The Conversion of Source Code to Machine Code

Explaining the Basics of Compiler Construction Using a Self-Made Compiler

Silas Groh, Mik Müller

February 1, 2023

CFG Wuppertal

Abstract

Programming languages are undoubtedly of great importance for various aspects of modern-day life. Even if they remain unnoticed, digital systems running programs written in some sort of programming language are ubiquitous. However, there are numerous ways of implementing a programming language. A language designer could choose an interpreted or a compiled approach for their language's implementation. Both ways of program execution come with their own advantages and disadvantages.

This paper aims to inform the reader about different means of program execution, focussing on compiler construction. However, we will only focus on the basics since implementing a programming language is often a demanding task. In order to include practical examples, we will explain concepts on the basis of our own programming language called *rush*. During the implementation of rush, the focus for this paper has shifted slightly. As the title suggests, we originally planned on only implementing a compiler. However, there are numerous architectures which a compiler could target and settling on just one felt like the reader would miss out on too much. Therefore, we have implemented rush using one interpreter, one virtual machine, and five compilers.

In chapter 1, we will give an introduction to implementing a programming language. Moreover, the rush programming language and its characteristics are presented. In chapter 2, the process of analyzing the program's syntax and semantics is explained. Chapter 3 focuses on how interpreters can be used in order to implement an interpreted programming language. Here, we will differentiate between a *tree-walking interpreter* and a *virtual machine*. The latter also serves as the target architecture for one of the five compilers. Chapter 4 illustrates how compilation to high-level[1] targets works. As examples for high-level targets, we will present the compiler targeting the virtual machine, a compiler targeting *WebAssembly*, and a compiler which uses the popular *LLVM* framework. Chapter 5 focuses on how compilers targeting low-level[2] architectures can be implemented. For this, we will present a compiler targeting *RISC-V* assembly and another compiler targeting *x86_64* assembly. Lastly, Chapter 6 presents final thoughts and a conclusion on the topic of implementing a programming language.

[1] high-level targets: in this case machine independent

[2] low-level targets: specific to one target architecture and operating system

Contents

1 Compiling to Low-Level Targets

In the previous chapter, we have learned how compilation to high-level targets can present an easy way of implementing a compiler. These compilers generated outputs which were still platform independent and portable. However, compilers can also target a specific computer architecture directly, thus removing another layer of abstraction. The concept of compiling to a specific architecture directly is similar to the VM since its compiler also targets its architecture directly. However, implementing a compiler targeting the architectures presented in this chapter has proven to be a lot more demanding since the VM uses a fictional architecture purposefully developed for this paper. Reasons for this chapter's difficulty mostly include target-specific constraints which were still irrelevant in the previous chapter. This chapter contains the term "low-level" in its name since the presented compilers generate code for specific target-architectures directly.

1.1 Low-Level Programming Concepts

Programming using high-level languages does not require knowledge about the target architecture of the program. However, in this chapter, two compilers targeting the low-level assembly language are presented. In order to make sections in which these compilers are explained more approachable, some of the most important low-level programming concepts are explained in this section. We will only explain concepts which play a significant role later on.

1.1.1 Sections of an ELF File

Since a program needs to be representable in a low-level manner, special formats are often required. ELF stands for "executable and linkable format" and is often found on unix-like systems like Linux. Programs using the ELF format can be represented in three different types of files. For instance, object files generated by a compiler, like in the previous LLVM section, might use the ELF format. Furthermore, libraries using *shared object files* might also leverage the ELF format. Most executable program use the format in order to represent a structured container for instructions, data, and additional information. This way, the unit is mostly self-contained and can be executed by the operating system easily. Therefore, ELF describes the format of a class of files and not just of an individual type of file [Zhi17, p. 74-76].

Even though a processor only has access to one physical memory unit for both instructions and program data, most assembly programs like to separate these types of memory into their separate components. Therefore, an object file and assembly program is divided into so-called *sections* [Zhi17, p. 19]. In ELF programs, some of the important sections are:

- `.text` stores the logic of the program represented using CPU instructions
- `.rodata` stores read-only global data, it is often used for global constants.
- `.data` stores mutable global data, such as mutable global variables

Even though a typical ELF file also contains other sections, this list only includes entries which are of importance later in later sections of the paper [Zhi17, p. 76].

1.1.2 Assemblers and Assembly Language

Assembly language describes a type of low-level programming languages which are directly influenced by the target architecture. Since the assembly code provides a slight abstraction over the computer's hardware, the assembly code must be translated to machine code before it can be executed. This process is performed by a program named the *assembler* and is often called *assembly*. A typical characteristic of assembly code is that it seems relatively cryptic to a human reading it. Furthermore, compared to high-level languages like C, assembly code is relatively low-level since it can be used to interact with the hardware directly.

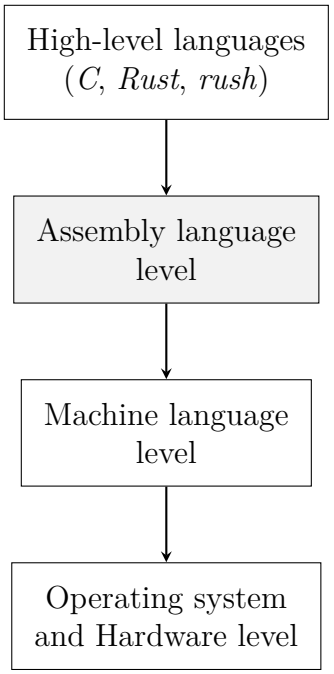

Figure 1.1 – Level of Abstraction provided by Assembly

For instance, the RISC-V instruction '`add a0, a0, 2`' would be used in integer addition. This example contains most characteristics of an assembly language program. Like hinted previously, the name of the instruction is a mnemonic. In this case, '`addi`' stands for "add immediate". Furthermore, the exact semantic meaning of the instruction is not immediately apparent. At last, the instruction for adding integers differs for most CPU architectures. For instance, an equivalent instruction for the x_86 architecture could be '`add %rdi, 2`'. Therefore, the fact that instructions differ on each target architecture is clearly apparent.

As seen above, the application code in form of instructions is placed in the '`.text`' section of an ELF binary. In most assembly dialects, the programmer is able to partition the code manually using sections. If the assembly code is assembled to an ELF object file, the '`.text`' section of the assembly should contain all the instructions. Just like parts of the LLVM IR, the '`.text`' section of an assembly program obeys a hierarchy. This section contains many labels which mark the beginning of a new basic block. However, these basic blocks do not come with all the constraints which are to be followed when using LLVM IR. For instance, a block in assembly might even contain no instructions, does not have to be terminated and could even be terminated twice. Here, terminating instructions mean jump- and return-instructions. Therefore, the constraints in assembly are much weaker than the ones introduced by LLVM. Because an assembly usually omits complex optimizations, struct rules constraining the assembly code can be omitted too.

Since assembly provides significantly less abstraction than high-level languages, the question of how much abstraction is lost emerges. In order to understand how much abstraction is provided by assembly, we should consider Figure 1.1. Here, the highest level of abstraction is provided by high-level programming languages like C or Rust. In the context of this paper, rush presents roughly the same level of abstraction like these languages. The next lower level of abstraction is provided by assembly. Now, the program is no longer independent of its target architecture and is much more demanding to furmulate. However, the next lower level of abstraction below assembly is represented by code in machine language. As of today, one only rarely encounters a programmer writing programs using this level of abstraction. Since the machine language program is represented in binary, it is nearly impossible for a human to write or understand. However, the machine language is also just an abstraction of the computer's hardware and operating system. Therefore, assembly provides enough

abstraction to be comprehensible for a human whilst being a low-level representation of the program. Although assembly provides more abstraction than the two levels at the bottom of the figure, programmers rarely program in assembly directly. Some benefits of using assembly language to formulate a program are increased runtime efficiency and decreased code size whilst having fine-grained control over the hardware and operating system. Therefore, it might be reasonable for a compiler to generate assembly code from the source program. This way, the program is translated into a low-level, target-specific representation which allows the program to be executed on the target machine directly. However, an assembler is still required in order to translate the assembly output of the compiler. Since most assemblers output object files, a linker is required to create the final executable program. Therefore, a compiler targeting the assembly of a specific architecture depends on these two additional steps before the program can be executed [Dan05, p. 5-6].

One might argue that the compiler could output object files directly. However, doing so rarely creates any significant benefits other than the omitted dependence on the assembler. Furthermore, implementing a compiler using this approach often significantly increases the complexity of the compiler since it now has to perform the role of the assembler as well. However, the compiler could also emit binary data directly, thus making its implementation significantly more demanding.

1.1.3 Registers

Most processors contain numerous registers in order to hold data, instructions, and state information. In a computer, registers are sometimes used to hold data stored in variables. However, most of the time, registers are used as temporary storage in large computations. Registers are used in the latter case because they are way faster than memory. Therefore, an algorithm using registers instead of memory will usually run faster. However, even though registers are faster than memory, they are not used all the time. This is because CPU registers are a limited resource, meaning every register-based CPU only has a finite amount of registers available. Registers should therefore be used carefully and only when they are really needed [Wat17, pp. 212-214].

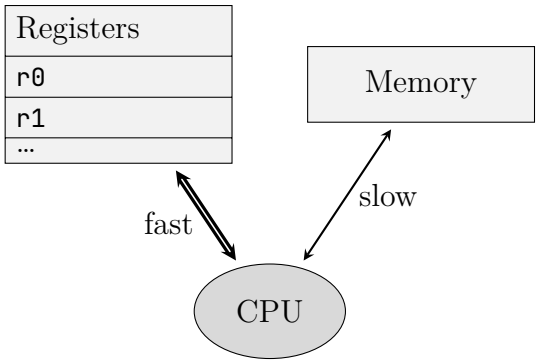

Figure 1.2 – Relationship Between Registers, Memory, and the CPU

Figure 1.2 shows how the CPU interacts with its registers and the computer's memory. The connection between the set of registers and the CPU is marked as a short and thick arrow because the connection between the CPU and its registers is very short and fast. Although the registers are often physically parts of the CPU, they are displayed as separate entities in this example. For the memory, the arrow is long and thin. This represents the long and therefore slow connection between the CPU and the memory modules. In most modern computers, this connection often spans across several centimeters on the motherboard. Therefore, higher latency during memory access is inevitable [Dan05, pp. 20-21].

Nearly every CPU architecture will include an individual register layout, differing in size, count, and type. Therefore, the exact information about registers always depends on which CPU is used. As a rule of thumb, having more registers in an architecture will almost always improve performance of that architecture. Furthermore, not every register is identical. There might be platforms with some general-purpose, some floating-point, and some special-

purpose registers. For instance, there may be registers which contain information about the CPU's current instruction. Furthermore, common architectures also include special-purpose registers for modifying the machine's memory. Therefore, there is nearly always an appropriate register matching the requirement [Dan05, Chapter 2].

```
4 li a0, 40
5 li a1, 2
6 add a0, a0, a1
```

Listing 1.1 – Example Assembly Program for Explaining Register Allocation

For a compiler, the limited amount of registers presents a big challenge. Some architectures may require that the operands of arithmetic or logical instructions have been explicitly loaded into registers beforehand. In this case, registers would be required in nearly all computations. In order to understand how registers management can present a challenge, Listing 1.1 should be considered. This snippet displays RISC-V assembly instructions which calculate the sum of the two integers 40 and 2. In line 4, the integer value 40 is placed in the register '`a0`'. In line 5, another integer, this time 2 is placed in '`a1`'.

Next, the '`add`' instruction is used to calculate the sum of these two integers. Since the first operand of the instruction specifies the register in which the result should be placed, the original value of 40 in the register '`a0`' would be overwritten by the instruction [PW17, reference]. Through this example, it becomes apparent that other instructions might use registers unexpectedly. Therefore, subtle bugs can be created if an instruction overwrites registers.

In compilers, the process of *register allocation* is responsible for managing how registers are used. This process attempts to use registers in a way leading to maximized efficiency of the output program. Since accessing registers is often faster, a register allocator will try to use registers as often as possible. Production-ready compilers often try to keep as much of the frequently accessed data in registers. For instance, this frequently accessed data may also include variables which are normally saved in memory. It is apparent that in most programs, the number of variables will certainly exceed the capacity provided by registers. Therefore, register allocation has to detect when no free registers are available anymore. In this case, the compiler has to save data in memory instead of registers. This process of saving excess data in memory instead of registers is called *register spilling*. Since register spilling introduces a performance penalty, register allocation algorithms often attempt to prevent it as much as possible. Therefore, sophisticated algorithms for register allocation are often mandatory as long as the factor of output code performance is non-trivial.

Apart from just managing the use of registers, most allocation algorithms are responsible for many other register-related tasks. For instance, register allocation should detect when a variable is no longer needed so that its register can be freed. Moreover, register allocation has to ensure that no conflicts between registers are introduced. Such a conflict may be that a register is accidentally overwritten by an instruction in a completely unrelated basic block. It is apparent that an algorithm performing all these tasks can not be implemented in an ad-hoc manner. Instead, this process often requires complex graph algorithms for determining which registers can be used and freed. Therefore, implementing register allocation in a compiler is often a very demanding task [Wat17, pp.212-214]. Since register allocation represents a complex topic, it will not be explained any further.

1.1.4 Using Memory: The Stack and the Heap

TODO: @RubixDev: write this subsection TODO: Useful information: [PH17, p. 99]

- Repetition: When is the stack used instead of registers?

- Insert very figure of the stack.
- Only mention that there are stack pointers and why they are needed; do not explain how they point / behave exactly
- Stack can be indexed

1.1.5 Calling Conventions

Programmers often use *procedures* or *functions* in order to structure their programs, both to make the program easier to understand and to allow shared logic to be reused. Therefore, procedures allow the programmer to focus on one portion of the tasks at the time. Often, parameters and returned values act as the interface between the procedure and the remaining code. Therefore, procedures present one way of how a high-level programming language might provide abstraction. During a procedure call in a high-level language like rush, many individual steps need to be performed in the program's low-level representation. Since assembly does not support the use of high-level functions, most architectures specify their own *calling convention* which describes how low-level function calls are to be performed. On most architectures, a low-level procedure call might take place like the following steps:

- The caller places the call arguments in a place where the procedure can access them.
- Control is transferred to the procedure, often using a jump or specialized call instruction. This specialized instruction often saves the *return address*[1] in a special register so that function returns are easier.
- The first task the called function performs is acquiring local storage resources needed by the procedure. Often, registers are spilled if required. Furthermore, stack memory for the procedure's local variables is often allocated in this step.
- Internal code of the procedure is run by executing the function body's instructions.
- The procedure's result value is placed somewhere so that the caller can access it. Here resources allocated in step 3 are also released again.
- Control is transferred back to the caller using a specialized return-instruction. A specialized instruction is often required since procedures can be called from multiple places in the program. Therefore, the return-instruction jumps back to the instruction which had initiated the function call.

In order to leverage optimal performance during function calls, passed arguments are usually kept in registers. However, in the previous subsection about registers, we have learned that most architectures state that only subset of their registers are to be used as function arguments. Therefore, if a function is called using more than eight arguments, every remaining argument has to be spilled to memory. In this case, each additional parameter of the function is placed inside the stack frame of the called function so that it can access it. [PH17, p. 98].

For instance, a target architecture might provide *four* registers which can hold function arguments. Now, a function is called using *six* arguments. Here, register spilling is mandatory since there are two more arguments than registers. Figure 1.3 illustrates how such a call would be managed on this fictional architecture.

Here, the registers '`r0`' — '`r3`' contain the first four argument values. The excess arguments which could not fit into registers are now placed on the stack. Even though argument 5 and 6 are placed on top of the stack in this example, the exact location of spilled call arguments differs for every architecture.

[1] A link to the caller site, allows the called procedure to jump back to the caller [PH17, p. 99].

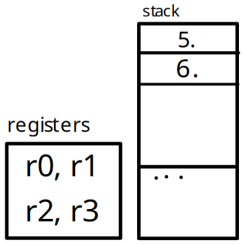

Figure 1.3 – DRAFT: Spilling Registers for Function Calls

For managing the calling convention in assembly, most functions contain a *prologue* and an *epilogue*. These blocks of the function are responsible for allocating and deallocating resources which the function might use. For instance, the stack- and frame-pointer is often adjusted in the prologue. This way, space on the stack is allocated for variable declarations found in the called function. However, the modification offset always depends on how much memory the called function will require during runtime. Therefore, a compiler would have to keep track of the count of variable declarations made by a function. Apart from allocating stack space, the prologue is also often used for storing special registers on the stack.

The epilogue however is required for deallocating all resources which were previously allocated by the function's prologue. For instance, the stack-related pointers are modified to reflect their state before the function call. Furthermore, any special registers previously saved on the stack are now restored. Obviously, a return-statement should always jump to the function's epilogue instead of terminating the function directly. At the end of most epilogues, a return-instruction jumping back to the caller location is often found. Therefore, the prologue is executed as the first code when calling the function and the epilogue is called as the last code before this function terminates.

It is to be mentioned that implementing a compiler which respects the calling convention of its target architecture is not required. However, following the convention is often reasonable in order to preserve ABI[2] compatibility of the compiled program.

[2]Short for "application binary interface", allows calling foreign functions from another program

1.2 RISC-V: Compiling to a Modern RISC Architecture

The *RISC-V ISA*[3] is a new and modern *reduced instruction set* architecture focussing on simplicity and expandability. The initial version was developed at *UC Berkely* in the context of another related research project. Since its introduction in 2011, the architecture has been rapidly growing in popularity. Since the beginning, the project has been managed and led by the *RISC-V foundation*, consisting of many individuals contributing to the project. Today, corporate members of the RISC-V foundation include companies like *Google*, *Microsoft*, *Samsung*, and *IBM*. Therefore, the general popularity and commercial attraction of the technology is apparent. However, unlike most previous ISAs, the RISC-V architecture is a completely *open-source* project and is therefore not controlled by a single large corporate entity. This can be regarded as a large competitive advantage over other popular RISC architectures like *ARM*. In the past, many ISAs have failed due to them being too restrictive with their licensing, thus preventing widespread commercial adoption. However, RISC-V is completely open and free to use, so that many companies like *Google* can leverage the technology commercially whilst contributing to the project. Unlike most of the previous ISAs, which were developed during the 1970s or 80s, RISC-V is one of the few ISAs which were developed during this decade. Therefore, it seems like RISC-V could be a significant architecture to be used in all sorts of devices in the near future [PW17, preface].

1.2.1 Register Layout

Most RISC architectures typically have a large number of registers [Dan05, Chapter 2]. When compared to other popular architectures, the truth of this statement becomes clear. For instance, the *x86_32* architecture has 8 registers. The popular RISC architecture *ARM-32* contains twice that amount, meaning 16 registers. However, RISC-V includes 32 registers, meaning drastically more than the previously mentioned architectures. Moreover, these 32 registers only include registers holding integer values. Just for floating-point numbers, the ISA even provides another 32 registers. Like previously explained, using more registers usually leads to increased efficiency of the output program. Therefore, a register allocation algorithm targeting the RISC-V architecture could be more aggressive compared to one targeting x_86 for instance.

Table 1.1 – Common Registers in RISC-V [WA19, p .155]

Register	Purpose
`zero`	Hardwired zero
`ra`	Return address
`sp`	Stack pointer
`t0` — `t6`	Temporary
`fp`	Frame Pointer
`a0`, `a1`	Function argument, return value
`a2` — `a7`	Function argument
`s1` — `s11`	Saved register
`fa0`, `fa1`	FP args, return value
`fa2` — `fa7`	FP args
`fs0` — `fs11`	FP saved registers
`ft0` — `ft11`	FP temporaries

The Table 1.1 shows most of the registers which the RISC-V platform provides. For the registers in this table, their official ABI names have been used in order to make the section easier to read. The '`zero`' register is a special integer register on RISC-V. Like its name suggests, it always holds the value of a constant 0. Unlike other registers, it is read-only, meaning that it can never be overwritten by accident.

The '`ra`' register saves the *return address*. For instance, this register is used when the '`call`' instruction is executed. First, the address of the instruction after the call-instruction is saved in '`ra`'. Next, the CPU jumps to the instruction at the specified location. If a

[3]Short for: "instruction set architecture"

return-instructions was to be used later, the value in '`ra`' would be used to jump back to where the call was made.

The '`sp`' and '`fp`' registers are used for managing stack memory. Their purpose is explained in a following subsection. The '`t0`' — '`t6`' registers are often used to store temporary values used in larger computations. Next, the register '`a0`' and '`a1`' can both serve as function call arguments and return values of functions. How functions are called using registers will also be explained in a following subsection. The remaining a-registers '`a2`' — '`a7`' can only be used as function call arguments. The *saved* registers '`s1`' — '`s11`' are typically preserved across function calls. However, this concept will also be explained in more detail later. All the previously named registers hold integer values. Depending on the exact RISC-V architecture, all registers either hold 32-bit or 64-bit integers.

Just like their integer counterparts, the floating-point registers '`fa0`' and '`fa1`' are used as function call arguments and as return values. The registers '`fa2`' — '`fa7`' can only be used as function arguments holding floating-point numbers. Just like the '`sx`' registers, the '`fs0`' — '`fs11`' registers are usually preserved across function calls. Last, the '`ft0`' — '`ft11`' registers can be used as temporary registers for floating-point numbers. [PW17, pp. 18f, p. 34], [WA19, p .155].

Now, it has become apparent that RISC-V includes many registers grouped into categories. Every category is meant to be used in the specified manner, however, these groups are merely a suggestion of how each register should be used. More information on the special registers like '`sp`', '`fp`', and '`ra`' is given in the next sections.

1.2.2 Memory Access Through the Stack

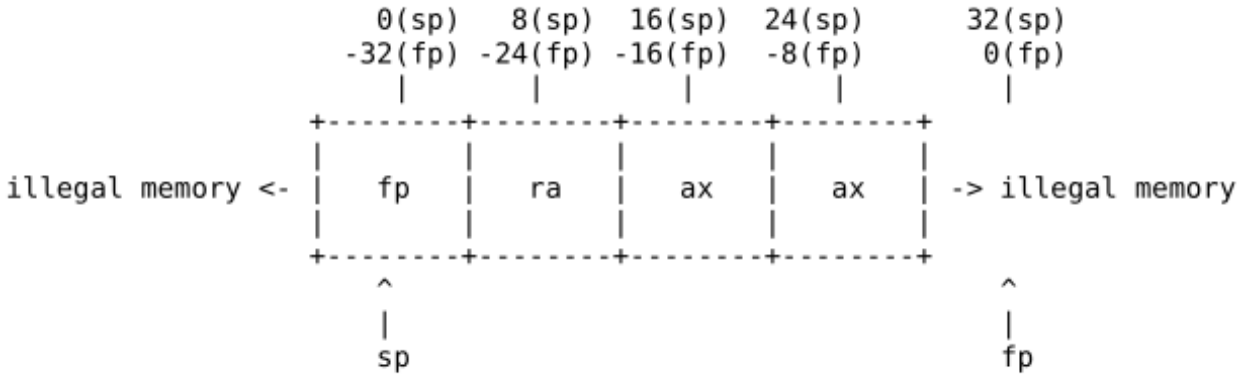

Figure 1.4 – DRAFT: Stack Memory on the RISC-V architecture

- In the current implementation, '`sp`' includes legal memory locations
- '`fp`' points to the first address after the last legal one
- The stack grows into lower memory regions

1.2.3 Calling Convention

Just like previously explained, most architectures provide a calling convention which dictates how low-level function calls should be managed. For most architectures, the calling convention is part of the ISA's official specification. In the case of RISC-V, the calling convention is specified in a separate document [22].

The first step of calling a function involves placing the arguments in a place where the function can access them. For RISC-V, this involves placing the arguments into specialized registers. Like described in the Table 1.1, only special classes of registers can be used as call arguments. For integer arguments, the first arguments are placed in the registers '`a0`'–'`a7`' For instance, the first two arguments of the rush function call '`foo(40, 2, 3.14)`' would be placed in the registers '`a0`' and '`a1`'. However, the third argument is a floating-point number and can therefore not be placed inside an integer register. Therefore, the first floating-point argument register '`fa0`' contains the argument '`3.14`'. In this case, all arguments can be held in regisers and spilling would not be required.

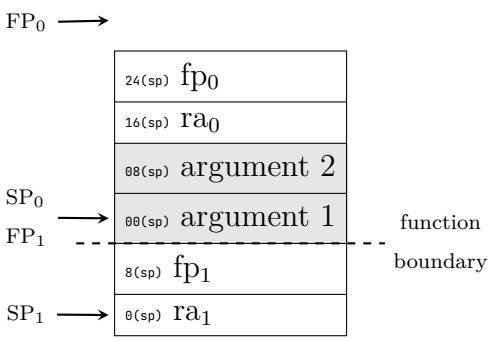

Figure 1.5 – Spilled Registers During a RISC-V Function Call **TODO: Every row should be equal in height**

In case the function accepted nine or more integer arguments, all further integer arguments upward of the ninth position would have to be spilled on the stack. Here, the successive registers '`a0`'–'`a7`' would contain the first eight integer arguments of the called function. The argument at position 9 however is then spilled on the stack since there are no registers left which could contain the additional argument [22, p .8].

The Figure 1.5 displays a possible state of the call stack during a function call which uses ten integer arguments. If ten integer arguments are used, two arguments would have to be spilled on the stack. In the figure, the spilled registers are placed in the stack cells "argument 1" and "argument 2". Here, the cell "argument 1" would hold the ninth argument whilst "argument 2" holds the tenth argument. Therefore, all spilled argument registers will be placed in the stack frame of the caller function. Normally, variables saved on the stack are aligned to reflect their sizes. In case of spilled argument registers however, every argument will occupy exactly 8 bytes on the stack, even if the data type itself requires less space.

Now that the first step of a procedure call is explained, the question of how the second step works in RISC-V remains. In the second step, the underlying procedure call is made using a specialized instruction. In RISC-V assembly, one typically uses the call *pseudoinstruction*[4]. Due to a lack of functions in assembly, the call-instruction uses the name of its target label as one operand. Therefore, labels can be called as if they were functions. This instruction will jump to the first instruction of the specified target label whilst saving the address of the next instruction after the '`call`' instruction in the register '`ra`' [PW17, p. 22]. As hinted previously, the '`ra`' register saves the *return address*. Therefore, the return address is set every time a function call is performed.

During the third step of the function call, the called function acquires local storage resources. To be precise, the function decrements the stack pointer by the amount required by the stack frame. Therefore, the function allocates as much stack space as required for storing local variables and other data. Additionally, the frame pointer and return address are saved on the stack so that nested function calls do not cause issues. For instance, if the return address was not saved on the stack, a nested function call would overwrite its stored value. In this case, the parent function could no longer return since the return address now holds an incorrect value. In order to mitigate issues like this, the return address and frame pointer are saved on the stack. Figure 1.5 shows that the two registers are saved at the two positions on the top of the new stack frame. This part of the function is often called the

[4]A macro generating multiple instructions from one pseudoinstruction. Therefore, the actual count of ISA instructions remains low whilst convenience features can be used in assembly [Dan05, p. 68].

prologue as it is executed before any of the function's internal code.

After the code of the function has been executed, the so-called *epilogue* is executed. Since the frame pointer and return address have been saved on the stack during the prologue, the epilogue restores these registers by loading their values from the stack. Furthermore, the amount which was subtracted from the stack pointer in the prologue is now added to the pointer in order to restore it to its original state before the function call. Here, incrementing the stack pointer represents deallocating the previously acquired stack space. However, the memory in the stack frame is not actually deleted since only the stack pointer is modified. Even though the used memory is not explicitly deleted, it is still freed since it will probably be overwritten by the next function call. If the prologue would not save the return address on the stack, a nested function call would overwrite the return address of the parent function, therefore creating a bug [PW17, p.33]. By saving the return address on the stack, nested function calls do not cause difficulties. It is apparent that this design contains a lot of similarities to the call-stack of the rush VM. However, in the VM, the process of saving and restoring the return address was managed automatically by the VM, whereas, here, the programmer has to manually pay attention to saving and restoring this important piece of data. Therefore, implementing function calls is definitely more demanding in RISC-V assembly than in the rush VM.

In case a function returns a value, it must be communicated to the caller so that it can access it. For integer-based types, the first return value of a function is placed in the register '`a0`' whilst floating-point numbers are placed in the register '`fa0`'. This way, the caller code can obtain a function's return value by accessing the '`a0`' and '`fa0`' registers respectively. If a function does not return a value, these steps are just omitted. It is to be mentioned that character and boolean values are also placed inside the '`a0`' register since these types can be represented by integers.

Lastly, the epilogue contains a return-instruction which should jump to the place where the function was called. This *ret* instruction reads the value stored in '`ra`' in order to jump to this address.

1.2.4 The Core Library

Like hinted in the section about the linker on page **??**, a program might use functionality provided by external libraries. In case of the rush RISC-V compiler, external functions are used for character-arithmetic, the mathematical power operator, and *kernel*[5] system-calls, like *exit*[6]. Since these concepts must introduce additional logic, the compiler should not be emitted their instructions everytime they are used. In that case, the repeated emission of redundant instructions would result in enlarged and unnecessary complex output code. In order to mitigate these issues, the compiler simply inserts call-instructions referencing external functions. External functions can be called just like any other function, however, their definition is not found in the same assembly file. As described previously, resolving these external calls is later handled by the linker. For this compiler, we later refer to this target specific library code by the term *corelib*.

For instance, a function for mathematical power operations is implemented in the corelib. Therefore, the compiler can emit a procedure call to this method every time rush's '`**`' operator is used in the source program. For this project, the entire corelib is written in RISC-V assembly. However, it is often rational to implement a corelib or standardlib using a high-level language like C. Since the corelib's functions are specified in separate files, they

[5] **TODO: Explaination + Citation**

[6] Terminates the execution of the program with an arbitrary exit code

are packaged into an *archive file*[7] which is later used by the linker.

1.2.5 RISC-V Assembly

The Listing 1.2 shows a rush program containing two functions and a global variable. In line 1 of the rush program, the mutable global variable '`m`' is defined with the initial value 42. In line 4 of the main-function, '`m`' is incremented by 1. Next, in line 5, the '`foo`' function is called using '`m`' as the only call argument. **TODO: continue here** In line 6, a return-statement is used to terminate the main-function explicitly. The body of the '`foo`' function only contains a call to the '`exit`' function. Therefore, the '`foo`' function only exits using the specified parameter '`n`' as the exit code. In this case, the exit code of the displayed program will be 43. The code in Listing 1.6 on page 12 shows the output assembly generated from this program by the rush RISC-V compiler[8]. Because the assembler code of the '`foo`' function would take up too much space in the assembler program, it is intentionally omitted from this listing. Since the excluded function does not introduce any new concepts anyway, omitting it will not lead to a loss of explained concepts.

```
1  let mut m = 42;
2
3  fn main() {
4      m += 1;
5      foo(m);
6      return;
7  }
8
9  fn foo(n: int) {
10     exit(n)
11 }
```

Listing 1.2 – Example rush Program Containing Two Functions

In line 1, the '`.global`' assembler directive is used to declare the global symbol '`_start`' [PW17, p .36]. On most architectures, the '`_start`' label indicates a program's entry point, therefore marking the first instruction to be executed [Zhi17, p. 19]. In line 5, the '`_start`' label is defined by placing a colon after its name. In line 6, the '`call`' instruction is used to call the '`main..main`' function. What strikes the eye here is that the already familiar '`main`' function is prepended by the '`main..`' prefix. Since this rush compiler implements name mangling[9], every function declared in a rush program will contain this prefix. However, unlike high-level function calls in LLVM, this call instruction is used alongside the previously explained low-level calling conventions of RISC-V.

In the next line, the '`li`' instruction is used to load the constant integer 0 into the register '`a0`' [PW17, reference]. Like explained in the previous section about the RISC-V calling convention, the register '`a0`' is used for the first integer call argument. In line 8, the '`exit`' function is called, however, one cannot see the definition of this function in the current file. This is because the exit function is provided by the rush RISC-V corelib which was explained previously. Since 0 was previously placed inside the register for the first integer call argument, the '`exit`' function is called using 0 as the argument. Therefore, the instructions in the lines 7–8 are responsible for terminating the program using the exit code 0. These

[7] **TODO: What are archives + Citation**
[8] Generated in Git commit `a0358c4`, automatically built with this document
[9] Compilers often *mangle* names in order to create a unique name for every function [Lev00, pp. 119-120]

two instructions are always inserted at the end of the '`_start`' label in order to terminate the program appropriately in case the rush code does not call '`exit`' on its own. This is required in order to prevent a segmentation fault which occurs if the program is not terminated properly.

```
1  .global _start
2
3  .section .text
4
5  _start:
6      call main..main
7      li a0, 0
8      call exit
9
10 main..main:
11     # begin prologue
12     addi sp, sp, -16
13     sd fp, 8(sp)
14     sd ra, 0(sp)
15     addi fp, sp, 16
16     # end prologue
17     # begin body
18     li a0, 1
19     ld a1, m                          # m
20     add a2, a1, a0
21     sd a2, m, t6
22     ld a0, m                          # m
23     call main..foo
24     j epilogue_0                      #
   ↪   return
25     # end body
26
27 epilogue_0:
28     ld fp, 8(sp)
29     ld ra, 0(sp)
30     addi sp, sp, 16
31     ret
   # ...
53 .section .data
54
55 m:
56     .dword 0x000000000000002a  # = 42
```

Figure 1.6 – Compiler Output from the Rush Program in Listing 1.2

Due to the function call in line 6, we will now shift our focus on the '`main..main`' label in line 10. In line 11, the first line of the '`main`' function, a comment indicates the beginning of the function's prologue. Just like demanded by the RISC-V calling convention, the rush compiler emits code for a *prologue* and an *epilogue* into each function.

As described in the previous sections about calling conventions, one task of the prologue is allocating stack space. In this prologue, the '`addi`' instruction in line 12 subtracts 16 from the value stored in the '`sp`'. Here, an addition instruction is used even though subtraction is required. In RISC-V, the '`addi`' instruction requires one register and one immediate value as its operands. Due to the third operand being an *immediate* value, the trailing '`i`' (*immediate*) appears in the instruction's name. Since this immediate value can be negative, an additional instruction for immediate subtraction is redundant [PW17, reference]. This example shows how the RISC-V ISA omits redundant instructions in places where it is feasible. In this case, the stack pointer is decremented by 16 since two 8-byte values are stored on the stack in the lines 13 and 14. Here, the values of the registers for the frame pointer and the return address are saved on the stack.

The comment in line 17 indicates the start of the function's body. First, the previously explained '`li`' instruction in line 18 places a constant 1 in register '`a0`'. Next, the '`ld`' instruction in line 19 is used in order to load the value of the global variable '`m`' into the register '`a0`' [PW17, reference]. Global variables, like '`m`' in this example are saved under the '`.rodata`'. section or under the '`.data`' section if they are mutable. In this example, '`m`' is not declared as mutable and therefore saved under the '`.rodata`' section. The start of the '`.rodata`' section is represented by the '`.section`' assembler directive found in line 53. Here, a label called '`m`' is defined. In this label, the '`.dword`' directive is used to define the global initializer value of the variable. In RISC-V, this directive stores 64 bit of information in successive memory doublewords [PW17, p. 39]. The initializer value of the global variable is 42 and is represented as '`0x2a`' using hexadecimal in the assembly code. Since these data labels require their contents to be specified in hexadecimal, the trailing comment shows the base 10, human-readable version

of the number. Because global variable are not saved on the stack, special instructions like '`ld`' are required to interact with global variables stored in the program's data sections.

At this point, the register '`a0`' would contain 1 and '`a1`' would contain 42. In line 20, the '`add`' instruction is used in order to save the sum of '`a0`' and '`a1`' in the register '`a2`'. Now, the value saved in '`a2`' would be 43. Next, the '`sd`' instruction in line 21 saves the value of the register '`a2`' at the memory location of the global variable '`m`', meaning that '`m`' is updated to reflect its new value 43. It now becomes apparent that these instructions are responsible for the add-assign expression in line 4 of the rush program. Another interesting observation is that the last operand of the '`sd`' instruction specifies the temporary register '`t6`'. The instruction uses this register for saving temporary data during the process of saving data in '`m`' [PW17, reference].

In line 22, the previously explained '`ld`' instruction is used in order to load the value of the same variable into the register '`a0`'. Then, the '`call`' instruction in line 23 is used in order to call the '`foo`' function using the value of m as its argument. However, one cannot easily observe how call arguments are passed here. Like explained previously, the first integer argument of a function call must be placed in the register '`a0`'. Since '`m`' was loaded into '`a0`' previously, it will be used as the call argument for '`foo`' automatically. Therefore, the '`foo`' function is called using 43 as the first argument.

Since the '`foo`' instruction only calls the '`exit`' function, its explanation will not be beneficial for introducing new concepts. Therefore, we will omit the explanation of the assembler output of the '`foo`' function. The final instruction of the main-function's body is the '`j`' instruction in line 24. This instruction will cause the CPU to jump to the address of the specified label. In this example, the CPU will jump to the first instruction of the '`epilogue_0`' label [PW17, p. 17]. Therefore, the rush compiler uses the '`call`' instruction for jumps caused by function calls and the '`j`' instruction for jumps between blocks of the current function.

Like explained previously, every function has a *prologue* and an *epilogue*. Since one of the tasks handled by the epilogue is releasing resources allocated by the prologue, the function's stack pointer is incremented in line 30. Finally, the '`ret`' instruction in line 31 is used in order to jump back to the instruction whose address is specified in the '`ra`' register [PW17, reference].

1.2.6 The rush Compiler Targeting RISC-V Assembly

1.3 x86_64: A Compiler for a CISC Architecture

- Present Some example instructions
- Register layout & count
- Calling convention
- INFORMATION: x86: An intel blog stated ca. 3600 Instructions in 2015 [RA17]
- RISC-V Reader: *ASCII adjust after addition* → *aaa* → obsolete, collectively occupy 1.6% of opcode space[PW17, p. 4]
- "Modular vs. Incremental ISAs Intel was betting its future on a high-end microprocessor, but that was still years away. To counter Zilog, Intel developed a stop-gap processor and called it the 8086. It was intended to be short-lived and not have any successors, but that's not how things turned out. The high-end processor ended up being late to market, and when it did come out, it was too slow. So the 8086 architecture lived on—it evolved into a 32-bit processor and eventually into a 64-bit one. The names kept changing (80186, 80286, i386, i486, Pentium), but the underlying instruction set remained intact. —Stephen P. Morse, architect of the 8086"[Mor17]. ⇒ only meant as a temporary architecture

1.3.1 Register Layout

For registers: [Kus18, p. 7]

1.3.2 Memory Access Through the Stack

1.3.3 Calling Convention

1.3.4 X86_64 Assembly

1.4 Conclusion

1.4.1 RISC vs CISC Architectures

TODO: RISC vs. CISC [Dan05, p. 5-6]

- RISC-V was less demanding and better

List of Figures

List of Tables

List of Listings

Bibliography

[Lev00] John R. Levine. *Linkers and Loaders*. San Francisco, CA: Morgan Kaufmann, 2000.

[Dan05] Sivarama P. Dandamudi. *Introduction to Assembly Language Programming: For Pentium and RISC Processors*. 2nd ed. Springer International Publishing, 2005. ISBN: 0-387-20636-1.

[Mor17] Stephen P. Morse. "The Intel 8086 Chip and the Future of Microprocessor Design". In: (Apr. 2017), pp. 8–9. DOI: 10.1109/MC.2017.120.

[PH17] David A. Patterson and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, Apr. 2017. ISBN: 978-0-12-812275-4.

[PW17] David A. Patterson and Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon LLC, Oct. 2017. ISBN: 978-0-9992491-0-9.

[RA17] Steven Rodgers and Richard A. Uhlig. "X86: Approaching 40 and Still Going Strong". In: (June 2017).

[Wat17] Des Watson. *A practical approach to compiler construction*. en. 1st ed. Undergraduate Topics in Computer Science. Cham, Switzerland: Springer International Publishing, Apr. 2017. ISBN: 978-3-319-52787-1.

[Zhi17] Igor Zhirkov. *Low-Level Programming: C, Assembly, and Program Execution on Intel® 64 Architecture*. 1st ed. Saint Petersburg, Russia: APress, June 2017. ISBN: 978-1-4842-2403-8.

[Kus18] Daniel Kusswurm. *Modern X86 Assembly Language Programming*. 2nd ed. Geneva, IL, USA: APress, Dec. 2018. ISBN: 978-1-4842-4063-2.

[WA19] Andrew Waterman and Krste Asanović. "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA". In: (Dec. 2019). Ed. by Andrew Waterman, Krste Asanović, and Sivive Inc. URL: https://riscv.org/technical/specifications/.

[22] *RISC-V ABIs Specification*. Nov. 2022. URL: https://github.com/riscv-non-isa/riscv-elf-psabi-doc.