gcc/gcc/config/riscv/riscv-selftests.cc:23:
In file included from ../../../gcc/gcc/system.h:227:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/map:2529:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/functional:526:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__functional/boyer_moore_searcher.h:27:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/vector:321:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_bool.h:20:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_integral.h:32:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/locale:202:
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:551:37: error:
↪  expected ';' at end of declaration list
    char_type toupper(char_type __c) const
                                     ^
```

Listing 52: Building Cross-Compiler results in Error, 1/12

```
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:557:48: error:
↪  too many arguments provided to function-like macro invocation
    const char_type* toupper(char_type* __low, const char_type*
    ↪  __high) const
                                                ^
../../../gcc/gcc/../include/safe-ctype.h:146:9: note: macro
↪  'toupper' defined here
#define toupper(c) do_not_use_toupper_with_safe_ctype
        ^
In file included from
↪  ../../../gcc/gcc/config/riscv/riscv-selftests.cc:23:
In file included from ../../../gcc/gcc/system.h:227:
In file included from
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/map:2529:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/functional:526:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__functional/boyer_moore_searcher.h:27:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/vector:321:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_bool.h:20:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_integral.h:32:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/locale:202:
```

Listing 53: Building Cross-Compiler results in Error, 2/12

```
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:569:48: error:
↪  too many arguments provided to function-like macro invocation
    const char_type* tolower(char_type* __low, const char_type*
    ↪  __high) const
                                                ^
../../../gcc/gcc/../include/safe-ctype.h:148:9: note: macro
↪  'tolower' defined here
#define tolower(c) do_not_use_tolower_with_safe_ctype
        ^
In file included from
↪  ../../../gcc/gcc/config/riscv/riscv-selftests.cc:23:
In file included from ../../../gcc/gcc/system.h:227:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/map:2529:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/functional:526:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__functional/boyer_moore_searcher.h:27:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/vector:321:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_bool.h:20:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_integral.h:32:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/locale:202:
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:660:5: error:
↪  '__abi_tag__' attribute only applies to structs, variables,
↪  functions, and namespaces
    _LIBCPP_INLINE_VISIBILITY
    ^
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__config:891:37: note:
↪  expanded from macro '_LIBCPP_INLINE_VISIBILITY'
#  define _LIBCPP_INLINE_VISIBILITY _LIBCPP_HIDE_FROM_ABI
                                     ^
```

Listing 54: Building Cross-Compiler results in Error, 3/12

```
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__config:870:26: note:
↪  expanded from macro '_LIBCPP_HIDE_FROM_ABI'

          ↪  __attribute__((__abi_tag__(_LIBCPP_TOSTRING(_LIBCPP_ODR_SIGNATURE))))
                        ^
In file included from
↪  ../../../gcc/gcc/config/riscv/riscv-selftests.cc:23:
In file included from ../../../gcc/gcc/system.h:227:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/map:2529:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/functional:526:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__functional/boyer_moore_searcher.h:27:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/vector:321:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_bool.h:20:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_integral.h:32:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/locale:202:
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:661:37: error:
↪  expected ';' at end of declaration list
    char_type toupper(char_type __c) const
                                    ^
```

Listing 55: Building Cross-Compiler results in Error, 4/12

```
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:667:48: error:
↪  too many arguments provided to function-like macro invocation
    const char_type* toupper(char_type* __low, const char_type*
    ↪  __high) const
                                                ^
../../../gcc/gcc/../include/safe-ctype.h:146:9: note: macro
↪  'toupper' defined here
#define toupper(c) do_not_use_toupper_with_safe_ctype
        ^
In file included from
↪  ../../../gcc/gcc/config/riscv/riscv-selftests.cc:23:
In file included from ../../../gcc/gcc/system.h:227:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/map:2529:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/functional:526:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__functional/boyer_moore_searcher.h:27:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/vector:321:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_bool.h:20:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_integral.h:32:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/locale:202:
```

Listing 56: Building Cross-Compiler results in Error, 5/12

```
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:679:48: error:
↪  too many arguments provided to function-like macro invocation
    const char_type* tolower(char_type* __low, const char_type*
    ↪  __high) const
                                                ^
../../../gcc/gcc/../include/safe-ctype.h:148:9: note: macro
↪  'tolower' defined here
#define tolower(c) do_not_use_tolower_with_safe_ctype
        ^
In file included from
↪  ../../../gcc/gcc/config/riscv/riscv-selftests.cc:23:
In file included from ../../../gcc/gcc/system.h:227:
In file included from
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/map:2529:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/functional:526:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__functional/boyer_moore_searcher.h:27:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/vector:321:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_bool.h:20:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_integral.h:32:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/locale:202:
```

Listing 57: Building Cross-Compiler results in Error, 6/12

```
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:795:21: error:
↪  too many arguments provided to function-like macro invocation
isspace(_CharT __c, const locale& __loc)
                    ^
../../../gcc/gcc/../include/safe-ctype.h:140:9: note: macro
↪  'isspace' defined here
#define isspace(c) do_not_use_isspace_with_safe_ctype
        ^
In file included from
↪  ../../../gcc/gcc/config/riscv/riscv-selftests.cc:23:
In file included from ../../../gcc/gcc/system.h:227:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/map:2529:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/functional:526:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__functional/boyer_moore_searcher.h:27:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/vector:321:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_bool.h:20:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_integral.h:32:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/locale:202:
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:795:1: error:
↪  declaration conflicts with target of using declaration
↪  already in scope
isspace(_CharT __c, const locale& __loc)
^
$MACOS_SDK/MacOSX.sdk/usr/include/_ctype.h:267:1: note: target of
↪  using declaration
isspace(int _c)
^
```

Listing 58: Building Cross-Compiler results in Error, 7/12

```
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/cctype:121:9: note:
↪  using declaration
using ::isspace _LIBCPP_USING_IF_EXISTS;
        ^
In file included from
↪  ../../../gcc/gcc/config/riscv/riscv-selftests.cc:23:
In file included from ../../../gcc/gcc/system.h:227:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/map:2529:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/functional:526:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__functional/boyer_moore_searcher.h:27:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/vector:321:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_bool.h:20:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_integral.h:32:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/locale:202:
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:797:5: error:
↪  expected expression
    return std::use_facet<ctype<_CharT>
    ↪  >(__loc).is(ctype_base::space, __c);
    ^
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:798:2: error:
↪  expected ';' at end of declaration
}
 ^
```

Listing 59: Building Cross-Compiler results in Error, 8/12

```
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:803:21: error:
↪  too many arguments provided to function-like macro invocation
isprint(_CharT __c, const locale& __loc)
                    ^
../../../gcc/gcc/../include/safe-ctype.h:136:9: note: macro
↪  'isprint' defined here
#define isprint(c) do_not_use_isprint_with_safe_ctype
        ^
In file included from
↪  ../../../gcc/gcc/config/riscv/riscv-selftests.cc:23:
In file included from ../../../gcc/gcc/system.h:227:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/map:2529:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/functional:526:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__functional/boyer_moore_searcher.h:27:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/vector:321:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_bool.h:20:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_integral.h:32:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/locale:202:
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:803:1: error:
↪  declaration conflicts with target of using declaration
↪  already in scope
isprint(_CharT __c, const locale& __loc)
^
$MACOS_SDK/MacOSX.sdk/usr/include/_ctype.h:255:1: note: target of
↪  using declaration
isprint(int _c)
^
```

Listing 60: Building Cross-Compiler results in Error, 9/12

```
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/cctype:119:9: note:
↪  using declaration
using ::isprint _LIBCPP_USING_IF_EXISTS;
        ^
In file included from
↪  ../../../gcc/gcc/config/riscv/riscv-selftests.cc:23:
In file included from ../../../gcc/gcc/system.h:227:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/map:2529:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/functional:526:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__functional/boyer_moore_searcher.h:27:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/vector:321:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_bool.h:20:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_integral.h:32:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/locale:202:
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:805:5: error:
↪  expected expression
    return std::use_facet<ctype<_CharT>
    ↪  >(__loc).is(ctype_base::print, __c);
    ^
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:806:2: error:
↪  expected ';' at end of declaration
}
 ^
```

Listing 61: Building Cross-Compiler results in Error, 10/12

```
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:811:21: error:
↪  too many arguments provided to function-like macro invocation
iscntrl(_CharT __c, const locale& __loc)
                    ^
../../../gcc/gcc/../include/safe-ctype.h:128:9: note: macro
↪  'iscntrl' defined here
#define iscntrl(c) do_not_use_iscntrl_with_safe_ctype
        ^
In file included from
↪  ../../../gcc/gcc/config/riscv/riscv-selftests.cc:23:
In file included from ../../../gcc/gcc/system.h:227:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/map:2529:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/functional:526:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__functional/boyer_moore_searcher.h:27:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/vector:321:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_bool.h:20:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_integral.h:32:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/locale:202:
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:811:1: error:
↪  declaration conflicts with target of using declaration
↪  already in scope
iscntrl(_CharT __c, const locale& __loc)
^
$MACOS_SDK/MacOSX.sdk/usr/include/_ctype.h:230:1: note: target of
↪  using declaration
iscntrl(int _c)
^
```

Listing 62: Building Cross-Compiler results in Error, 11/12

```
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/cctype:115:9: note:
↪  using declaration
using ::iscntrl _LIBCPP_USING_IF_EXISTS;
        ^
In file included from
↪  ../../../gcc/gcc/config/riscv/riscv-selftests.cc:23:
In file included from ../../../gcc/gcc/system.h:227:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/map:2529:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/functional:526:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__functional/boyer_moore_searcher.h:27:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/vector:321:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_bool.h:20:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__format/formatter_integral.h:32:
In file included from
↪  $MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/locale:202:
$MACOS_SDK/MacOSX.sdk/usr/include/c++/v1/__locale:813:5: error:
↪  expected expression
    return std::use_facet<ctype<_CharT>
    ↪  >(__loc).is(ctype_base::cntrl, __c);
    ^
fatal error: too many errors emitted, stopping now
↪  [-ferror-limit=]
1 warning and 20 errors generated.
make[2]: *** [riscv-selftests.o] Error 1
make[1]: *** [all-gcc] Error 2
make: *** [stamps/build-gcc-linux-stage1] Error 2
```

Listing 63: Building Cross-Compiler results in Error, 12/12

D.11 Failure during build of Linux image for QEMU

```
gmake ARCH=riscv CROSS_COMPILE=riscv64-unknown-elf- -j $(sysctl
↪  -n hw.physicalcpu)
  HOSTCC  scripts/sorttable
  HOSTCC  scripts/selinux/genheaders/genheaders
  HOSTCC  scripts/selinux/mdp/mdp
scripts/sorttable.c:27:10: fatal error: 'elf.h' file not found
#include <elf.h>
         ^~~~~~~
In file included from scripts/selinux/genheaders/genheaders.c:18:
In file included from ./security/selinux/include/classmap.h:2:
In file included from ./include/uapi/linux/capability.h:17:
./include/uapi/linux/types.h:5:10: fatal error: 'asm/types.h'
↪  file not found
#include <asm/types.h>
         ^~~~~~~~~~~~
```

Listing 64: Building Linux image for Qemu results in Error (1/2)

```
In file included from scripts/selinux/mdp/mdp.c:36:
In file included from ./security/selinux/include/classmap.h:2:
In file included from ./include/uapi/linux/capability.h:17:
./include/uapi/linux/types.h:5:10: fatal error: 'asm/types.h'
↪  file not found
#include <asm/types.h>
         ^~~~~~~~~~~~~
1 error generated.
1 error generated.
1 error generated.
gmake[4]: *** [scripts/Makefile.host:114:
↪  scripts/selinux/genheaders/genheaders] Error 1
gmake[4]: *** [scripts/Makefile.host:114:
↪  scripts/selinux/mdp/mdp] Error 1
gmake[3]: *** [scripts/Makefile.build:480: scripts/selinux/mdp]
↪  Error 2
gmake[3]: *** Waiting for unfinished jobs....
gmake[3]: *** [scripts/Makefile.build:480:
↪  scripts/selinux/genheaders] Error 2
gmake[2]: *** [scripts/Makefile.build:480: scripts/selinux] Error
↪  2
gmake[2]: *** Waiting for unfinished jobs....
gmake[2]: *** [scripts/Makefile.host:114: scripts/sorttable]
↪  Error 1
gmake[1]: *** [$REPO_ROOT/linux/linux/Makefile:1186: scripts]
↪  Error 2
gmake[1]: *** Waiting for unfinished jobs....
gmake: *** [Makefile:234: __sub-make] Error 2
```

Listing 65: Building Linux image for Qemu results in Error (2/2)

E Required code changes to enable RISC-V

E.1 Table of expected changes in the Kotlin Repository codebase

Table 7: List of expected changes based on ARM64 occurrences

File	Required change	Special hint
`kotlin-native/runtime/build.gradle.kts`	Add RISCV64 as Target architecture to determine the ELF size	
`kotlin-native/runtime/src/main/cpp/Runtime.cpp`	Add RISCV64 as c++ value, which is called by Platfom.kt	The returned Integer value is the ordinal of the enum entry of CPUArchitecture within Platform.kt. The entries for ordinals 5-7 are deprecated and therefore are not specified in the Runtime.cpp.
`kotlin-native/runtime/src/main/kotlin/Kotlin/Native/Platform.kt`	Add RISCV64 as CPUArchitecture enum entry	RISCV64 will have the ordinal 8, since it's the 8th enum. This is important for the return value of Konan_ Platform_ getCPUArchitecture() function of Runtime.cpp.
`libraries/tools/kotlin-gradle-plugin-api/src/common/kotlin/org/jetbrains/kotlin/gradle/plugin/KotlinHierarchyBuilder.kt`	Provide function to select linuxRiscV64 as Target in gradle	
`libraries/tools/kotlin-gradle-plugin/src/common/kotlin/org/jetbrains/kotlin/gradle/dsl/KotlinTargetContainerWithPresetFunctions.kt`	Provide functions to select linuxRiscV64 as Target in gradle	

Continued on next page

Table 7: (Continued) List of expected changes based on ARM64 occurrences

File	Required change	Special hint
`libraries/tools/kotlin-gradle-plugin/src/common/kotlin/org/jetbrains/kotlin/gradle/plugin/hierarchy/KotlinHierarchyBuilderImpl.kt`	Implements function to select linuxRiscV64 as Target in gradle	
`libraries/tools/kotlin-gradle-plugin/src/common/kotlin/org/jetbrains/kotlin/gradle/targets/native/tasks/FatFrameworkTask.kt`	Exclude RISCV64 as mapping option for apple architectures	
`libraries/tools/kotlin-gradle-plugin/src/common/kotlin/org/jetbrains/kotlin/gradle/targets/native/tasks/artifact/KotlinArtifactsExtensionImpl.kt`		According to a comment in the file, this change is generated and not needs to be done manually.
`native/commonizer/src/org/jetbrains/kotlin/commonizer/mergedtree/PlatformWidthIndex.kt`	Set the platform Integer size for RISCV64 to PlatformIntWidth.LONG	Naming of JetBrains here is questionable. In fact, it represents the Platform width for the common IR in Kotlin, which is used before compiling for the specific hardware target.
`native/utils/src/org/jetbrains/kotlin/konan/target/Architecture.kt`	Add RSICV64 as target Architecture enum entry	
`native/utils/src/org/jetbrains/kotlin/konan/target/HostManager.kt`	Add LINUX_RISCV64 to the list of commonTargets	The list of common targets is used indirectly by different gradle tasks to generate platform-specific binaries and libraries

Continued on next page

Table 7: (Continued) List of expected changes based on ARM64 occurrences

File	Required change	Special hint
`native/utils/src/org/jetbrains/kotlin/konan/target/KonanTarget.kt`	Define LINUX_RISCV64 as Kotlin/Native specific target which runs on LINUX on RISCV64 architecture and add it as predefined target	
`native/utils/src/org/jetbrains/kotlin/konan/target/KonanTargetExtenstions.kt`	Provide additional information for the targets, which are required in the build process	
`kotlin-native/build.gradle.kts`	Add LINUX_RISCV64 to the list of targets without Zlib	Zlib is not in the sysroot directory of the cross-compile-toolchain

E.2 Initialize RISC-V when LLVM gets initialized in CAPIExtensions.cpp

```
void LLVMKotlinInitializeTargets() {
#define INIT_LLVM_TARGET(TargetName) \
    LLVMInitialize##TargetName##TargetInfo();\
    LLVMInitialize##TargetName##Target();\
    LLVMInitialize##TargetName##TargetMC();
#if KONAN_MACOS
+    INIT_LLVM_TARGET(RISCV)
    INIT_LLVM_TARGET(AArch64)
    INIT_LLVM_TARGET(ARM)
    INIT_LLVM_TARGET(Mips)
    INIT_LLVM_TARGET(X86)
    INIT_LLVM_TARGET(WebAssembly)
#elif KONAN_LINUX
+    INIT_LLVM_TARGET(RISCV)
    INIT_LLVM_TARGET(AArch64)
    INIT_LLVM_TARGET(ARM)
    INIT_LLVM_TARGET(Mips)
    INIT_LLVM_TARGET(X86)
    INIT_LLVM_TARGET(WebAssembly)
#elif KONAN_WINDOWS
+    INIT_LLVM_TARGET(RISCV)
    INIT_LLVM_TARGET(AArch64)
    INIT_LLVM_TARGET(ARM)
    INIT_LLVM_TARGET(X86)
    INIT_LLVM_TARGET(WebAssembly)
#endif
```

Listing 66: Add RISCV as target when LLVM gets initialized in CAPIExtensions.cpp

E.3 Add RISC-V to LLVM C-Interop configuration

```
linkerOpts.osx = \
    -Wl,-search_paths_first -Wl,-headerpad_max_install_names \
    -lpthread -lz -lm  -lcurses -Wl,-U,_futimens
    ↪  -Wl,-U,_LLVMDumpType \
    -Wl,-exported_symbols_list,llvm.list \
    -lLLVMMipsDisassembler -lLLVMMipsCodeGen -lLLVMMipsAsmParser
    ↪  -lLLVMMipsDesc -lLLVMMipsInfo -lLLVMX86Disassembler
    ↪  -lLLVMX86AsmParser \
    -lLLVMX86CodeGen -lLLVMX86Desc -lLLVMX86Info
    ↪  -lLLVMWebAssemblyDisassembler -lLLVMWebAssemblyCodeGen
    ↪  -lLLVMWebAssemblyDesc \
    -lLLVMWebAssemblyAsmParser -lLLVMWebAssemblyInfo
    ↪  -lLLVMAArch64Disassembler -lLLVMAArch64CodeGen
    ↪  -lLLVMAArch64AsmParser -lLLVMAArch64Desc \
    -lLLVMAArch64Utils -lLLVMAArch64Info -lLLVMARMDisassembler
    ↪  -lLLVMARMCodeGen -lLLVMCFGuard -lLLVMGlobalISel
    ↪  -lLLVMSelectionDAG \
    -lLLVMAsmPrinter -lLLVMDebugInfoDWARF -lLLVMARMAsmParser
    ↪  -lLLVMARMDesc -lLLVMMCDisassembler -lLLVMARMUtils
    ↪  -lLLVMARMInfo -lLLVMLTO \
    -lLLVMPasses -lLLVMCoroutines -lLLVMObjCARCOpts
    ↪  -lLLVMExtensions -lLLVMCodeGen -lLLVMipo
    ↪  -lLLVMInstrumentation -lLLVMVectorize \
    -lLLVMScalarOpts -lLLVMIRReader -lLLVMAsmParser
    ↪  -lLLVMInstCombine -lLLVMFrontendOpenMP
    ↪  -lLLVMAggressiveInstCombine \
    -lLLVMTarget -lLLVMLinker -lLLVMTransformUtils
    ↪  -lLLVMBitWriter -lLLVMAnalysis -lLLVMProfileData
    ↪  -lLLVMObject -lLLVMTextAPI -lLLVMMCParser \
    -lLLVMMC -lLLVMDebugInfoCodeView -lLLVMDebugInfoMSF
    ↪  -lLLVMBitReader -lLLVMCore -lLLVMRemarks
    ↪  -lLLVMBitstreamReader -lLLVMBinaryFormat \
-    -lLLVMSupport -lLLVMDemangle
+    -lLLVMSupport -lLLVMDemangle \
+    -lLLVMRISCVCodeGen -lLLVMRISCVAsmParser -lLLVMRISCVDesc
↪  -lLLVMRISCVUtils -lLLVMRISCVInfo -lLLVMRISCVDisassembler
```

Listing 67: Add LLVM RISC-V libraries to llvm.def

E.4 New entries in konan.properties

```
# Linux RiscV64
toolchainDependency.linux_riscv64 =
↪  riscv64-glibc-ubuntu-20.04-gcc-nightly-2023.12.14-nightly
gccToolchain.linux_riscv64 = $toolchainDependency.linux_riscv64

targetToolchain.linux_x64-linux_riscv64 =
↪  $gccToolchain.linux_riscv64/riscv64-unknown-linux-gnu/
targetToolchain.mingw_x64-linux_riscv64 = $llvmHome.mingw_x64
targetToolchain.macos_x64-linux_riscv64 = $llvmHome.macos_x64
targetToolchain.macos_arm64-linux_riscv64 = $llvmHome.macos_arm64

emulatorDependency.linux_x64-linux_riscv64 = qemu-riscv64
emulatorExecutable.linux_x64-linux_riscv64 = qemu-riscv64

dependencies.linux_x64-linux_riscv64 = \
    $toolchainDependency.linux_riscv64
dependencies.mingw_x64-linux_riscv64 = \
    $toolchainDependency.linux_riscv64 \
    $toolchainDependency.mingw_x64
dependencies.macos_x64-linux_riscv64 = \
    $toolchainDependency.linux_riscv64
dependencies.macos_arm64-linux_riscv64 = \
    $toolchainDependency.linux_riscv64

targetTriple.linux_riscv64 = riscv64-unknown-linux-gnu
linkerNoDebugFlags.linux_riscv64 = -S
linkerDynamicFlags.linux_riscv64 = -shared
linkerOptimizationFlags.linux_riscv64 = --gc-sections
targetSysRoot.linux_riscv64 = $gccToolchain.linux_riscv64/sysroot
# We could reuse host toolchain here.

linkerKonanFlags.linux_riscv64 = -lstdc++ -Bdynamic -ldl -lm
↪  -lpthread \
  --defsym __cxa_demangle=Konan_cxa_demangle

# todo: would be great if this would work as well:
#linkerKonanFlags.linux_riscv64 = -Bstatic -lstdc++ -Bdynamic
↪  -ldl -lm -lpthread \
#  --defsym __cxa_demangle=Konan_cxa_demangle
```

Listing 68: New entries in konan.properties file, 1/3

```
# targetSysroot-relative.
libGcc.linux_riscv64 =
↪  ../lib/gcc/riscv64-unknown-linux-gnu/13.2.0
# to show cpuFeatures use:
↪  ~/.konan/dependencies/apple-llvm-20200714-macos-x64-1/bin/llc
↪  -march=riscv64 -mattr=help
# sifive works
# targetCpu.linux_riscv64=sifive-u54
# generic-rv64 not yet
targetCpu.linux_riscv64=generic-rv64
targetCpuFeatures.linux_riscv64=+m,+a,+relax,+d,+f,+64bit
# Super important note: -cc1 has to be the very first flag!
↪  otherwise '-triple' in the computed clangflags wont work!
clangFlags.linux_riscv64 = -cc1 -v -emit-obj -disable-llvm-optzns
↪  -target-abi lp64d -x ir
# TODO: Double check clangNooptFlags
clangNooptFlags.linux_riscv64 = -O1
# TODO: Double check clangOptFlags
clangOptFlags.linux_riscv64 = -O3 -ffunction-sections
# TODO: Double check clangDebugFlags
clangDebugFlags.linux_riscv64 = -O0
# TODO: Double check dynamicLibraryRelocationMode
dynamicLibraryRelocationMode.linux_riscv64 =
# TODO: Double check staticLibraryRelocationMode
staticLibraryRelocationMode.linux_riscv64 =
linker.linux_x64-linux_riscv64 =
↪  $targetToolchain.linux_x64-linux_riscv64/bin/ld.gold
# Todo: Should linkerHostSpecificFlags be empty ?
linkerHostSpecificFlags.linux_x64-linux_riscv64 =
# TODO: Check ld.lld from LLVM.
linker.mingw_x64-linux_riscv64 =
↪  $targetToolchain.mingw_x64/bin/ld.gold
linkerHostSpecificFlags.mingw_x64-linux_riscv64 =
```

Listing 69: New entries in konan.properties file, 2/3

```
# todo: llvm ld.lld < version 15 does not support relaxation.
↪  cross-compile-toolchain is build with relaxation. The
↪  following would work, but i think its not recommended to use
↪  different llvm builds.
# todo: it might make sense to rebuild the kotlin compiler with
↪  newer llvm build.
linker.macos_x64-linux_riscv64 =
↪  $targetToolchain.macos_x64-linux_riscv64/../
↪  apple-llvm-20230725-macos-x64-1/bin/ld.lld
linkerHostSpecificFlags.macos_x64-linux_riscv64 =
linker.macos_arm64-linux_riscv64 =
↪  $targetToolchain.macos_arm64-linux_riscv64/bin/ld.lld
#  Todo: Should linkerHostSpecificFlags be empty ?
linkerHostSpecificFlags.macos_arm64-linux_riscv64 =
dynamicLinker.linux_riscv64 = /lib/ld-linux-riscv64-lp64d.so.1
# targetSysRoot relative
abiSpecificLibraries.linux_riscv64 = lib usr/lib
# targetSysRoot relative
crtFilesLocation.linux_riscv64 = usr/lib
```

Listing 70: New entries in konan.properties file, 3/3

E.5 Avoid crash on unknown dependencies

```
        /** Performs an attempt to download a specified file into
        ↪  the specified location */
        private fun tryDownload(url: URL, tmpFile: File) {
+        if (url.file.contains("riscv64")) {
+            println("tryDownload(): ignore $url to $tmpFile")
+            return
+        }

        val connection = url.openConnection()

        (connection as?
        ↪  HttpURLConnection)?.checkHTTPResponse(HttpURLConnection.HTTP_OK,
        ↪  url)

        if (connection is HttpURLConnection && tmpFile.exists())
        ↪  {
            resumeDownload(url, connection, tmpFile)
        } else {
            connection.connect()
            val totalBytes = connection.contentLengthLong
            doDownload(url, connection, tmpFile, 0, totalBytes,
            ↪  false)
        }
    }
```

Listing 71: Changes in DependencyDownloader.kt to avoid crash druing downloading dependencies

E.6 Remove deprecated std::iterator usage

```
    // The iterator class
    template <typename octet_iterator>
-    class iterator : public std::iterator
↪   <std::bidirectional_iterator_tag, uint32_t> {
+    class iterator{
      octet_iterator it;
      octet_iterator range_start;
      octet_iterator range_end;
+
      public:
+      using difference_type = uint32_t;
+      using iterator_category = std::bidirectional_iterator_tag;
+      using value_type = uint32_t;
+      using pointer = uint32_t*;
+      using reference = uint32_t&;
+
      iterator () {}
```

Listing 72: Code change to resolve error of 41 in checked.h

E.7 Add new platformlibs directory for Linux RISC-V target

```
// region: Util functions.
+ fun platformDirectory(family : Family, architecture:
↪  org.jetbrains.kotlin.konan.target.Architecture) =
↪  "src/platform/${family.visibleName}" +
↪  if(architecture.toString().contains("RISCV")) "_for_riscv"
↪  else ""

fun KonanTarget.defFiles() =
-     project.fileTree("src/platform/${family.visibleName}")
+ project.fileTree(platformDirectory(family, architecture))
    .filter { it.name.endsWith(".def") }
    // The libz.a/libz.so and zlib.h are missing in MIPS
    ↪  sysroots.
    // Just workaround it until we have sysroots corrected.
    .filterNot { (this in targetsWithoutZlib) && it.name ==
    ↪  "zlib.def" }
    .map { DefFile(it, this) }
```

Listing 73: Code change to use custom platformlib configuration in /kotlin-native/platformLibs/build.gradle.kts

F Required code changes to enable LLVM 17

F.1 Update references in llvm.def

```
headers = llvm-c/Core.h llvm-c/Target.h llvm-c/Analysis.h
↪  llvm-c/BitWriter.h \
-    llvm-c/BitReader.h llvm-c/Transforms/PassManagerBuilder.h
↪  llvm-c/Transforms/IPO.h \
-    llvm-c/TargetMachine.h llvm-c/Target.h llvm-c/Linker.h
↪  llvm-c/Initialization.h \
+    llvm-c/BitReader.h llvm-c/Transforms/PassBuilder.h
↪  llvm-c/Types.h \
+    llvm-c/TargetMachine.h llvm-c/Target.h llvm-c/Linker.h \
    llvm-c/DebugInfo.h DebugInfoC.h CAPIExtensions.h
    ↪  RemoveRedundantSafepoints.h OpaquePointerAPI.h

headerFilter = llvm-c/* llvm-c/**/* DebugInfoC.h CAPIExtensions.h
↪  RemoveRedundantSafepoints.h OpaquePointerAPI.h

compilerOpts = -std=c99 \
    -Wall -W -Wno-unused-parameter -Wwrite-strings
    ↪  -Wmissing-field-initializers \
    -pedantic -Wno-long-long -Wcovered-switch-default
    ↪  -Wdelete-non-virtual-dtor \
    -DNDEBUG -D__STDC_CONSTANT_MACROS -D__STDC_FORMAT_MACROS
    ↪  -D__STDC_LIMIT_MACROS


linker = clang++

linkerOpts = -fvisibility-inlines-hidden \
    -Wall -W -Wno-unused-parameter -Wwrite-strings -Wcast-qual
    ↪  -Wmissing-field-initializers \
    -pedantic -Wno-long-long -Wcovered-switch-default
    ↪  -Wnon-virtual-dtor -Wdelete-non-virtual-dtor \
    -std=c++17 \
    -DNDEBUG -D__STDC_CONSTANT_MACROS -D__STDC_FORMAT_MACROS
    ↪  -D__STDC_LIMIT_MACROS \
-    -ldebugInfo -lllvmext
+    -ldebugInfo -lllvmext \
+    -L/usr/local/opt/zstd/lib/ -lzstd
```

Listing 74: Update llvm.def file to enable functions of LLVM 17, 1/5

```
# ./llvm-config --libs analysis bitreader bitwriter core linker
↪  target analysis ipo instrumentation lto objcarcopts arm
↪  aarch64 webassembly x86 mips
linkerOpts.osx = \
    -Wl,-search_paths_first -Wl,-headerpad_max_install_names \
    -lpthread -lz -lm  -lcurses -Wl,-U,_futimens
    ↪  -Wl,-U,_LLVMDumpType \
    -Wl,-exported_symbols_list,llvm.list \
+    -lLLVMMCCAS -lLLVMIRPrinter -lLLVMTargetParser
↪  -lLLVMCodeGenTypes -lLLVMCAS -lLLVMCASUtil \
    -lLLVMMipsDisassembler -lLLVMMipsCodeGen -lLLVMMipsAsmParser
    ↪  -lLLVMMipsDesc -lLLVMMipsInfo -lLLVMX86Disassembler
    ↪  -lLLVMX86AsmParser \
-    -lLLVMX86CodeGen -lLLVMX86Desc -lLLVMX86Info
↪  -lLLVMWebAssemblyDisassembler -lLLVMWebAssemblyCodeGen
↪  -lLLVMWebAssemblyDesc \
+    -lLLVMX86CodeGen -lLLVMX86Desc -lLLVMX86Info
↪  -lLLVMWebAssemblyUtils -lLLVMWebAssemblyDisassembler
↪  -lLLVMWebAssemblyCodeGen -lLLVMWebAssemblyDesc \
    -lLLVMWebAssemblyAsmParser -lLLVMWebAssemblyInfo
    ↪  -lLLVMAArch64Disassembler -lLLVMAArch64CodeGen
    ↪  -lLLVMAArch64AsmParser -lLLVMAArch64Desc \
    -lLLVMAArch64Utils -lLLVMAArch64Info -lLLVMARMDisassembler
    ↪  -lLLVMARMCodeGen -lLLVMCFGuard -lLLVMGlobalISel
    ↪  -lLLVMSelectionDAG \
    -lLLVMAsmPrinter -lLLVMDebugInfoDWARF -lLLVMARMAsmParser
    ↪  -lLLVMARMDesc -lLLVMMCDisassembler -lLLVMARMUtils
    ↪  -lLLVMARMInfo -lLLVMLTO \
    -lLLVMPasses -lLLVMCoroutines -lLLVMObjCARCOpts
    ↪  -lLLVMExtensions -lLLVMCodeGen -lLLVMipo
    ↪  -lLLVMInstrumentation -lLLVMVectorize \
    -lLLVMScalarOpts -lLLVMIRReader -lLLVMAsmParser
    ↪  -lLLVMInstCombine -lLLVMFrontendOpenMP
    ↪  -lLLVMAggressiveInstCombine \
    -lLLVMTarget -lLLVMLinker -lLLVMTransformUtils
    ↪  -lLLVMBitWriter -lLLVMAnalysis -lLLVMProfileData
    ↪  -lLLVMObject -lLLVMTextAPI -lLLVMMCParser \
    -lLLVMMC -lLLVMDebugInfoCodeView -lLLVMDebugInfoMSF
    ↪  -lLLVMBitReader -lLLVMCore -lLLVMRemarks
    ↪  -lLLVMBitstreamReader -lLLVMBinaryFormat \
    -lLLVMSupport -lLLVMDemangle \
-    -lLLVMRISCVCodeGen -lLLVMRISCVAsmParser -lLLVMRISCVDesc
↪  -lLLVMRISCVUtils -lLLVMRISCVInfo -lLLVMRISCVDisassembler
+    -lLLVMRISCVCodeGen -lLLVMRISCVAsmParser -lLLVMRISCVDesc
↪  -lLLVMRISCVInfo -lLLVMRISCVDisassembler
```

Listing 75: Update llvm.def file to enable functions of LLVM 17, 2/5

```
# ./llvm-config --libs analysis bitreader bitwriter core linker
 ↪  target analysis ipo instrumentation lto arm aarch64
 ↪  webassembly x86 mips
linkerOpts.linux = \
    -Wl,-z,noexecstack \
    -lrt -ldl -lpthread -lz -lm \
    -lLLVMMipsDisassembler -lLLVMMipsCodeGen -lLLVMMipsAsmParser
     ↪  -lLLVMMipsDesc -lLLVMMipsInfo -lLLVMX86Disassembler
     ↪  -lLLVMX86AsmParser \
    -lLLVMX86CodeGen -lLLVMX86Desc -lLLVMX86Info
     ↪  -lLLVMWebAssemblyDisassembler -lLLVMWebAssemblyCodeGen
     ↪  -lLLVMWebAssemblyDesc \
    -lLLVMWebAssemblyAsmParser -lLLVMWebAssemblyInfo
     ↪  -lLLVMAArch64Disassembler -lLLVMAArch64CodeGen
     ↪  -lLLVMAArch64AsmParser -lLLVMAArch64Desc \
    -lLLVMAArch64Utils -lLLVMAArch64Info -lLLVMARMDisassembler
     ↪  -lLLVMARMCodeGen -lLLVMCFGuard -lLLVMGlobalISel
     ↪  -lLLVMSelectionDAG \
    -lLLVMAsmPrinter -lLLVMDebugInfoDWARF -lLLVMARMAsmParser
     ↪  -lLLVMARMDesc -lLLVMMCDisassembler -lLLVMARMUtils
     ↪  -lLLVMARMInfo -lLLVMLTO \
    -lLLVMPasses -lLLVMCoroutines -lLLVMObjCARCOpts
     ↪  -lLLVMExtensions -lLLVMCodeGen -lLLVMipo
     ↪  -lLLVMInstrumentation -lLLVMVectorize \
    -lLLVMScalarOpts -lLLVMIRReader -lLLVMAsmParser
     ↪  -lLLVMInstCombine -lLLVMFrontendOpenMP
     ↪  -lLLVMAggressiveInstCombine \
    -lLLVMTarget -lLLVMLinker -lLLVMTransformUtils
     ↪  -lLLVMBitWriter -lLLVMAnalysis -lLLVMProfileData
     ↪  -lLLVMObject -lLLVMTextAPI -lLLVMMCParser \
    -lLLVMMC -lLLVMDebugInfoCodeView -lLLVMDebugInfoMSF
     ↪  -lLLVMBitReader -lLLVMCore -lLLVMRemarks
     ↪  -lLLVMBitstreamReader -lLLVMBinaryFormat \
    -lLLVMSupport -lLLVMDemangle \
-    -lLLVMRISCVCodeGen -lLLVMRISCVAsmParser -lLLVMRISCVDesc
 ↪  -lLLVMRISCVUtils -lLLVMRISCVInfo -lLLVMRISCVDisassembler
+    -lLLVMRISCVCodeGen -lLLVMRISCVAsmParser -lLLVMRISCVDesc
 ↪  -lLLVMRISCVInfo -lLLVMRISCVDisassembler
```

Listing 76: Update llvm.def file to enable functions of LLVM 17, 3/5

```
# ./llvm-config --libs analysis bitreader bitwriter core linker
 ↪  target analysis ipo instrumentation lto arm aarch64
 ↪  webassembly x86
linkerOpts.mingw = \
    -lLLVMX86Disassembler -lLLVMX86AsmParser -lLLVMX86CodeGen
     ↪  -lLLVMX86Desc -lLLVMX86Info -lLLVMWebAssemblyDisassembler
     ↪  \
    -lLLVMWebAssemblyCodeGen -lLLVMWebAssemblyDesc
     ↪  -lLLVMWebAssemblyAsmParser -lLLVMWebAssemblyInfo
     ↪  -lLLVMAArch64Disassembler \
    -lLLVMAArch64CodeGen -lLLVMAArch64AsmParser -lLLVMAArch64Desc
     ↪  -lLLVMAArch64Utils -lLLVMAArch64Info
     ↪  -lLLVMARMDisassembler \
    -lLLVMARMCodeGen -lLLVMCFGuard -lLLVMGlobalISel
     ↪  -lLLVMSelectionDAG -lLLVMAsmPrinter -lLLVMDebugInfoDWARF
     ↪  -lLLVMARMAsmParser -lLLVMARMDesc \
    -lLLVMMCDisassembler -lLLVMARMUtils -lLLVMARMInfo -lLLVMLTO
     ↪  -lLLVMPasses -lLLVMCoroutines -lLLVMObjCARCOpts
     ↪  -lLLVMExtensions -lLLVMCodeGen \
    -lLLVMipo -lLLVMInstrumentation -lLLVMVectorize
     ↪  -lLLVMScalarOpts -lLLVMIRReader -lLLVMAsmParser
     ↪  -lLLVMInstCombine -lLLVMFrontendOpenMP \
    -lLLVMAggressiveInstCombine -lLLVMTarget -lLLVMLinker
     ↪  -lLLVMTransformUtils -lLLVMBitWriter -lLLVMAnalysis
     ↪  -lLLVMProfileData \
    -lLLVMObject -lLLVMTextAPI -lLLVMMCParser -lLLVMMC
     ↪  -lLLVMDebugInfoCodeView -lLLVMDebugInfoMSF
     ↪  -lLLVMBitReader -lLLVMCore -lLLVMRemarks \
    -lLLVMBitstreamReader -lLLVMBinaryFormat -lLLVMSupport
     ↪  -lLLVMDemangle \
-    -lLLVMRISCVCodeGen -lLLVMRISCVAsmParser -lLLVMRISCVDesc
 ↪  -lLLVMRISCVUtils -lLLVMRISCVInfo -lLLVMRISCVDisassembler \
+    -lLLVMRISCVCodeGen -lLLVMRISCVAsmParser -lLLVMRISCVDesc
 ↪  -lLLVMRISCVInfo -lLLVMRISCVDisassembler \
    -lpsapi -lshell32 -lole32 -luuid -ladvapi32
```

Listing 77: Update llvm.def file to enable functions of LLVM 17, 4/5

```
# It looks like mingw port compiled without LLVM_ENABLE_DUMP
#Note: ld on mingw process -Wl,-U,_LLVMDumpType use different
↪  from other platform
# way, using this option cause linkage error:
# ld: -r and -shared may not be used together
# TODO: Is ^^^ still relative? Especially since we use native
↪  Windows LLVM.
excludedFunctions.mingw = LLVMDumpType

# Functions from LLVMIntPtrType to LLVMModuleCreateWithName are
↪  excluded because they work with the GlobalContext.
# This might not be safe if the compiler is called from a daemon
↪  process.
#
# Also exclude the functions that rely on typed pointers as we
↪  get rid of them from the code generator.
+ # See OpaquePointer usage:
+ #
↪  https://releases.llvm.org/17.0.1/docs/OpaquePointers.html#frontends
excludedFunctions = LLVMInitializeAllAsmParsers
↪  LLVMInitializeAllAsmPrinters LLVMInitializeAllDisassemblers \
    LLVMInitializeAllTargetInfos LLVMInitializeAllTargetMCs
    ↪  LLVMInitializeAllTargets LLVMInitializeNativeTarget \
    LLVMInitializeNativeAsmParser LLVMInitializeNativeAsmPrinter
    ↪  LLVMInitializeNativeDisassembler \
    LLVMIntPtrType LLVMIntPtrTypeForAS LLVMGetMDKindID
    ↪  LLVMInt1Type LLVMInt8Type \
    LLVMInt16Type LLVMInt32Type LLVMInt64Type LLVMInt128Type
    ↪  LLVMIntType LLVMHalfType LLVMFloatType LLVMDoubleType \
    LLVMX86FP80Type LLVMFP128Type LLVMPPCFP128Type LLVMX86MMXType
    ↪  LLVMStructType LLVMVoidType LLVMLabelType \
    LLVMMDString LLVMMDNode LLVMConstString LLVMConstStruct
    ↪  LLVMAppendBasicBlock LLVMInsertBasicBlock
    ↪  LLVMCreateBuilder \
    LLVMParseBitcode LLVMParseBitcode2 LLVMGetBitcodeModule
    ↪  LLVMGetBitcodeModule2 LLVMGetGlobalContext
    ↪  LLVMModuleCreateWithName \
    LLVMBuildLoad LLVMBuildGEP LLVMBuildStructGEP LLVMConstGEP
    ↪  LLVMConstInBoundsGEP LLVMAddAlias LLVMBuildInvoke
    ↪  LLVMBuildCall \
-    LLVMGetElementType
+    LLVMGetElementType LLVMBuildInBoundsGEP LLVMBuildPtrDiff

strictEnums = LLVMIntPredicate LLVMOpcode LLVMDLLStorageClass
↪  LLVMCallConv LLVMThreadLocalMode LLVMAtomicOrdering
```

Listing 78: Update llvm.def file to enable functions of LLVM 17, 5/5

F.2 Update references in clang.def

```
-headers = clang-c/Index.h clang-c/ext.h
+headers = clang-c/Platform.h clang-c/CXSourceLocation.h
↪  clang-c/Index.h  clang-c/CXFile.h clang-c/CXDiagnostic.h
↪  clang-c/ext.h

headerFilter = clang-c/**

compiler = clang

-compilerOpts = -std=c99
+compilerOpts = -std=c99 -ferror-limit=0

linkerOpts.linux = -Wl,-z,noexecstack

linker = clang++

strictEnums = CXErrorCode CXCursorKind CXTypeKind
↪  CXDiagnosticSeverity CXLoadDiag_Error CXSaveError \
   CXTUResourceUsageKind CXLinkageKind CXVisibilityKind
    ↪  CXLanguageKind CXCallingConv CXChildVisitResult \
-    CXTokenKind CXEvalResultKind CXVisitorResult CXResult
↪  CXIdxEntityKind
+    CXTokenKind CXEvalResultKind CXVisitorResult CXResult
↪  CXIdxEntityKind CXType CXChoice CXSourceLocation

# These functions are not available in 11.1.0 upstream.
-excludedFunctions = clang_Cursor_getVarDeclInitializer
↪  clang_Cursor_hasVarDeclGlobalStorage
↪  clang_Cursor_hasVarDeclExternalStorage
+# excludedFunctions = clang_Cursor_getVarDeclInitializer
↪  clang_Cursor_hasVarDeclGlobalStorage
↪  clang_Cursor_hasVarDeclExternalStorage
```

Listing 79: Changes of clang.def required by the increase to LLVM 17

F.3 Update MappingBridgeGeneratorImpl.kt

```
when (unwrappedReturnType) {
                is VoidType -> {
                    out(nativeResult + ";")
                    ""
                }
                is RecordType -> {
                    val kniStructResult = "kniStructResult"

                    out("${unwrappedReturnType.decl.spelling}
                    ↪  $kniStructResult = $nativeResult;")
-                    out("memcpy(${bridgeNativeValues.last()},
↪  &$kniStructResult, sizeof($kniStructResult));")
+                    out("memcpy((void
↪  *)${bridgeNativeValues.last()}, &$kniStructResult,
↪  sizeof($kniStructResult));")
                    ""
                }
                else -> {
                    nativeResult
                }
            }
        }
```

Listing 80: Cast pointer to void * for memcpy invocation

F.4 Add ZSTD library to input path

```
    ldflags.addAll(llvmLibs)
    ldflags.addAll(listOf("-lpthread", "-lz", "-lm", "-lcurses"))
+
↪  ldflags.addAll(listOf("-L/usr/local/opt/zstd/lib/","-lzstd"))
```

Listing 81: Link zstd library for clangStub generation

G File analysis

G.1 LLVM IR comparison

```
; ModuleID = 'hello-world.c'
source_filename = "hello-world.c"
target datalayout =
↪  "e-m:o-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx14.0.0"
; Function Attrs: noinline nounwind optnone ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 5, i32* %1, align 4
  store i32 42, i32* %2, align 4
  %4 = load i32, i32* %1, align 4
  %5 = load i32, i32* %2, align 4
  %6 = sub nsw i32 %4, %5
  store i32 %6, i32* %3, align 4
  ret i32 0
}
attributes #0 = { noinline nounwind optnone ssp uwtable
↪  "correctly-rounded-divide-sqrt-fp-math"="false"
↪  "disable-tail-calls"="false" "frame-pointer"="all"
↪  "less-precise-fpmad"="false" "min-legal-vector-width"="0"
↪  "no-infs-fp-math"="false" "no-jump-tables"="false"
↪  "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
↪  "no-trapping-math"="true" "stack-protector-buffer-size"="8"
↪  "target-cpu"="penryn"
↪  "target-features"="+cx16,+cx8,+fxsr,+mmx,+sahf,+sse,+sse2,
↪  +sse3,+sse4.1,+ssse3,+x87" "unsafe-fp-math"="false"
↪  "use-soft-float"="false" }
!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}
!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"clang version 11.1.0
↪  (https://github.com/apple/llvm-project
↪  9205ffc7869a87cf3906b80dbd45b969c5794ef7)"}
```

Listing 82: LLVM 11.1.0 IR for simple C-program

```
; ModuleID = 'hello-world.c'
source_filename = "hello-world.c"
target datalayout =
↪  "e-m:o-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx14.0.0"

; Function Attrs: noinline nounwind optnone ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 5, ptr %1, align 4
  store i32 42, ptr %2, align 4
  %4 = load i32, ptr %1, align 4
  %5 = load i32, ptr %2, align 4
  %6 = sub nsw i32 %4, %5
  store i32 %6, ptr %3, align 4
  ret i32 0
}

attributes #0 = { noinline nounwind optnone ssp uwtable
↪  "frame-pointer"="all" "min-legal-vector-width"="0"
↪  "no-trapping-math"="true" "stack-protector-buffer-size"="8"
↪  "target-cpu"="penryn"
↪  "target-features"="+cmov,+cx16,+cx8,+fxsr,+mmx,+sahf,+sse,
↪  +sse2,+sse3,+sse4.1,+ssse3,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}
!2 = !{i32 7, !"uwtable", i32 2}
!3 = !{i32 7, !"frame-pointer", i32 2}
!4 = !{!"clang version 17.0.5 (https://github.com/apple/
↪  llvm-project 3d6e0f96eb2caa264b6de28533c62c423aa85f22)"}
```

Listing 83: LLVM 17.0.5 IR for simple C-program

G.2 ELF analysis of crt1.o

```
readelf -dr \
$KONAN_HOME/dependencies/
↪  riscv64-glibc-ubuntu-20.04-gcc-nightly-2023.12.14-nightly/
↪  sysroot/usr/lib/crt1.o

There is no dynamic section in this file.

Relocation section '.rela.text' at offset 0x4e0 contains 11
↪  entries:
  Offset          Info           Type           Sym. Value
   ↪  Sym. Name + Addend
000000000000  00000000002b R_RISCV_ALIGN                        2
000000000002  000f00000013 R_RISCV_CALL_PLT  000000000000002c
↪  load_gp + 0
000000000002  000000000033 R_RISCV_RELAX                        0
00000000000c  001800000017 R_RISCV_PCREL_HI2 0000000000000000
↪  main + 0
00000000000c  000000000033 R_RISCV_RELAX                        0
000000000010  001000000018 R_RISCV_PCREL_LO1 000000000000000c .L0
↪  + 0
000000000010  000000000033 R_RISCV_RELAX                        0
000000000022  001b00000013 R_RISCV_CALL_PLT  0000000000000000
↪  __libc_start_main + 0
000000000022  000000000033 R_RISCV_RELAX                        0
00000000002c  001600000017 R_RISCV_PCREL_HI2 0000000000000000
↪  __global_pointer$ + 0
000000000030  001100000018 R_RISCV_PCREL_LO1 000000000000002c .L0
↪  + 0
```

Listing 84: ELF representation of crt1.o, 1/2

```
Relocation section '.rela.eh_frame' at offset 0x5e8 contains 3
↪  entries:
  Offset          Info           Type           Sym. Value
   ↪  Sym. Name + Addend
00000000001c  001200000039 R_RISCV_32_PCREL  0000000000000002 .L0
↪  + 0
000000000020  001300000023 R_RISCV_ADD32     000000000000002c .L0
↪  + 0
000000000020  001200000027 R_RISCV_SUB32     0000000000000002 .L0
↪  + 0

Relocation section '.rela.preinit_array' at offset 0x630 contains
↪  1 entry:
  Offset          Info           Type           Sym. Value
   ↪  Sym. Name + Addend
000000000000  000f00000002 R_RISCV_64        000000000000002c
↪  load_gp + 0
```

Listing 85: ELF representation of crt1.o, 2/2

References

[1] 2022. URL: https://www.jetbrains.com/lp/devecosystem-2022/kotlin/ (visited on 01/02/2024).

[2] Nov. 2023. URL: https://kotlinlang.org/docs/multiplatform-dsl-reference.html (visited on 01/02/2024).

[3] 2023. URL: https://kotlinlang.org/docs/native-target-support.html (visited on 01/02/2024).

[4] URL: https://kotlinlang.org/docs/compiler-reference.html (visited on 01/15/2024).

[5] Enfang Cui, Tianzheng Li, and Qian Wei. "RISC-V Instruction Set Architecture Extensions: A Survey". In: *IEEE Access* 11 (2023), pp. 24696–24711. DOI: 10.1109/ACCESS.2023.3246491.

[6] URL: https://go.dev/doc/go1.14 (visited on 01/02/2024).

[7] URL: https://go.dev/doc/devel/release (visited on 01/02/2024).

[8] URL: https://wiki.riscv.org/display/HOME/Language+Runtimes (visited on 01/02/2024).

[9] URL: https://releases.llvm.org/9.0.0/docs/ReleaseNotes.html (visited on 01/02/2024).

[10] Campbell Jones. URL: https://youtrack.jetbrains.com/issue/KT-43854/Kotlin-Native-Linux-RISC-V-targets (visited on 09/25/2023).

[11] URL: https://github.com/JetBrains/kotlin/blob/master/license/LICENSE.txt (visited on 01/15/2024).

[12] JetBrains s.r.o. *kotlin-native/README*. Version v2.0.0-Beta2. Jan. 26, 2024. URL: https://github.com/JetBrains/kotlin/blob/v2.0.0-Beta2/kotlin-native/README.md.

[13] LLVM Project. URL: https://llvm.org/docs/LangRef.html (visited on 03/19/2024).

[14] Amy Brown and Greg Wilson. *The architecture of Open source applications*. Creative Commons, 2011. ISBN: 9781257638017.

[15] LLVM Project. URL: https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html (visited on 03/19/2024).

[16] LLVM Project. URL: https://llvm.org/doxygen/group__LLVMC.html (visited on 03/19/2024).

[17] LLVM Project. URL: https://llvm.org/docs/CommandGuide/llvm-objdump.html (visited on 06/11/2024).

[18] LLVM Project. URL: https://llvm.org/docs/CommandGuide/llvm-readelf.html (visited on 06/11/2024).

[19] LLVM Project. URL: https://llvm.org/docs/CommandGuide/llvm-dis.html (visited on 06/11/2024).

[20] URL: https://releases.llvm.org/15.0.0/docs/OpaquePointers.html (visited on 03/18/2024).

[21] LLVM Project. URL: https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl04.html#llvm-optimization-passes (visited on 03/19/2024).

[22] LLVM Project. URL: https://llvm.org/docs/NewPassManager.html (visited on 03/18/2024).

[23] LLVM Project. URL: https://llvm.org/docs/Passes.html (visited on 03/19/2024).

[24] LLVM Project. URL: https://llvm.org/docs/LinkTimeOptimization.html (visited on 03/19/2024).

[25] URL: https://releases.llvm.org/13.0.0/docs/ReleaseNotes.html (visited on 03/18/2024).

[26] URL: https://releases.llvm.org/17.0.1/docs/ReleaseNotes.html (visited on 03/19/2024).

[27] URL: https://plugins.jetbrains.com/docs/intellij/psi.html (visited on 01/22/2024).

[28] Amanda Hinchman-Dominguez. "Crash course on the Kotlin compiler — 1. Frontend: Parsing phase". In: *Google Developer experts* (2022). URL: https://medium.com/google-developer-experts/crash-course-on-the-kotlin-compiler-1-frontend-parsing-phase-9898490d922b.

[29] Andrey Polyakov. URL: https://blog.jetbrains.com/kotlin/2023/11/kotlin-1-9-20-released/#the-new-kotlin-k2-compiler-is-beta-for-all-targets (visited on 01/23/2024).

[30] Sergei Pecherkin. URL: https://youtrack.jetbrains.com/issue/KT-52604/Release-K2-Beta (visited on 01/23/2024).

[31] JetBrains s.r.o. *Annotations.kt*. Version v2.0.0-Beta2. Mar. 21, 2024. URL: https://github.com/JetBrains/kotlin/blob/v2.0.0-Beta2/kotlin-native/runtime/src/main/kotlin/kotlin/native/internal/Annotations.kt.

[32] JetBrains s.r.o. *KonanConfig.kt*. Version v2.0.0-Beta2. Mar. 21, 2024. URL: https://github.com/JetBrains/kotlin/blob/v2.0.0-Beta2/kotlin-native/backend.native/compiler/ir/backend.native/src/org/jetbrains/kotlin/backend/konan/KonanConfig.kt.

[33] JetBrains s.r.o. *kotlin-native/platformLibs/src/platform/linux*. Version v2.0.0-Beta2. Mar. 21, 2024. URL: https://github.com/JetBrains/kotlin/tree/v2.0.0-Beta2/kotlin-native/platformLibs/src/platform/linux.

[34] JetBrains. URL: https://kotlinlang.org/docs/native-platform-libs.html (visited on 03/21/2024).

[35] JetBrains. URL: https://kotlinlang.org/docs/native-c-interop.html (visited on 03/21/2024).

[36] JetBrains s.r.o. *TopLevelPhases.kt*. Version v2.0.0-Beta2. Mar. 21, 2024. URL: https://github.com/JetBrains/kotlin/blob/v2.0.0-Beta2/kotlin-native/backend.native/compiler/ir/backend.native/src/org/jetbrains/kotlin/backend/konan/driver/phases/TopLevelPhases.kt.

[37] JetBrains s.r.o. *NativeGenerationState.kt*. Version v2.0.0-Beta2. Mar. 21, 2024. URL: https://github.com/JetBrains/kotlin/blob/v2.0.0-Beta2/kotlin-native/backend.native/compiler/ir/backend.native/src/org/jetbrains/kotlin/backend/konan/NativeGenerationState.kt.

[38] JetBrains s.r.o. *BitcodeGeneration.kt*. Version v2.0.0-Beta2. Mar. 21, 2024. URL: https://github.com/JetBrains/kotlin/blob/v2.0.0-Beta2/kotlin-native/backend.native/compiler/ir/backend.native/src/org/jetbrains/kotlin/backend/konan/driver/phases/BitcodeGeneration.kt.

[39] JetBrains s.r.o. *ObjectFiles.kt*. Version v2.0.0-Beta2. Mar. 21, 2024. URL: https://github.com/JetBrains/kotlin/blob/v2.0.0-Beta2/kotlin-native/backend.native/compiler/ir/backend.native/src/org/jetbrains/kotlin/backend/konan/driver/phases/ObjectFiles.kt.

[40] JetBrains s.r.o. *BitcodeCompiler.kt*. Version v2.0.0-Beta2. Mar. 21, 2024. URL: https://github.com/JetBrains/kotlin/blob/v2.0.0-Beta2/kotlin-native/backend.native/compiler/ir/backend.native/src/org/jetbrains/kotlin/backend/konan/BitcodeCompiler.kt.

[41] RISC-V International. *RISC-V ELF Specification*. This is part of RISC-V Non-ISA Specifications. URL: https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-elf.adoc#reloc-table (visited on 06/09/2024).

[42] John R. Levine. *Linkers and Loaders*. 2.2. The Morgan Kaufmann Series in Software Engineering and Programming. Morgan Kaufmann, 1999. ISBN: 1558604960.

[43] *Java Native Interface Specification*. Available at https://docs.oracle.com/en/java/javase/22/docs/specs/jni/intro.html. Oracle. 500 Oracle Parkway, Redwood Shores, CA 94065 USA, 2024.

[44] RISC-V Foundation Andrew Waterman Krste Asanović. *"The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. Version 20191213. Dec. 2019. URL: https://drive.google.com/file/d/1s0lZxUZaa7eV_O0_WsZzaurFLLww7ou5/view.

[45] Inc. SiFive. URL: https://www.sifive.com/ (visited on 04/16/2024).

[46] LLVM Project. URL: https://llvm.org/docs/RISCVUsage.html (visited on 04/16/2024).

[47] Chris Simmonds Frank Vasquez. *Mastering Embedded Linux Programming - Third Edition: Create fast and reliable embedded solutions with Linux 5.4 and the Yocto Project 3.1 (Dunfell)*. 3rd ed. Packt Publishing Ltd., 2021. ISBN: 978-1-78953-038-4.

[48] *The GNU C Library (glibc)*. This is free and unencumbered software released into the public domain by David Burela. URL: https://sourceware.org/glibc/ (visited on 05/03/2024).

[49] et al. Tim Bird. *Toolchains*. 2023. URL: https://elinux.org/Toolchains (visited on 04/30/2024).

[50] Apple Inc. *Apple LLVM Branching Scheme*. Version v2.0.0-Beta2. Apr. 29, 2024. URL: https://github.com/apple/llvm-project/blob/apple/main/apple-docs/AppleBranchingScheme.md.

[51] URL: https://releases.llvm.org/14.0.0/docs/ReleaseNotes.html#changes-to-the-risc-v-target (visited on 03/18/2024).

[52] Free Software Foundation. *Using the GNU Compiler Collection*. Free Software Foundation. Feb. 2023. URL: https://gcc.gnu.org/onlinedocs/gcc-13.2.0/gcc.pdf.

[53] *CrossTool-NG: Introduction*. URL: https://crosstool-ng.github.io/docs/introduction/ (visited on 04/08/2024).

[54] *crosstool-NG: A versatile (cross-)toolchain generator*. URL: https://github.com/crosstool-ng/crosstool-ng (visited on 04/08/2024).

[55] JetBrains. URL: https://github.com/JetBrains/kotlin/commit/28a736ea3076157c099087f61023d11e3d6b6075 (visited on 04/08/2024).

[56] JetBrains. URL: https://github.com/JetBrains/kotlin/commit/333685c7ee18d4c0bb9c1d5bce75dbd3b02c4fed (visited on 04/08/2024).

[57] *GNU toolchain for RISC-V, including GCC*. Apr. 23, 2024. URL: https://github.com/riscv-collab/riscv-gnu-toolchain.

[58] URL: https://releases.llvm.org/11.0.0/tools/lld/docs/ReleaseNotes.html (visited on 02/09/2024).

[59] URL: https://reviews.llvm.org/D71820 (visited on 02/09/2024).

[60] JetBrains s.r.o. *DependencyDownloader.kt*. Version v2.0.0-Beta2. Apr. 23, 2024. URL: https://github.com/JetBrains/kotlin/blob/v2.0.0-Beta2/native/utils/src/org/jetbrains/kotlin/konan/util/DependencyDownloader.kt.

[61] cppreference.com was created and is maintained by a c++ open source community. URL: https://en.cppreference.com/w/cpp/utility/optional (visited on 04/23/2024).

[62] JetBrains s.r.o. *ThreadSuspension.cpp*. Version v5.0.0-Beta2. Apr. 23, 2024. URL: https://github.com/JetBrains/kotlin/blob/v2.0.0-Beta5/kotlin-native/runtime/src/mm/cpp/ThreadSuspension.cpp.

[63] JetBrains s.r.o. *SafePoint.cpp*. Version v5.0.0-Beta2. Apr. 23, 2024. URL: https://github.com/JetBrains/kotlin/blob/v2.0.0-Beta5/kotlin-native/runtime/src/mm/cpp/SafePoint.cpp.

[64] *Deprecating Vestigial Library Parts in C++17*. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0174r2.html (visited on 04/23/2024).

[65] Inc. Free Software Foundation. ABI Laboratory provides API/ABI changes analysis reports for most popular C, C++ and Java libraries. URL: https://abi-laboratory.pro/index.php?view=changelog&l=glibc&v=2.26 (visited on 04/22/2024).

[66] Inc. Free Software Foundation. ABI Laboratory provides API/ABI changes analysis reports for most popular C, C++ and Java libraries. URL: https://abi-laboratory.pro/?view=changelog&l=glibc&v=2.27 (visited on 04/22/2024).

[67] Inc. Free Software Foundation. ABI Laboratory provides API/ABI changes analysis reports for most popular C, C++ and Java libraries. URL: https://abi-laboratory.pro/index.php?view=changelog&l=glibc&v=2.28 (visited on 04/22/2024).

[68] Inc. Free Software Foundation. ABI Laboratory provides API/ABI changes analysis reports for most popular C, C++ and Java libraries. URL: https://abi-laboratory.pro/index.php?view=changelog&l=glibc&v=2.30 (visited on 04/22/2024).

[69] Inc. Free Software Foundation. ABI Laboratory provides API/ABI changes analysis reports for most popular C, C++ and Java libraries. URL: https://abi-laboratory.pro/index.php?view=changelog&l=glibc&v=2.33 (visited on 04/22/2024).

[70] Inc GitHub. llvm fork by apple on tag stable/20230725. URL: https://github.com/apple/llvm-project/tree/stable/20230725 (visited on 04/16/2024).

[71] LLVM Project. URL: https://llvm.org/docs/GettingStarted.html (visited on 03/18/2024).

[72] URL: https://reviews.llvm.org/D144970 (visited on 03/18/2024).

[73] Jan Svoboda. *[libclang] Split-out parts of Index.h*. Committed on 2022-09-29, released with LLVM 16.0.0-rc1. URL: https://github.com/llvm/llvm-project/commit/29e0435ac04957861aa1f85d41291c8b19db0122 (visited on 06/08/2024).

[74] Alexander Shabalin. *[K/N] Remove LLVM coverage*. Committed on 2023-12-06, released with build-2.0.0-241-1. URL: https://github.com/JetBrains/kotlin/commit/4f77434ea57fea4a2f8b49abf9c495447c34f15a (visited on 06/08/2024).

[75] Kazu Hirata. *[clang] Use std::optional instead of llvm::Optional (NFC)*. Committed on 2023-01-14, released with LLVM 16.0.0-rc1. URL: https://github.com/llvm/llvm-project/commit/6ad0788c332bb2043142954d300c49ac3e537f34 (visited on 06/17/2024).

[76] Dani Ferreira Franco Moura. *[clang][dataflow] Remove unused argument in getNullability*. Committed on 2022-12-16, released with LLVM 16.0.0-rc1. URL: https://github.com/llvm/llvm-project/commit/0da4cecfb6ad14ee0f0f9fa904e685fd6b64be60 (visited on 06/17/2024).

[77] cppreference.com was created and is maintained by a c++ open source community. URL: https://en.cppreference.com/w/cpp/language/copy_constructor (visited on 06/06/2024).

[78] URL: https://reviews.llvm.org/D128465 (visited on 06/08/2024).

[79] Cyris Kissane. URL: https://github.com/llvm/llvm-project/commit/d449c600767284486615f3b79601ced15a00af61 (visited on 06/08/2024).

[80] Announcements: LLVM 15.0.3 Released! URL: https://discourse.llvm.org/t/llvm-15-0-3-released/66036 (visited on 06/08/2024).

[81] Svyatoslav Scherbina. URL: https://youtrack.jetbrains.com/issue/KT-61299/Native-patch-LLVM-to-prevent-it-from-using-signal-handlers-incompatibly-with-JVM (visited on 05/06/2024).

[82] Free Software Foundation. *The GNU C Library: 24.2.1 Program Error Signals*. Version 2.38. Free Software Foundation. URL: https://www.gnu.org/software/libc/manual/html_node/Program-Error-Signals.html (visited on 05/06/2024).

[83] URL: https://reviews.llvm.org/D134567 (visited on 06/08/2024).

[84] Siva Chandra Reddy. URL: https://github.com/llvm/llvm-project/commit/215c9fa4deac9ec6b4e504843830551f03b60620 (visited on 06/08/2024).

[85] URL: https://reviews.llvm.org/D102136 (visited on 03/18/2024).

[86] URL: https://releases.llvm.org/15.0.0/docs/ReleaseNotes.html (visited on 03/18/2024).

[87] URL: https://releases.llvm.org/16.0.0/docs/ReleaseNotes.html (visited on 03/18/2024).

[88] *RISC-V Getting Started Guide: Linux on QEMU*. URL: https://risc-v-getting-started-guide.readthedocs.io/en/latest/linux-qemu.html (visited on 06/08/2024).

[89] Homebrew. *QEMU Formula — Homebrew*. URL: https://formulae.brew.sh/formula/qemu (visited on 06/08/2024).

[90] riscv-software-src. *homebrew-riscv*. URL: https://github.com/riscv-software-src/homebrew-riscv (visited on 06/08/2024).

[91] David Burela. *RISC-V Emulator Docker Image*. URL: https://github.com/DavidBurela/riscv-emulator-docker-image (visited on 06/08/2024).

[92] Chanchal Dey and Sunit Kumar Sen, eds. *Industrial Automation Technologies*. 1st. CRC Press, 2020. ISBN: 9780429299346. URL: https://doi-org.ezproxy.bib.hm.edu/10.1201/9780429299346.

[93] Hossameldin Eassa, Ihab Adly, and Hanady H. Issa. "RISC-V based implementation of Programmable Logic Controller on FPGA for Industry 4.0". In: *2019 31st International Conference on Microelectronics (ICM)*. 2019, pp. 98–102. DOI: 10.1109/ICM48031.2019.9021939.

[94] Aiken Pang and Peter Membrey, eds. *Beginning FPGA:Programming Metal*. Apress, 2017. ISBN: 978-1-4302-6247-3. URL: https://link-springer-com.ezproxy.bib.hm.edu/book/10.1007/978-1-4302-6248-0.

[95] Tiago Catalão and Mário de Sousa. "IEC 61131-3 Front-End for the LLVM Compiler Family". In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. 2020, pp. 1191–1194. DOI: 10.1109/ETFA46521.2020.9211921.

[96] Edouard Tisserant, Laurent Bessard, and Mario de Sousa. "An Open Source IEC 61131-3 Integrated Development Environment". In: *2007 5th IEEE International Conference on Industrial Informatics*. Vol. 1. 2007, pp. 183–187. DOI: 10.1109/INDIN.2007.4384753.

[97] Quentin Ducasse et al. "Porting a JIT Compiler to RISC-V: Challenges and Opportunities". In: *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR '22)*. https://hal.science/hal-03725841/document. Brussels, Belgium, Sept. 2022.

[98] ARM Limited. *ARM Architecture Reference Manual for A-profile architecture*. Version 21.0. LES-PRE-20349. ARM Limited. 110 Fulbourn Road, Cambridge, England CB1 9NJ, 2024.

[99] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. Intel Corporation. Mar. 2024. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[100] User avatar Sergey Bogolepov. URL: https://youtrack.jetbrains.com/issue/KT-67046/Upgrade-Kotlin-Native-GCC-toolchain-or-use-latest-LLD-for-linking (visited on 06/08/2024).

[101] Alexander Shabalin. *[K/N] Move llvm interop generation to kotlin-native/llvmInterop/*. Committed on 2024-03-05, released with build-2.0.0-Beta5-1. URL: https://github.com/JetBrains/kotlin/commit/bce550dce45d936d6c80da905d40d8320c80abba (visited on 06/08/2024).

[102] Troels Bjerre Lund. *Port to new llvm pass manager*. Committed on 2024-04-11, not released yet. URL: https://github.com/JetBrains/kotlin/commit/87c043ead295d11cc6f51885e402dc9ace15c413 (visited on 06/08/2024).