Bachelor Thesis

Porting Selfie to RISC-V

Native Toolchain Support

Christian Barthel
`bch@barthel.ch`

University of Salzburg
Department of Computer Sciences
Austria

Advisor: Professor Christoph Kirsch

June 12, 2017

Contents

Abstract

During the last year, I was part of a group that worked on *The Selfie Project* of the Computational Systems Group at the Department of Computer Sciences of the University of Salzburg in Austria.
The goal of this project is to provide an educational system that consists of a simple self-compiling compiler, a self-executing instruction set emulator and a minimal self-hosting hypervisor, based on the RISC-V instruction set. Its main purpose is to introduce students to principal systems engineering techniques and teach students about computer architecture, compilers and operating systems. Everything is designed to be self referential, to allow students to discover the intrinsics of system code which is known to be a big challenge.
During this project, I worked on adding the ELF32 binary format to the compiler and emulator and making the overall system compatible with the RISC-V environment. Beside adding the ELF header, this involved also code adaption and synchronization with the operating system interface provided by the `pk` kernel. The work bridges the gap between the existing toolchain that is based on an artificial RAW binary format, and the RISCV32 GNU toolchain which is widely used on various operating systems and computer platforms.
It is now possible to run binaries generated by `starc` on top of the RISC-V instruction set reference implementation `spike`. Furthermore, it should be simple to run binaries directly on RISC-V microprocessors.

1. Introduction

1.1. Problem Description

With the transition of the original MIPS Instruction Set Architecture used by Selfie to RISC-V, described in [1], it became desirable to make Selfie-generated binaries compatible with tools of the RISC-V toolchain. Some advantages are:

- Selfie binaries run on another emulator with another operating system interface.
- It provides an additional possibility to spot errors.
- Selfie may run natively on RISC-V microprocessor chips.
- Debugging and analysis tools provided by RISC-V are readily usable.
- Move Selfie from an educational system to a real world toolchain.
- Paves the way for further opportunities like using the GNU `ld(1)` linker system.

The goal therefore is to make Selfie compatible and run binaries natively on the RISC-V platform.

1.2. Environment

1.2.1. Selfie

The Selfie Project[2] is an educational software system that contains a self-compiling compiler `starc` that compiles a tiny subset of C, which produces assembly instructions conforming to a subset of the RISC-V instruction set[1]. Selfie also includes a self-executing emulator `rocstar` that is capable of executing all instructions emitted by `starc`.
Until now, Selfie used a minimal execution format that is called `RAW`. Selfie's emulator is capable of only loading binaries in this format. The reason for this is mainly simplicity. The `RAW` format holds a list of instructions and data without any additional information. Additionally, the machine state (special registers:

[1]Selfie uses the RV32I instruction set with the standard extension M for doing basic multiplication and division

stack pointer, global pointer) was initialized by instructions previously generated by the Selfie compiler. This was to minimize the state information that has to be assumed.
Selfie provides a tiny operating system interface between user code and the emulated hardware. This operating system interface allows 5 simple system calls: `write`, `read`, `open`, `malloc` and `exit`. The implementation of those system calls is similar to that of a standard Linux kernel, but not absolutely identical. For example, `malloc` is usually not implemented as system call.

1.2.2. RISC-V

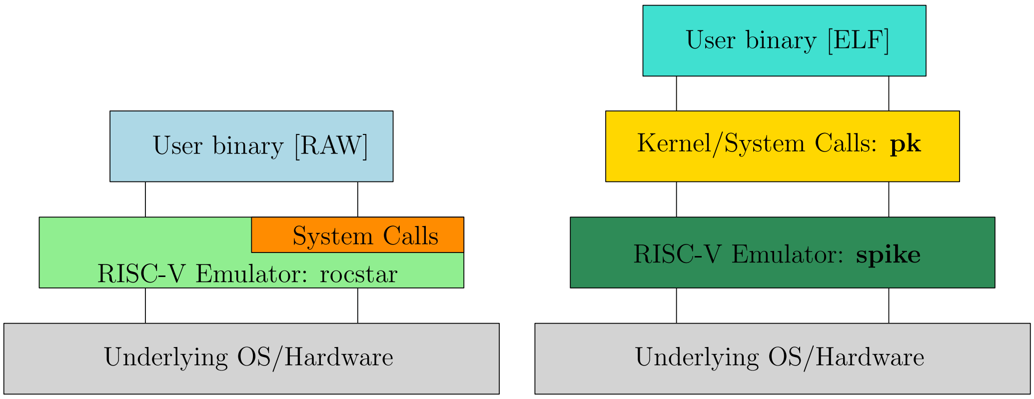
RISC-V is an open instruction set architecture based on the reduced instruction set computing principle. It was originally designed to support education and computer architecture research and aims to become an open standard for industry implementations[3].
The RISC-V project offers a repository with software related to the project[4]: First and most important, it provides the instruction set specification document that was used to port Selfie from MIPS to RISC-V[1, 5]. The RISC-V project also offers a reference implementation of that ISA, called `spike`. `spike` comes without an operating system, so to make user binaries run on top of this emulator, a kernel called `pk` (proxy kernel) is provided which serves as an interface between hardware and user applications. Moreover, the RISC-V project offers the *riscv-gnu-toolchain* as cross-compiler which also includes several GNU binary utilities, named binutils. The `gcc` cross-compiler produces binaries according to the Executable and Linkage Format, ELF[6].

1.2.3. Runtime Stack

Figure 1.1a and 1.1b show the current runtime stacks for both projects. Binaries are either generated by the `gcc` cross-compiler for the RISC-V stack or by `starc` for the Selfie runtime stack. Until now, the execution format differed and therefore, binaries were not interchangeable despite implementing the same ISA specification.

1.3. What Needs to be Done?

The main task of this work is to add the standardized ELF binary header, described in [7], to the code generation facility within Selfie. This boils down to adding the ELF header, Program Header Table and Segments that contain executable code or data. Figure 1.2a and 1.2b illustrate the differences between the two binary formats.
Till now, the memory address layout of Selfie was pretty simple: It started with address $0x00000000$ which usually contains the first instruction. This space is



(a) Selfie's runtime stack: The system call interface is integrated into the emulator

(b) RISC-V runtime stack with ELF binaries running on top of the Proxy Kernel `pk` and `spike`

Figure 1.1.: The runtime stack for the Selfie and the RISC-V project

Register	Name	Purpose
Global Pointer	`GP`	used in addressing global variables and strings
Heap Pointer		Dynamically allocated memory, not exposed to the user application as register value
Stack Pointer	`SP`	used in stack manipulation, function calls and local variables
Frame Pointer	`FP`	temporary pointer for providing a function frame on the stack
Program Counter		pointer to the next instruction that will be executed

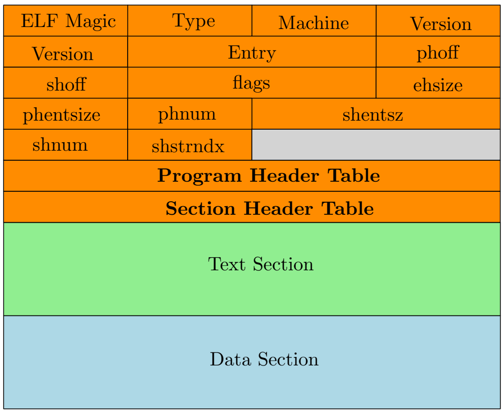
Table 1.1.: Most important state information

reserved for ELF binaries and a relocation is necessary. Due to this fact, every absolute jump or branch needs to be inspected and rewritten into a relative jump or branch. Addressing data and strings should not be affected because addressing is done relative to the Global Pointer.

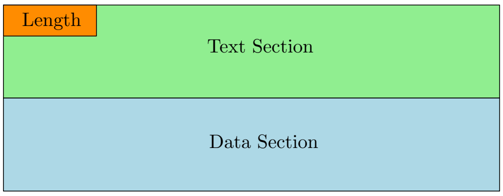
Furthermore, it is a requisite to adapt the initialization phase: Previously, Selfie versions tried to reduce the assumptions of the machine state as much as possible. Therefore, the Stack Pointer, Heap Pointer and the Global Pointer have been initialized by machine instructions that were generated as part of a so-called *program prologue*. Since the RISC-V toolchain follows ABI conventions used in the Linux operating system, the initialization phase must be adapted to meet the requirements of the RISC-V toolchain and ideally, only as few changes as possible need to be made to the emulator `rocstar`. Table 1.1 provides information about the most important state information.

Thirdly, it is necessary to verify that the system call interface works as expected.

The impact to `open`, `read`, `write` and `exit` calls should be minimal but a way needs to be found to deal with the imaginary `malloc` system call since memory allocation is done by either `mmap` or `brk` system calls on Linux systems.
Finally, it must be ensured that the convention, how to push the initial parameters of `main` on the stack, is identical for RISC-V and Selfie[8]. `argv` is pushed onto the stack as a vector of strings and the way `rocstar` does this must be identical.



(a) Generated execution binary with ELF header



(b) `RAW` binary format, similar to `.COM` binary format. A length field indicates the binary length.

Figure 1.2.: Differences of the binary format

1.4. Why ELF?

The remaining question before getting into the implementation details is if there are any other opportunities and why exactly ELF? There are a number of different binary formats out there. `a.out` is the classical Unix object format and its structure is very simple: It contains `.text`, `.data` and `.bss` segments for code and data. Furthermore, it contains a symbol- and a string table[9, 10].
The Common Object File Format COFF allows the definition of multiple `.text` and `.data` segments. It was introduced by SVR4 and is still used by Windows systems[11, 12]. The ELF binary format is now widely used by Linux and BSD Unix platforms. It is designed to support a highly flexible way of statically and dynamically linking various object files. Since it is not bound to any particular processor architecture, a lot of platforms use it nowadays. Examples are Solaris, Linux, BSD Unix, Microsoft Windows 10, Android mobile phones and a lot more[13, 14]. Since the RISC-V project only supports ELF binary format, the decision was simple.

2. Implementation

2.1. ELF

2.1.1. Format Overview

The ELF binary format is a standardized file format for relocatable, executable or shared object files[7]. Table 2.1 provides additional information about those different file types. This work will focus on the executable format to make binaries directly runnable. The overall file organization is illustrated in Figure 2.1.

relocatable	file holds code/data with unresolved symbols
executable	file is ready to execute (system can create a process image)
shared object	useful for statically and dynamically linking in two or more contexts.

Table 2.1.: ELF Object files[15, 16]

The ELF binary header is generated by the `starc` C* Compiler. Since the fundamental design goal is simplicity, the compiler produces only one code and data segment. The ELF header and the additional Program and Section headers are

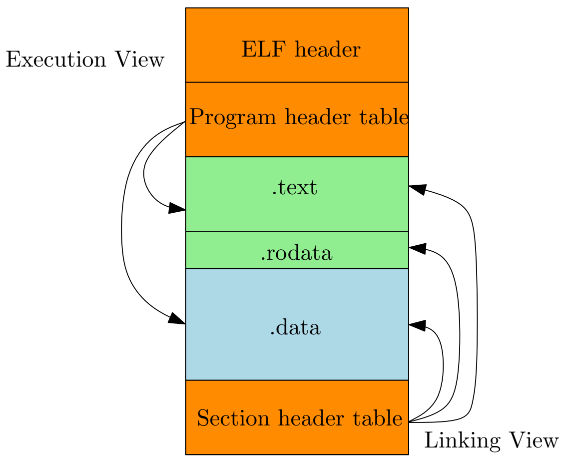


Figure 2.1.: ELF binary format, conceptional view

written to the output file after the compiler has parsed the source code and emitted assembly instructions. This is done because the header needs information about the length of code and data and for a single pass compiler, this is only possible after having processed the whole input file. The complete ELF header is created inside the `createELFHeader` procedure, stored in a global buffer called `ELF_header` and written to disk before code and data sections follow. Listing 2.1 shows the main functionality added to create a binary with an ELF header at the top.

```
  void selfie_output() {
    ...
    createELFHeader();
    write(fd, ELF_header, ELF_HEADER_LEN);
    // code to write code and data sections
}
```

Listing 2.1: creating and writing the ELF header information

The ELF format also describes some data types and their length in bytes which are listed in Table 2.2. The ELF specification [7] notes that no bit fields are used to make ELF binaries portable to various machine architectures.

Name	Size in Bytes	Name	Size in Bytes
`ELF32_Addr`	4	`ELF32_Half`	2
`ELF32_Off`	4	`ELF32_Sword`	4
`ELF32_Word`	4	`unsigned char`	1

Table 2.2.: Data types and sizes of ELF32

2.1.2. Header

The ELF header provides a way to identify a binary as ELF binary with a magic number and provides a roadmap of the object file. Basically, it holds information where to find the program header, section header table and how many entries there are. Both tables will be explained further in Sections 2.1.3 and 2.1.4. Listing 2.3 shows the general ELF32 header format.

```
  typedef struct {
    unsigned char   e_ident[16];    // 16 Byte ELF Identification
    Elf32_Half      e_type;         // Object file type
    Elf32_Half      e_machine;      // Machine architecture
    Elf32_Word      e_version;      // Object file version
    Elf32_Addr      e_entry;        // address, first instruction
    Elf32_Off       e_phoff;        // Offset to PHDR
    Elf32_Off       e_shoff;        // Offset to SHDR
    Elf32_Word      e_flags;        // processor-specific flags
    Elf32_Half      e_ehsize;       // ELF header size
    Elf32_Half      e_phentsize;    // PHDR entry size
    Elf32_Half      e_phnum;        // PHDR entry number
```

```
    Elf32_Half      e_shentsize;    // SHDR entry size
    Elf32_Half      e_shnum;        // SHDR entry number
    Elf32_Half      e_shstrndx;     // Index to string section
} Elf32_Ehdr;
```

Listing 2.2: ELF32 header as standardized in [7]

Most of those fields are handled as constants within Selfie. The first 4 bytes of `e_ident` hold the ELF magic number: `0x7f454c46` or interpreted as ASCII characters:

`0x7f 'E' 'L' 'F'`.[1]

The next four bytes encode the ELF class, either a 32-bit or 64-bit binary and declare the two's complement encoding and the byte order which is either MSB or LSB. The last 8 Bytes in `e_indent` declare the version (which is always 0 at the moment) and a padding which is reserved for future usage.
The next field `e_type` describe the type of the ELF object file. This is one of those listed in Table 2.1. Since we are only using the executable file format, the value is a constant within Selfie and its value is 2, meaning that the ELF binary is an executable where all symbols have been resolved.
The `e_machine` holds information about the underlying machine architecture and which ISA implementation is used for instructions. This value is a constant that declares the machine type as `RISC-V` with the value 243.[2] The `e_version` and `e_flags` are set to 1 and 0 respectively. Both of them are not used currently.
More important is `e_entry`: This field specifies the virtual address where the program starts executing. The previously used `RAW` format used address `0x00000000` as entry point. Most binary formats and virtual memory layouts specify the first page to be invalid to deny pointer access to `NULL`. Therefore, Selfie was modified to use `0x10000` as new entry point by adapting the code to be PC-relative. This adaption is discussed further in Section 2.1.4.
The fields `e_phoff` and `e_shoff` specify the location of the Program Header Table or Section Header Tables. Basically, they hold the location of the according tables and are used by applications to find and decode the correct table. The offset of those tables are statically calculated because the ELF header and the PHDR Table (which come before the SHDR Table) are always the same size at the moment. The `e_phentsize` and `e_shentsize` hold information about how long an entry in those tables is and `e_phnum` and `e_shnum` how many entries there are. Those

[1] The Magic number of the `a.out` binary format encodes a `branch` instruction on PDP-11 which is 0407 and jumps over the next view words, thus skipping the header and going directly to the text segment.[15]

[2] During the development of this project, the RISC-V project provided an adapted GNU Toolchain with all binutils. In January 2017, RISC-V got officially accepted in the upstream GNU Compiler Collection and future releases of the GCC Software Package will support ELF RISC-V binaries. The new number is `0xF3`.[17, 18]

numbers are constants in Selfie. `e_shstrndx` is a number that points to a segment which holds strings that are used in the ELF header.

```
void createELFHeader() {
  ...
  ELF_header = malloc(ELF_HEADER_LEN);

  *(ELF_header + 0) = 1179403647; // part 1 of ELF magic number
  *(ELF_header + 1) = 65793;      // part 2 of ELF magic number
  *(ELF_header + 2) = 0;          // part 3 of ELF magic number
  *(ELF_header + 3) = 0;          // part 4 of ELF magic number

  *(ELF_header + 4)  = 15925250; // Type and Machine fields
  *(ELF_header + 5)  = 1;        // Version number
  *(ELF_header + 6)  = ELF_ENTRY_POINT;
  *(ELF_header + 7)  = startOfProgHeaders;
  *(ELF_header + 8)  = startOfSecHeaders;
  *(ELF_header + 9)  = 0;        // Flags
  *(ELF_header + 10) = 2097204;  // Size of: ELF header, phdr
  *(ELF_header + 11) = 2621441;  // #Program header, size of shdr
  *(ELF_header + 12) = 196612;   // #Section headers, String tblidx
  ...
}
```

Listing 2.3: ELF32 header in Selfie. Values are encoded in decimal since `starc` is not capable of handling hexadecimal numbers as input. Since Selfie's granularity is 32-bit, two `ELF32_Half` fields are handled as a single number.

Listing A.4 shows the ELF Header spanning line one to line 20.

2.1.3. Section Header Table

This table holds information about sections which are present in the binary. Generally, the Section Header Table is not absolutely necessary since program loading is controlled by the Program Header Table. Nevertheless, this section is useful for reading an ELF binary with tools like `riscv32-unkown-linux-readelf` and `riscv32-unkown-linux-objdump`.

Sections are grouped into segments which may be loaded or ignored, depending on what should be done with an ELF binary. According to [7], every section in an object file has exactly one section header, section headers do not overlap, each section occupies a contiguous sequence of bytes within an ELF file and it is allowed to have inactive space (i. e. padding bytes), that are not described by a section header. Listing 2.4 shows the overall structure of an entry in the Section Header Table.

```
typedef struct {
    Elf32_Word    sh_name;
    Elf32_Word    sh_type;
    Elf32_Word    sh_flags;
    Elf32_Addr    sh_addr;
    Elf32_Off     sh_offset;
    Elf32_Word    sh_size;
    Elf32_Word    sh_link;
    Elf32_Word    sh_info;
    Elf32_Word    sh_addralign;
    Elf32_Word    sh_entsize;
} Elf32_Shdr;
```

Listing 2.4: ELF32 SHDR Table Entry as standardized in [7]

The `sh_name` is an index into the String Table which is described in Section 2.1.5 and is declared in the Section Header Table with number 3. The type field describes the content and semantics of a section. Some possible types are described in Table 2.3.

Name	Description
`NULL`	Mark section header inactive
`PROGBITS`	Content that is defined by the program, such as data or instructions
`HASH`	Symbol hash table, not used in Selfie
`STRTAB`	String table
`REL`	Relocation entries, not used in Selfie

Table 2.3.: Selected Section Header types[7, 15]

`sh_flags` section holds attribute flags that are used for this section. The `W` (Write) flag specifies that this section is writable, `X` specifies executable permission during process execution. `A` (Allocation) declares the segment to occupy storage and it is off for some segments that are not loaded during runtime.
`sh_addr`, `sh_offset` and `sh_size`, describe the address where the segment will be loaded while running, the offset where the segment may be found inside the ELF binary and the size of that section, respectively. The `sh_addralign` particularize constraints with respect to address alignment. The other entries are usually zero and very specific to some corner cases which are not necessary for our purposes.
Selfie uses three special sections named `.text`, `.data` and `.shstrtab`. Those sections are created by a procedure `createELFSectionHeader()` which is called inside `createELFHeader()`. Listing 2.5 shows Selfie's code to generate one ELF section. Selfie generates four sections which can be seen in Listing A.4 in line 22 until 27.

```
void createELFSectionHeader(int start, int name, int type,
```

```
                                  int flags, int addr, int off,
                                  int size, int link, int info,
                                  int align, int entsize)
{
  *(ELF_header + start)     = name;
  *(ELF_header + (start+1)) = type;
  *(ELF_header + (start+2)) = flags;
  *(ELF_header + (start+3)) = addr;
  *(ELF_header + (start+4)) = off;
  *(ELF_header + (start+5)) = size;
  *(ELF_header + (start+6)) = link;
  *(ELF_header + (start+7)) = info;
  *(ELF_header + (start+8)) = align;
  *(ELF_header + (start+9)) = entsize;
}
```

Listing 2.5: Function that creates an ELF Program Header Table entry

One particular useful thing of having those Section headers is the possibility to use `riscv32-unkown-linux-objdump`. This utility is very similar to Selfie's `-d` disassemble switch: it prints and disassembles the encoded instructions into a human readable format. A sample Listing is shown in A.2.

2.1.4. Program Header Table

The most important part in an ELF binary is to tell the underlying system loader how to load a binary into memory and where execution starts. This is the purpose of the PHDR Table. Since ELF binaries are designed to work as static or dynamic (shared) object files which are subject to multiple relocations, Program header entries hold further information about different segments inside the object file.
Since the goal was a very simple ELF binary, we decided to use only one entry in the Program Header that is loaded all in one go. Normally, various operating systems have different policies for different segments, like no write permission to pages which hold executable code or no executable permissions for the data section. Listing 2.6 shows the definition of an entry in the PHDR table.

```
  typedef struct {
    Elf32_Word    p_type;
    Elf32_Off     p_offset;
    Elf32_Addr    p_vaddr;
    Elf32_Addr    p_paddr;
    Elf32_Word    p_filesz;
    Elf32_Word    p_memsz;
    Elf32_Word    p_flags;
    Elf32_Word    p_align;
  } Elf32_Phdr;
```

Listing 2.6: ELF32 PHDR Table Entry as standardized in [7]

The `p_type` field describes how to interpret the content of this array. Selfie generates only the type `LOAD` which specifies a segment that is loaded into virtual memory. Other types specify dynamic linking information (`DYNAMIC`), interpreter information (`INTERP`) or auxiliary information (`NOTE`).
For a loadable segment, the field `p_filesz` specifies the size of this segment. Usually, this size is identical to `p_memsz`. An exception for this may be a `.bss` section which contains data whose value is 0 initially, but those values are not stored in the ELF binary[19].[3] The field `p_filesz` is calculated by the `binaryLength` variable. The entry `p_vaddr` specifies the virtual address, at which the first byte of the segment will be loaded. The `p_offset` is the position of the first byte in the ELF binary.[7, 15]
The Listing A.4 shows the program header in line 29 up to 31. The sample Listing shows that the program header is at location `0x110` in the file and will be loaded at `0x10000`. The file size is the length of the segment, which is the sum of text and data. Figure 2.2 illustrates how an ELF binary is structured and loaded into memory.
One thing that was already mentioned is that due to the relocation to the virtual address specified in the PHDR Table, all address references generated during compiletime must be *relative* to either a register or the PC. We also do not have a separate linking stage where such unresolved references would be fixed and therefore, Selfie needs to generate position independent code (PIC). All operations related to addresses have been carefully examined. This includes

- branches in `while` and `if` statements,
- pointer access for local variables,
- pointer access for global variables,
- storage allocation,
- procedure calls.

Since branches of `while` and `if` were already PC-relative, there was no need to change anything. Local variables are addressed relative to the FP which means that accesses to them are also position independent. Otherwise, nested function calls and recursion would not work. Global variables are addressed as offset of the GP and therefore, position independence is guaranteed. For storage allocation, it is also true that no relocation is necessary since `malloc` returns a pointer to a new, unused block of memory based on the Heap Pointer. The Heap Pointer is

[3] Some authors call it *better-safe-space*, since there is no allocation inside the ELF binary.[20]

initialized after loading the ELF binary into main memory. Furthermore, `malloc` is a runtime concept and the compiler does not need to take care of this.
The only place where a change was necessary was within function calls. The MIPS standard declares the `JAL` as PC-region branch[21] but actually, Selfie implemented it as absolute jump instruction. The adaptation was made during the transistion of Selfie from MIPS to RISC-V described by [1]: during a function call, a simple calculation produces a PC relative jump value which is either negative, if the function appeared above, or positive, if the function appears below the current call. The second case must be handled with a regular fixup[22].
The creation of PHDR Table entries is encapsulated into a function which is listed in Listing 2.7. This makes reusage simple and further table entries can be added. Most fields are currently statically initialized as constants. The function is called inside `createELFHeader()`.

```
void createELFProgramHeader(int type, int offset, int vaddr,
                            int paddr, int fsize, int memsize,
                            int flags, int align)
{
  *(ELF_header + 13) = type;    // Type of program header, LOAD
  *(ELF_header + 14) = offset;  // Offset to 1. byte of segment
  *(ELF_header + 15) = vaddr;   // Virtual address
  *(ELF_header + 16) = paddr;   // Physical address
  *(ELF_header + 17) = fsize;   // File size
  *(ELF_header + 18) = memsize; // Memory size
  *(ELF_header + 19) = flags;   // Flags (Read, Write, Execute)
  *(ELF_header + 20) = align;   // Alignment of segments
}
```

Listing 2.7: Function that creates an ELF Program Header Table entry

Figure 2.2 show the ELF file organization that is produced by Selfie and how the Program Header and Section Header Table are organized. Additionally, the Figure shows how such an ELF binary is loaded into the virtual memory by the `pk` kernel[23].

2.1.5. Strings

The `.shstrtab` Section holds `NULL` terminated Strings. Those are used within the ELF header for constants like `.text` or `.data` strings. Additionally, ELF uses that section to maintain and build relocation information, hold function names or debugging information. Within Selfie, the String section is only used to store ELF related strings. Strings used inside C files are stored in the data section. However, it is possible to extend this scheme and store function names or even variables and their corresponding location.
Listing 2.8 shows a hexadecimal representation of this section. Figure 2.3 shows the corresponding section and how it is organized and stored.

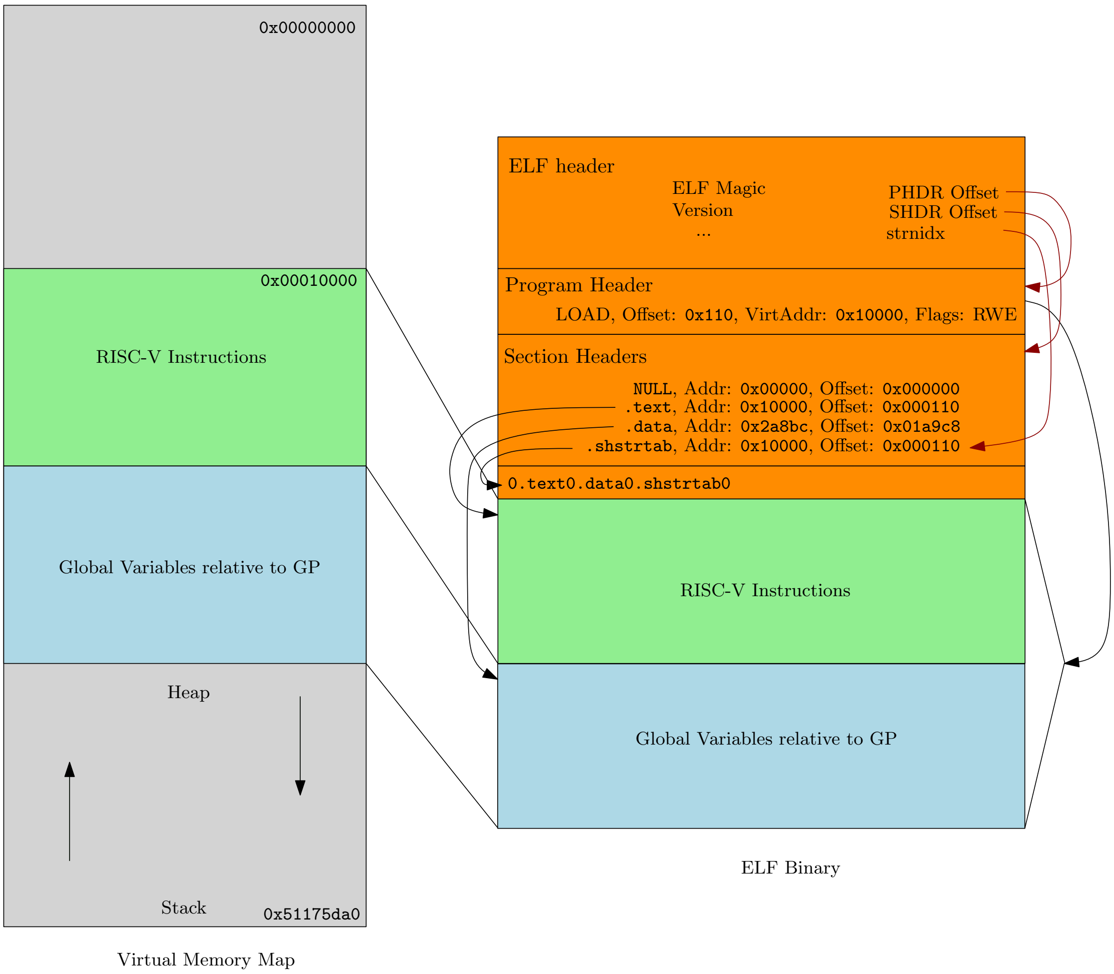


Figure 2.2.: ELF binary format generated by Selfie and mapped into address space. Please note that the picture shows low addresses at the top.

```
000000f0  00000000002e7465  7874002e64617461   |......text..data|
00000100  002e736873747274  61620000d4020000   |..shstrtab......|
```

Listing 2.8: `hexdump(1)` of the String section created by Selfie

2.2. System Call Interface

As already mentioned, Selfie's emulator `rocstar` contains a tiny operating system interface and provides the most important system calls to user applications. The calls have been carefully selected so that the system supports self-referentiality. All used calls are explained in Table 2.4. `rocstar` implements the calling convention described by System V/ABI32[24] and in [5, p107].
The operating system calls `open`, `read`, `write` and `exit` are semantically identical

\0	.	t	e	x	t	\0	.
d	a	t	a	\0	.	s	t

Figure 2.3.: Strings stored by Selfie

Name	Description
`malloc`	allocate storage and return pointer to the newly allocated storage
`open`	open a file and return an file descriptor
`read`	read a specified number of bytes from a given file descriptor
`write`	write a specified number of bytes to a given file descriptor
`exit`	terminate a program and send an exit status

Table 2.4.: System calls provided by `rocstar`

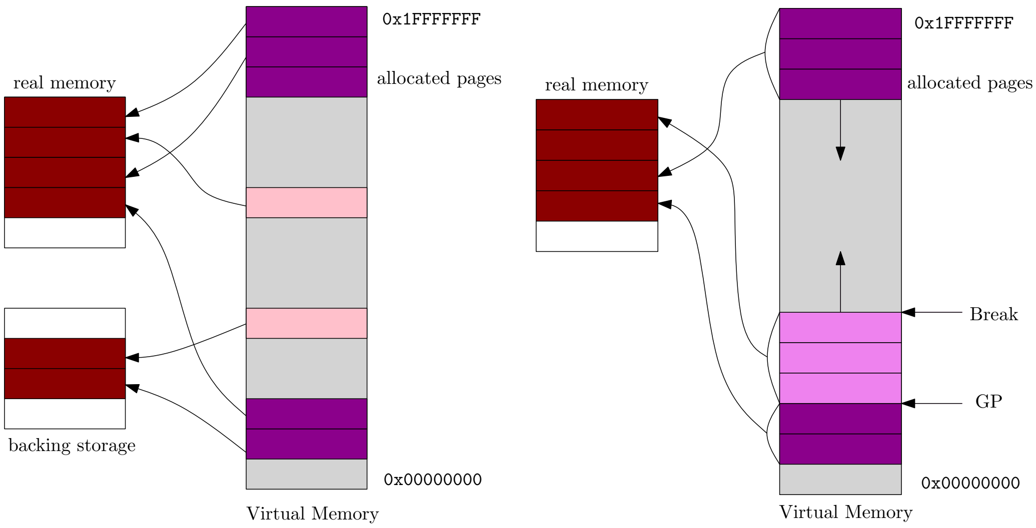
for `pk` kernel and Selfie's emulator and after updating the system call number in Selfie, those calls worked without any problems. The `malloc` system call in Selfie is not available on `pk` since it is not a Unix system call. Therefore, some adaption was necessary.

Before explaining possibilities to solve this problem, I want to take a look into the details about how a general purpose operating system allocates storage. The `malloc` system call in Selfie was created since dynamically allocated storage within the compiler is a very important concept and is used a lot. Without dynamically allocated memory, storage allocation would have been much more complicated because Selfie contains a lot of global state information that is constantly updated, expanded and changed. On the other hand, Selfie should be easy and understandable even for students without a lot of prior knowledge and therefore, the `malloc` interface was chosen because it is a well known procedure.

Usually, a user application requests memory by calling `malloc` with the number of bytes to allocate. `malloc` is a function within the `libc` library and its purpose is to manage dynamically allocated memory with the underlying system calls. It also provides the `free` interface to release previously allocated memory but due to the fact that implementing a `free` procedure makes Selfie's code more complicated, this mechanism was not implemented.

On BSD operating systems, the `malloc` call requests memory usually by `mmap` system call[25, 26]. This call is used to map memory into the address space of a process. The underlying system allocates pages and maps those pages to physical memory. For compatibility, most systems provide `brk` and `sbrk` system calls[27]. Those system calls move the so-called *break* pointer so that a user process can allocate and use more memory. Since most operating systems have a sophisticated virtual memory system based on paging, the `brk` and `sbrk` calls are obsolete by now[27]. Figure 2.4a and 2.4b visualize the differences of how `mmap` and `brk` work.

All developed solutions should be simple to understand and minimize the changes



(a) Memory Management wit `mmap` provides a highly flexible way for storage allocation.

(b) `brk` and `sbrk` are the conventional allocation calls of Unix systems, initially introduced in Unix v6.

Figure 2.4.: With hardware being available to support and utilize Virtual Memory, the conventional calls have been replaced in 4.4BSD.

to the codebase. The self-referentiality of Selfie should be possible on top of the X86 and RISC-V processor architectures. Additionally, it should work on macOS and GNU/Linux with LLVM/clang and the GNU Compiler Collection before. Figure 2.5 visualizes the possibilities of generating a Selfie binary.
Three different possibilities have been worked out, each having some advantages and disadvantages.

2.2.1. Using the `mmap` interface for storage allocation

The first idea is to replace the `malloc` system call with `mmap`. Listing 2.9 shows the function definition found in BSD Unix operating systems. To make the transition not overly complex, a simple function will be written within Selfie that works as allocator and makes use of `mmap` in the background.

```
void *
mmap(void *addr, size len, int prot, int flag, int fd, int off)
```

Listing 2.9: Definition of `mmap`. `addr` specifies a location in the address space, length is used to map a specific amount of memory, most often one page i. e. 4096 Bytes. `prot` and `flags` make pages sharable and declare a basic protection mechanism. `fd` and `off` may be used in combination with files (mounting a file-backed element into an address space)

The advantage of this approach is that it resembles real world operating systems by using a modern interface for storage allocation. It opens a door for new possibilities

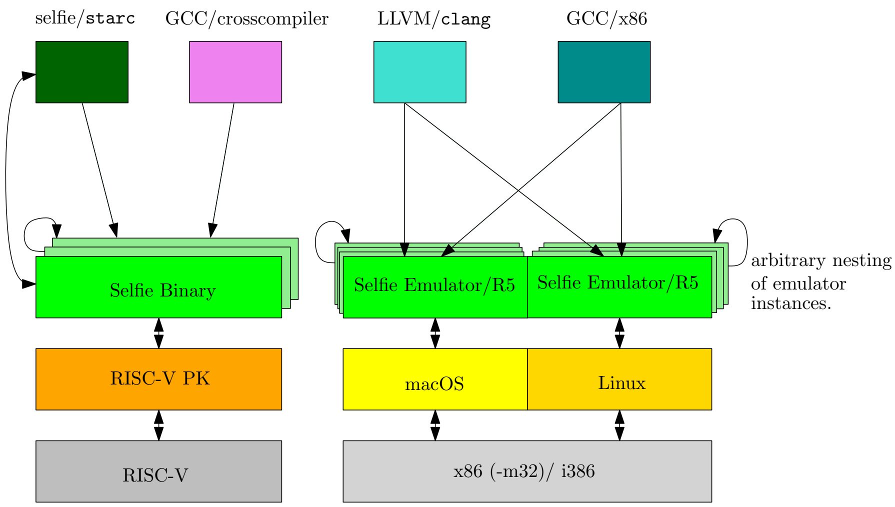


Figure 2.5.: How and where Selfie binaries are generated and run. Selfie's compiler `starc` can be viewed as cross-compiler if running on the X86 computer platform and creates binaries for RISC-V processors or the `spike/rocstar` emulator.

like implementing an educational storge allocator and adding the `free` interface. The disadvantage is that a way needs to be found how to expose the `malloc` function to user programs: either a `malloc` function has to be emitted like a conventional system call, and `malloc` needs to be written in assembly instructions or every user binary needs to be linked with a library that provides the `malloc` function. Furthermore, as can be seen in Listing 2.9, the `mmap` interface is much more complicated than the conventional `malloc` functions and a lot of functionality is currently not used in Selfie.
After exploring this approach, another problem turned up: The version of the GNU toolchain that is provided to us failed to compile programs that use the `mmap` call. Listing 2.10 shows that it is not possible to compile Selfie with the cross-compiler while using `mmap`. After examining this problem further, it turned out that `libc` from the GNU toolchain does not provide the `mmap` system call. This makes binaries not cross-compileable without further rewriting tricks.

```
selfie.c:258:2: warning: implicit declaration of function mmap
  mmap(0, 1024, 0, 0, -1, 0) ;
  ^
/tmp/ccOV9eSk.o: In function ‘main’:
test.c:(.text+0x3c): undefined reference to ‘mmap’
collect2: error: ld returned 1 exit status
compilation terminated.
```

Listing 2.10: Using `mmap` for storage allocation

2.2.2. Using the `brk/sbrk` interface

Since using `mmap`, described in Section 2.2.1, did not lead to a successful outcome, a new approach based on using either `brk` or `sbrk` was developed. As was said previously, the `brk` and `sbrk` interfaces are considered as being obsolete. However most operating systems provide an interface for those system calls and it may be suitable to use those calls.
An advantage is that `brk` and `sbrk` are easier to understand compared to the more abstract `mmap` call. The disadvantages are similar to using the `mmap` call: The compiler `starc` needs to provide a library function with `malloc` functionality somehow as library function since most programs are using it. This is illustrated in Figure 2.6. Internally, Selfie can be constructed to use up to two `brk` calls inside a self written `malloc` function. Since `brk` is considered as "historical curiosity"[27], an additional drawback is that the implemented version is not up to date what users might expect.

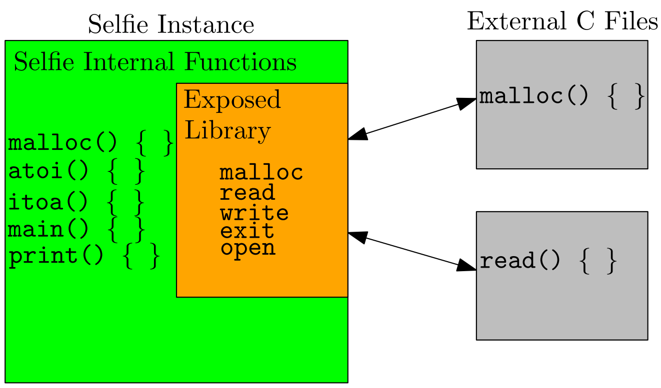


Figure 2.6.: Visualization of Selfie's library

Listing 2.11 shows the definition of those two functions and Table 2.5 holds information about the difference between those two calls as described on a Linux operating system. It is interesting to point out that the `sbrk` function is very similar to `malloc` with respect to the programmer's API. Both functions will be called with a size in bytes that indicates how much memory to allocate and return a pointer to the newly allocated space. However, those two functions are very different internally: The `malloc` function makes sure to allocate the given amount of bytes and return a pointer to it. Eventually, it makes a system call to allocate more storage with `mmap` or, on older systems, with `brk`. The `sbrk` system call directly asks the operating systems to acquire more memory, it rounds up to the page size and returns a pointer to it. On Linux, `sbrk` is implemented as macro based on `brk` with some internal bookkeeping.[28]

```
int
brk(void *addr);

```

```
void *
sbrk(int incr);
```

Listing 2.11: Definition of `brk` and `sbrk`.

`brk`	`sbrk`
`brk()` sets the end of the data segment to the value specified by `addr`, if that value is reasonable, the system has enough memory and the process does not exceed its maximum data size. `brk` returns 0 on success, and −1 on failure. `errno` will be set to `ENOMEM`.	`sbrk()` increments the programs data space by `incr` bytes. Calling `sbrk()` with `incr` of 0 can be used to find the current location of the program's break point. On success, `sbrk` returns a pointer to the previous program break. On error, `sbrk` returns −1 and `errno` will be set to `ENOMEM`.

Table 2.5.: Comparison between `brk` and `sbrk`.[27]

Implementing `brk` was not possible due to the same problem as previously encountered with `mmap`: Programs that use `brk` were not possible to cross-compile and the linking phase failed. Listing 2.12 shows a detailed error message.

```
# /root/toolchain/bin/riscv32-unknown-elf-gcc brk_test.c
brk_test.c: In function 'main':
brk_test.c:3:6: warning: implicit declaration of function 'brk'
  a = brk(0xFFFFF);
      ^

/tmp/ccUwtKON.o: In function `main':
brk_test.c:(.text+0x18): undefined reference to `brk'
collect2: error: ld returned 1 exit status
```

Listing 2.12: Linking to `brk` failed

The second attempt was to use `sbrk`. Since the functionality of `brk` can be replicated with `sbrk`, it is not much of a problem to do so. After verifying that the provided GNU toolchain actually implements `sbrk` so it is usable, a version of Selfie was created that makes use of `sbrk`.
However during implementation and transition to `sbrk`, it turned out that the RISC-V `pk` kernel maps the `sbrk` system call to `brk`[29]. Interestingly, running Selfie on top of Linux and macOS results results in use of the the correct `sbrk` call, each with slightly different behavior. Figure 2.7 illustrates different platforms and the semantics of `sbrk`. Using `sbrk` with actual `sbrk` semantics (instead of the way `pk` implements it) resulted in code generation that was not executable on the RISC-V `pk` kernel. This is due to the fact that calling `sbrk` with small numbers resulted in setting the `break` pointer to a very small memory address which leads to a segmentation fault. As example, calling `sbrk` with 4096 Bytes as parameter results in setting the `break` pointer on the RISC-V platform to the memory address

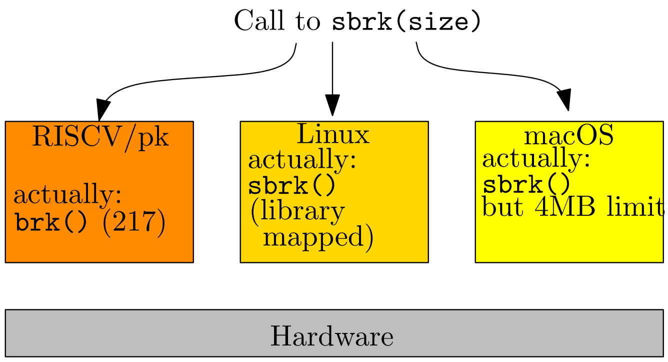


Figure 2.7.: `sbrk()` system call mapping on different operating systems. On `pk`, `sbrk` is actually mapped to `brk` semantics. On Linux, `sbrk` is only a library function/macro within the `glibc`. On macOS, `sbrk` semantic is implemented as a fixed size 4 MB array allocated with `malloc`

$(0x1000)_{16} = (4096)_{10}$ and further read and write accesses overwrite parts of the text segments or access memory that is not mapped. Figure 2.9a demonstrates what happens to the `break` pointer.

Using `sbrk` with `brk` semantics resulted in different behavior: Since the mapping (`sbrk` → `brk`) in RISC-V is done that way, it worked flawlessly on this platform. It also worked on Linux platforms but during testing, some side effects were discovered. The parameter to `brk` is a pointer which represents most likely a very high number, the memory consumption of Selfie exploded. Figure 2.8 shows memory usage of `htop(1)`, a process usage monitor[30].

Selfie is using more than 3GB memory but the resident set (RES) is only 3.5MB. Because the virtual memory system is based on demand paging, Selfie runs on computers that do not have as much memory but it is clearly a sign that this approach is also not optimal.

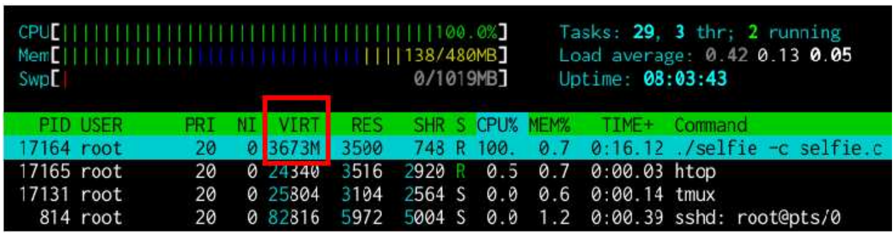
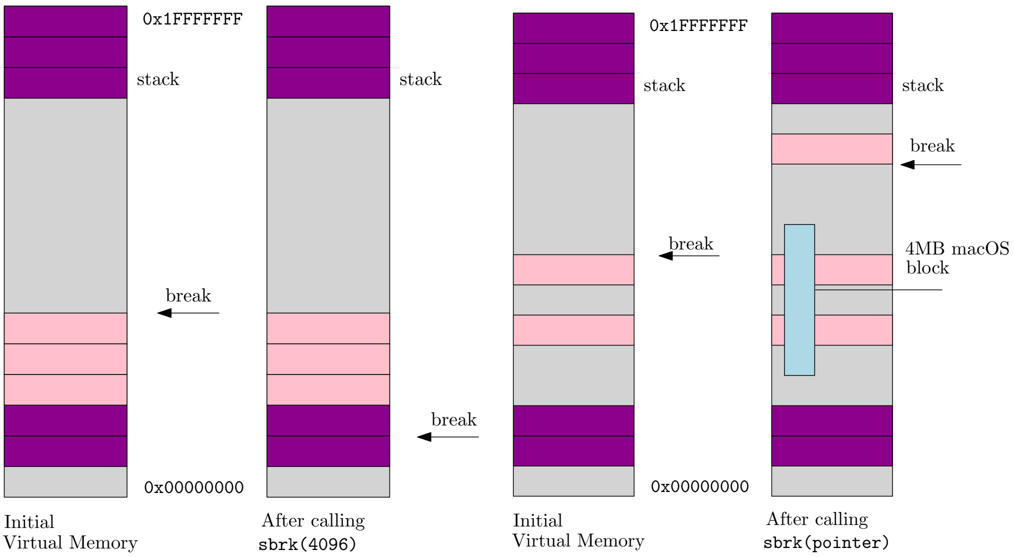


Figure 2.8.: Running Selfie: memory usage with `htop`

Despite that, self-compilation and emulation works on Linux systems. The situation is different on macOS operating systems: The `sbrk` system call is only implemented for compatibility reasons and it is strongly advised not to use this system facility. Internally, the `sbrk` system call uses a fixed size array of 4MB that is allocated with `malloc`. Since Selfie needs more memory than that, self-compilation and emulation failed on this system[31]. Figure 2.9b illustrates the situation on Linux and macOS systems.



(a) Using `sbrk` with the correct semantics results in setting the `break` pointer wrong on the RISC-V platform, since it implements `brk` semantics. Both arrays show the memory organization of RISC-V (before and after the system call)

(b) Using `sbrk` with `brk` semantics results in wrong behavior on Linux and macOS. On macOS, the limit of memory allocation with `sbrk` is 4MB. Both arrays show the memory allocation on Linux/macOS (before and after the system call)

Figure 2.9.: Visualization of memory organization with different semantics used at the `sbrk` system call.

A workaround was developed that is based on a dynamic library that is preloaded and intercepts every `sbrk` call. With that in place, Selfie was able to self-compile and emulate even on macOS systems. Figure 2.10 shows the interception. The code basically does a proper `malloc` call on the host system. The library is available at [32].

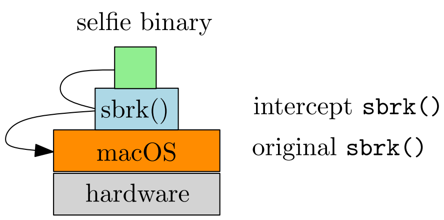


Figure 2.10.: `sbrk` interception on macOS

2.2.3. Kernel Patch for `pk`

Since the `sbrk` approach was quite complicated and involved abusing the virtual memory system and made the compilation of Selfie more complicated on macOS, a better solution was developed.
The new attempt is based on changing the RISC-V `pk` kernel and adding functionality to allocate storage. The advantage is that changes to Selfie and memory allocation are minimized. Adding a system call to RISC-V seems to be simpler in comparison with the approaches examined in Section 2.2.1 and Section 2.2.2.
The disadvantage is that running Selfie-generated binaries need a patched version of the RISC-V `pk` kernel. One way to simplify that is outlined in Section 3.3.
The first step was to reverse all changes and use the ordinary `malloc` interface within Selfie as done before. After that, a verification step consisting of self-compilation and emulation of Selfie on all platforms except on `spike` and `pk` was carried out. The library that is provided by Selfie to be used by user programs which will be compiled by `starc` emits a system call code stub with the system call number 213 (as done before).
This call will be handled inside `rocstar` as regular `malloc` system call. On RISC-V `pk`, an additional system call was added that implements the `sbrk` functionality with system call number 213. Therefore, a call to malloc by a user program ends up as being executed as `sbrk` system call by the `pk` kernel. It is not even necessary to name this call `sbrk` inside the `pk` kernel, since it is referenced by a system call number that is generated inside `starc`.

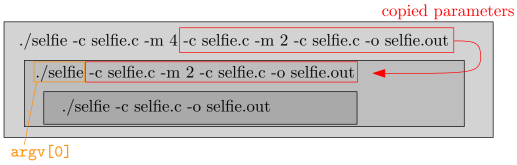
2.3. Initial Stack Organization

Another important topic to make Selfie compatible with the toolchain of RISC-V is the initial stack organization: Selfie heavily depends on the arguments that are passed on the command line. The usage of Selfie is listed in Listing 2.13. The `main` function has two parameters: an integer value in `argc` that specifies the number of strings which can be addressed with the `argv` vector. The usual declaration of `main` is described in Listing 2.14.

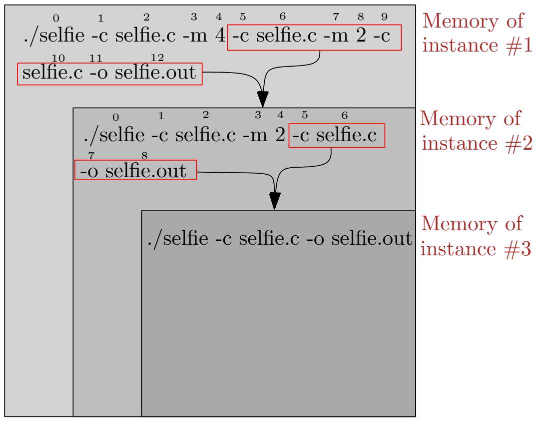
```
selfie { -c { source } | -o binary | -s assembly | -l binary }
                       [ (-m | -d | -y | -min | -mob ) size ... ]
```

Listing 2.13: Command line arguments

The emulator needs to be modified so that the outer emulator copies some parts of the given parameters on top of the stack for the application that runs inside the emulator. Usually, memory is allocated for the inner program and the parameters need to be copied onto the runtime stack of that program inside the emulator. This is especially important when cascading multiple instances of the emulator and compiler. Figure 2.11a shows two emulator instances and how the parameters are copied. The program name is always located at the first position in `argv`.



(a) Multiple emulator/compiler instances: the outer emulator must copy parts of the arguments to the runtime stack of the inner program.



(b) Different Memory Areas

Figure 2.11.: Conceptional View and concrete memory view of runtime stack.

Additionally, the RISC-V emulator/`pk` kernel and the Selfie emulator `rocstar` must be synchronized so that both emulators handle those arguments the same way.

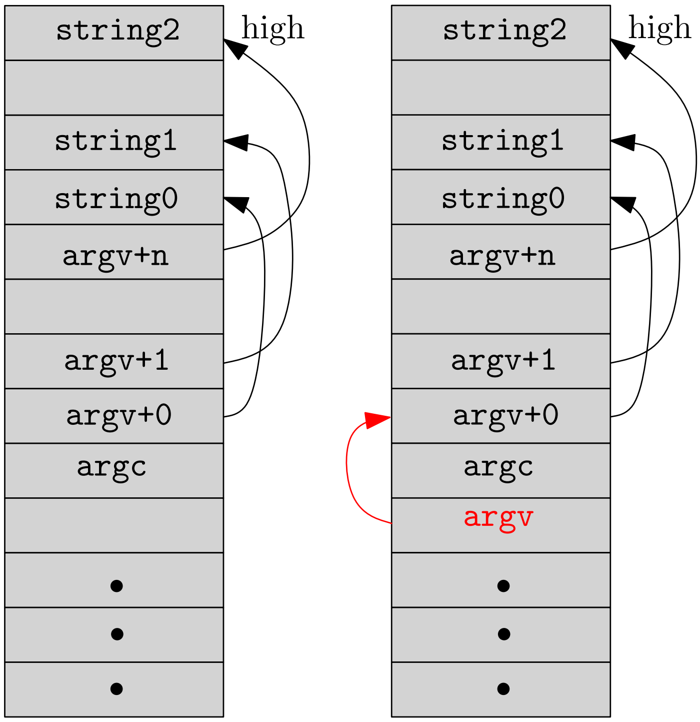
```
int main(int argc, char **argv) {
  ...
}
```

Listing 2.14: Commandline arguments

Since there is no documentation available, the way how RISC-V's `pk` kernel pushes arguments to the top of the stack had to be reverse engineered by manually stepping through parts of the stack. The only difference is that RISC-V did not push a pointer to the `argv` vector. Figure 2.12a and Figure 2.12b show the differences. Selfie's compiler output was adapted to manually push the pointer to `argv` onto the stack in [33].

2.4. Additional Changes

The main tasks were discussed in Sections 2.2 and 2.3. This section describes a few minor issues during the transition.
One thing that was necessary to be adapted was the initialization of the GP: Previously, the GP has been initialized by code produced by the compiler[34]. The RISC-V `pk` kernel, however, sets the value of the GP itself (due to information found in the ELF binary header). Therefore, the emulator and compiler had been changed in commit [35] to not initialize the GP. In the beginning, the compiler was extended to support two output formats: either the historical `RAW` format or the newer ELF binary format. The desired output format could be selected through command line arguments (`-c` for a binary in RAW format, `-C` for one in ELF



(a) Initial stack generated by `pk`.

(b) Initial stack used by `rocstar`

Figure 2.12.: Different initial runtime stack.

format). This approach allowed for testing the execution of ELF binaries on the RISC-V `pk` kernel without losing the ability to run `RAW` binaries on `rocstar`. Later, `rocstar` was adapted to load ELF binaries with `-M`. It is important to note that the ELF handling code within `rocstar` is not able to load arbitrary ELF binaries but only the specific ones produced by `starc`. The compiler distinguished the two output binary formats with an internal flag until the `RAW` format was removed entirely. The main reason for that is to reduce the code complexity and one binary format is certainly good enough for most scenarios[36].

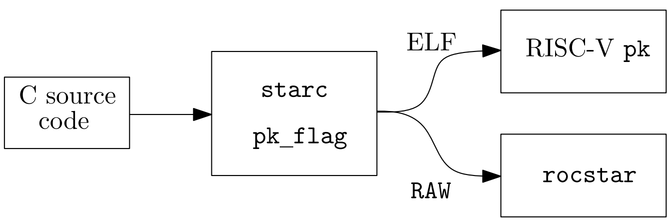


Figure 2.13.: Output generation for `RAW` and ELF

Another minor issue that manifested during the transition involved a wrongly encoded instruction, namely `sub`. This happened because the RISC-V manual was unclear about which register should hold which value for a subtraction, so they were switched. This mistake was easily discovered by looking at the output of `objdump` and fixed in [37]. The relevant part is shown in 2.15.

```
10178:     ff842303            lw      t1,-8(s0)
1017c:     3e6482b3            0x3e6482b3
10180:     fe542e23            sw      t0,-4(s0)
```

Listing 2.15: wrongly encoded instruction

3. Results

This section is a summary of the implementation details described in Section 2 and shows some runtime measures and performance tests.

3.1. What is Working

The compiler `starc` is able to produce a binary with a valid ELF header that can be used by RISC-V `pk`, `objdump` and `readelf`. It is largely based on constants and some dynamically generated numbers that are written to disk after the compilation is done. With the provided patch described in Section 2.2.3, all system calls work as expected on the RISC-V `pk` kernel.

`rocstar` is able to load the ELF binary and the initialization phase was adapted to mimic the behavior of RISC-V's `spike` emulator with respect to register initialization. The initial stack organization was also adapted so that binaries generated with an ELF header run on `rocstar`. Table 3.1 shows which constellations of compiler and emulator are working.

	rocstar	spike
gcc		X
starc	X	X

Table 3.1.: Runtime support

Self-compilation is now working on RISC-V `pk` and `rocstar`. Binaries generated by Selfie can run directly on RISC-V `pk`. An interesting thing to note is that Selfie can cross-compile itself on a standard X86 architecture and the resulting binary can be self-compiled on top of `spike` and `pk`. Listing 3.1 shows that process.

```
1 # selfie -c selfie.c -o selfie.r5
2 # spike --isa=RV32IMAFDC pk ./selfie.r5 -c selfie.c -o s.r5
3 # diff s.r5 selfie.r5
```

Listing 3.1: Line one shows self-compilation on X86, Line 2 shows the self-compilation with that binary on top of `spike` and `pk`.

3.2. What is not Working

The compiler `starc` produces the same ELF header for every input source except for a few numbers. A more dynamical approach may be interesting because the ELF PHDR and SHDR generation facility inside Selfie can handle the generation of multiple sections and program headers.
Another drawback is that currently, not all strings are located in the strings section which was described in Section 2.1.5. To do this properly, the entire string handling inside the kernel would have to be changed. However, the benefit of doing this would not outweigh the required work. Function names are not included in the ELF output binary and therefore, `objdump` prints a long list of assembly instructions without structural information like where a certain functions starts and ends.
Additionally, debugging symbols according to DWARF format[38] are not generated.
The emulator `rocstar` is also not capable of loading arbitrary ELF binaries. Since `rocstar` is limited to handling a tiny subset of the RISC-V instruction set, it cannot load ELF binaries that are generated by another compiler. Most compilers generate a wide range of instructions and those would immediately fail on `rocstar`.
Furthermore, all binaries generated by `starc` which use `malloc` to allocate memory need the patch described in Section 2.2.3. Without it, program execution will fail due to a missing memory allocation function. Programs that do not use any memory allocation facility work without such an adaptation.

3.3. Standardized VM Environment

As was stated earlier, RISC-V provides a very large toolset for 32- and 64-Bit RISC computers. To simplify testing and usage, a virtual machine has been created that is preloaded and already configured to compile and run Selfie binaries with the RISC-V toolchain. In particular, the RISC-V toolchain is installed as 32-bit and 64-bit version, patched and compiled. The virtual machine is based on the Open Virtualization Format that works on a wide number of virtualization platforms[39].
The runtime measures of section 3.4 have been carried out on a virtual machine that can be downloaded at [40]. The underlying hardware configuration is described in Table 3.2.

CPU	Intel(R) Core(TM) i5 CPU M520@2.40GHz
Memory	7801MiB system memory
Architecture	x86 64 bit, smbios-2.6
Disk	Intel SSDSC2CW12, 120 GB SSD
OS	Debian GNU/Linux 7.8[41]
Virtualization	VMware(Inc) Workstation 11[42]

Table 3.2.: Hardware Environment

3.4. Runtime Performance: `spike/rocstar`

The first comparison has been carried out on a simple program that calculates fibonacci numbers[43] , defined as:

$$f_k = f_{k-1} + f_{k-2}.$$

The code is listed in appendix A.1. The approach is done in an inefficient way by recursion with a runtime complexity of $\mathcal{O}(2^n)$. Figure 3.1 shows the runtime of the fibonacci program compiled with either `gcc` or `starc` running on top of both platforms. The parameter k is adapted and the runtime is measured in seconds.

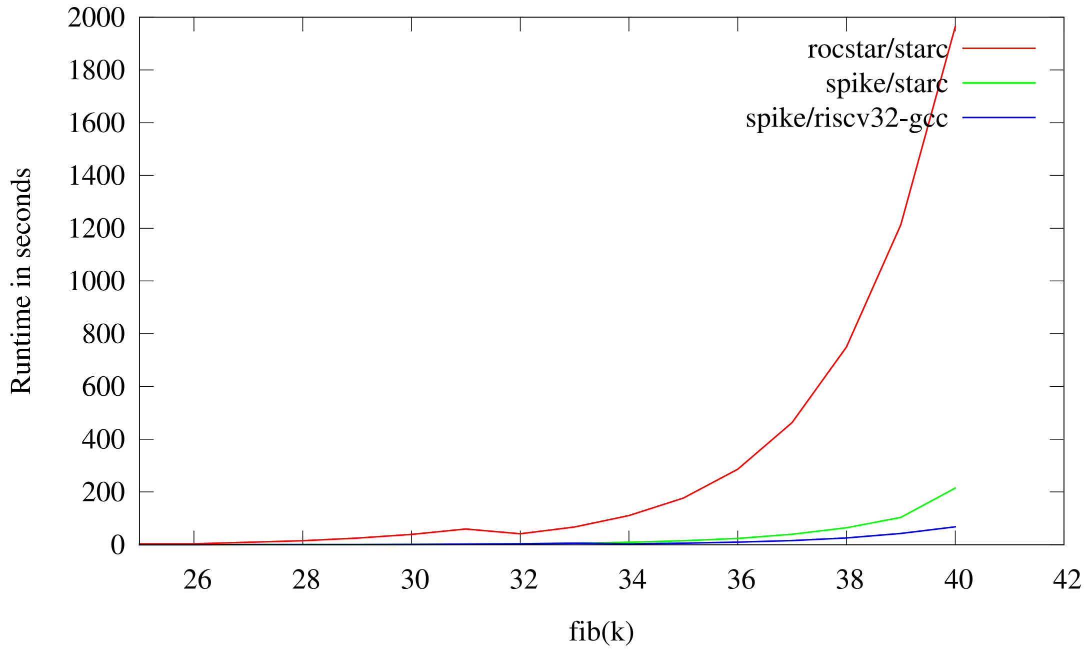
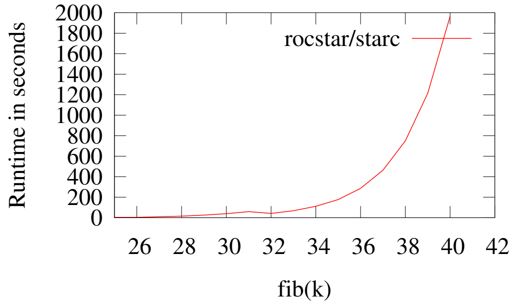
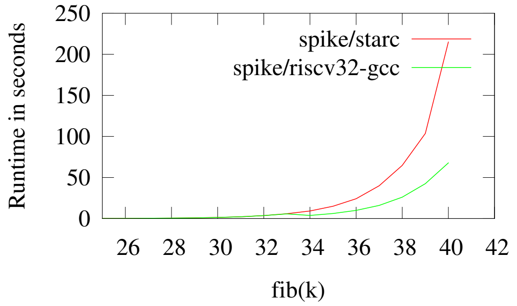


Figure 3.1.: Runtime of `fib.c` with different input size

The numbers have been measured up to $k = 40$ because the combination of `rocstar` and `starc` took roughly 30 minutes to finish execution. The `spike` platform seems to be much faster. Since Selfie's emulator is not designed to be fast, some simple operations are implemented in a way that slows down the system. One example are shift operators which are frequently used, but implemented with a `while` loop and multiplication/division. This is done because `starc` does not support shift operators.
The second performance test was carried out by self-compiling Selfie in different versions of cascading. Table 3.3 shows different constellations and their runtime. Cascading multiple emulator instances increases the runtime extremely.



(a) Runtime of Fibonacci on `rocstar`, binary generated by `starc`



(b) Runtime of Fibonacci on `spike` with binaries produced by `starc` and `gcc`

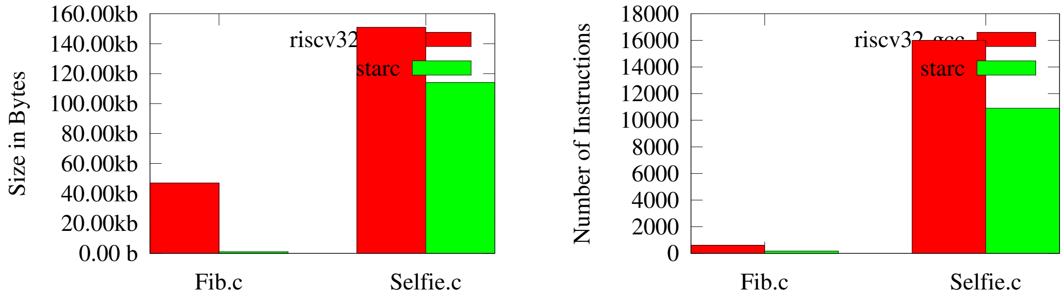
Figure 3.2.: Same as Figure 3.1 but separated `rocstar` and `spike`.

Command	
Compilation Stack	Runtime
`selfie -c selfie.c -o selfie.o`	
direct compilation	0.3 sec
`selfie -l selfie.r -m 4 -c selfie.c -o selfie.o`	
one rocstar emulator instance	711.5 sec
`spike pk selfie -c selfie.c -o selfie.o`	
one spike/pk emulator instance	65.737 sec

Table 3.3.: Self-compilation Times

3.5. Code Comparison

The next graphs show a statistical analysis of the output generated by the RISC-V `gcc` cross-compiler and by `starc`. Figure 3.3a and 3.3b show the file size in bytes and the number of instructions emitted for the two C files `fib.c` and `selfie.c` generated by `gcc` or `starc` respectively.
`starc` produces fewer instructions than `gcc`. The difference is especially large for smaller binaries. `Gcc` outputs a lot of libraries by default which are either partly unused or used during the startup phase only. For bigger programs, the differences are not that big anymore which can be seen in the case of `selfie.c` program.
Figure 3.4 shows the distribution of instructions for the fib.c program. The red bars are RISC-V instructions used by the `gcc` RISC-V cross-compiler and the green bars are instructions used by `starc`. Since Selfie is using only a tiny subset of RISC-V, not all instructions can be used by `starc`. Both compilers make heavy use of `li`, `lw`, `addi` instructions. As already said, `gcc` produces much more instructions than `starc` for small C programs.
Figure 3.5 shows the number of instructions for the `selfie.c` C program. It excludes instructions which are used fewer than 25 times inside the code. The graph suggests that both compilers produce roughly the same set of instructions.



(a) File size in bytes of generated binary for either fib.c or selfie.c

(b) Number of instructions generated for either fib.c or selfie.c

Figure 3.3.: Difference of file size and instruction counts. Both programs have been compiled with `gcc` and `starc` and the numbers are based on a disassemble output generated by `objdump`.

3.6. Conclusion

This work presented the ELF binary format and how it was added to the Selfie Project. The ELF binary format seems complicated at first, but after dealing with it for some time, the format feels quite simple and usable. The main advantage of using the ELF binary format is to bridge the gap between an artificial teaching system and a real world toolchain. Therefore, I think by extending Selfie to support ELF binaries, a very natural and simple enhancement has been made that has not made the whole platfor much more complicated.

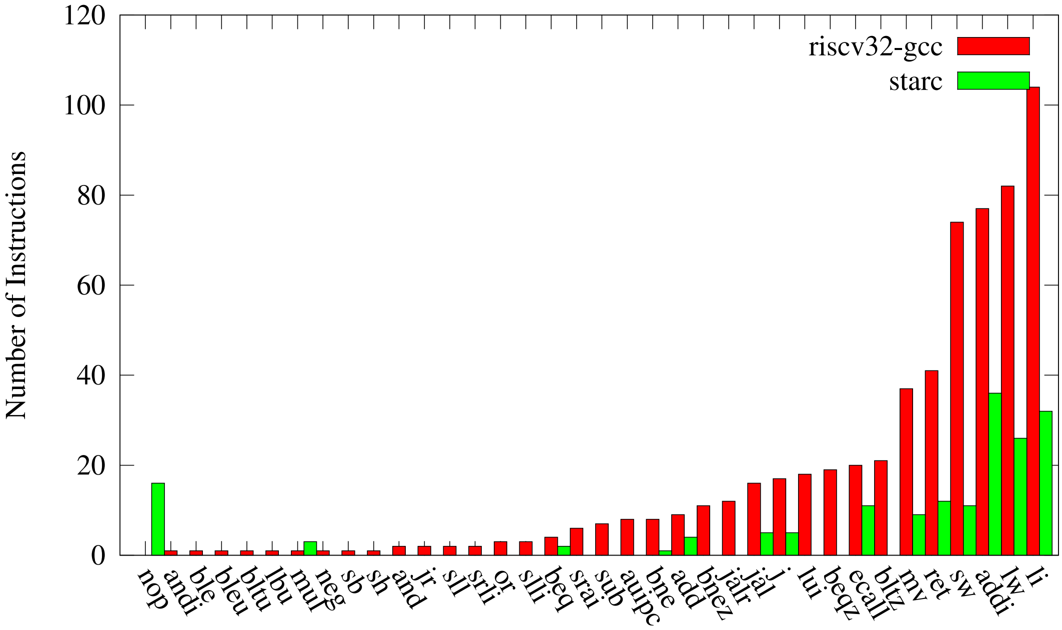


Figure 3.4.: Distribution of instructions

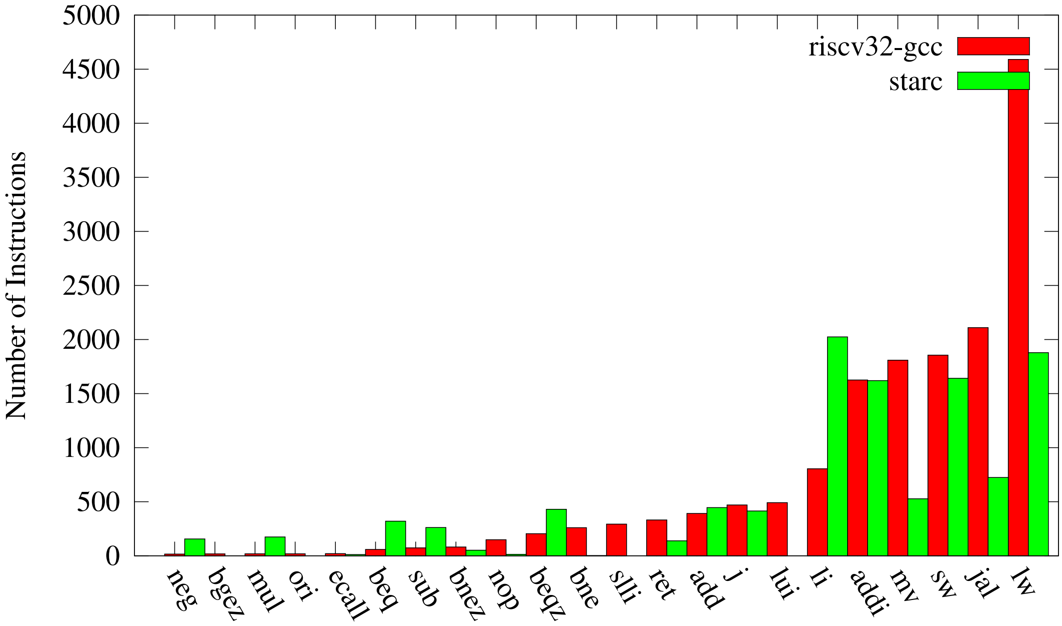


Figure 3.5.: Distribution of instructions for selfie.c

A. Appendix

A.1. Sample Fibonacci Source Code

```c
int fib(int n) {
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}

int main(int argc, int *argv) {
  exit(fib(9));
}
```

Listing A.1: Fibonacci Code

A.2. Objdump of Fibonacci

```
fib:     file format elf32-littleriscv


Disassembly of section .text:

00010000 <.text>:
   10000:       04000293        li      t0,64
   10004:       40000313        li      t1,1024
   10008:       025302b3        mul     t0,t1,t0
   1000c:       2d428293        addi    t0,t0,724
   10010:       00028193        mv      gp,t0
   10014:       02201663        bne     zero,sp,0x10040
   10018:       03f00293        li      t0,63
   1001c:       40000313        li      t1,1024
   10020:       025302b3        mul     t0,t1,t0
   10024:       3ff28293        addi    t0,t0,1023
   10028:       40000313        li      t1,1024
```

```
    1002c:      025302b3            mul     t0,t1,t0
    10030:      3fc28293            addi    t0,t0,1020
    10034:      0002a103            lw      sp,0(t0)
    10038:      2d400313            li      t1,724
    1003c:      00030193            mv      gp,t1
    10040:      00000013            nop
          ...
    1007c:      00000013            nop
    10080:      00410293            addi    t0,sp,4
    10084:      ffc10113            addi    sp,sp,-4
    10088:      00512023            sw      t0,0(sp)
    1008c:      1f4000ef            jal     0x10280
    10090:      ffc10113            addi    sp,sp,-4
    10094:      00a12023            sw      a0,0(sp)
    10098:      00012503            lw      a0,0(sp)
    1009c:      00410113            addi    sp,sp,4
    100a0:      05d00893            li      a7,93
    100a4:      00000073            ecall
    100a8:      00012603            lw      a2,0(sp)
    100ac:      00410113            addi    sp,sp,4
    100b0:      00012583            lw      a1,0(sp)
    100b4:      00410113            addi    sp,sp,4
    100b8:      00012503            lw      a0,0(sp)
    100bc:      00410113            addi    sp,sp,4
    100c0:      03f00893            li      a7,63
    100c4:      00000073            ecall
    100c8:      00008067            ret
    100cc:      00012603            lw      a2,0(sp)
    100d0:      00410113            addi    sp,sp,4
    100d4:      00012583            lw      a1,0(sp)
    100d8:      00410113            addi    sp,sp,4
    100dc:      00012503            lw      a0,0(sp)
    100e0:      00410113            addi    sp,sp,4
    100e4:      04000893            li      a7,64
    100e8:      00000073            ecall
    100ec:      00008067            ret
          ...
```

Listing A.2: Shortened `riscv32-unkown-linux-objdump` of a simple program (the fibonacci programming in A.1

A.3. ELF Dump of Fibonacci

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
```

```
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x10000
  Start of program headers:          52 (bytes into file)
  Start of section headers:          84 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         1
  Size of section headers:           40 (bytes)
  Number of section headers:         4
  Section header string table index: 3

Section Headers:
  [Nr] Name      Type      Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]           NULL      00000000 000000 000000 00      0   0  0
  [ 1] .text     PROGBITS  00010000 000110 0002d4 00 WAX  0   0  0
  [ 2] .data     PROGBITS  000102d8 0003e4 000000 00 WAX  0   0  0
  [ 3] .shstrtab STRTAB    00000000 0000f4 0002d4 00      0   0  0

Program Headers:
  Type Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD 0x000110 0x00010000 0x00000000 0x002d4 0x002d4 RWE 0x1000
```

Listing A.3: Full ELF32 dump generated by `riscv32-unknown-elf-readelf`. The binary was generated by `starc` (simple Fibonacci program)

A.4. ELF Code in Selfie

```
void createELFHeader() {
  int startOfProgHeaders;
  int startOfSecHeaders;
  int stringBytes;

  startOfProgHeaders = 52;
  startOfSecHeaders  = 84;
  stringBytes        = 24;

  // store all numbers necessary to create a valid
  // ELF header incl. program header and section headers.
  // For more info about specific fields, consult ELF documentation.
  ELF_header = malloc(ELF_HEADER_LEN);

  // ELF magic number
  *(ELF_header + 0) = 1179403647; // part 1 of ELF magic number
  *(ELF_header + 1) = 65793;      // part 2 of ELF magic number
  *(ELF_header + 2) = 0;          // part 3 of ELF magic number
  *(ELF_header + 3) = 0;          // part 4 of ELF magic number

```

```
  // ELF Header
  *(ELF_header + 4)  = 15925250; // Type and Machine fields (16 bit each)
  *(ELF_header + 5)  = 1;        // Version number
  *(ELF_header + 6)  = ELF_ENTRY_POINT;
  *(ELF_header + 7)  = startOfProgHeaders;
  *(ELF_header + 8)  = startOfSecHeaders;
  *(ELF_header + 9)  = 0;        // Flags
  *(ELF_header + 10) = 2097204;  // Size of header, program header
  *(ELF_header + 11) = 2621441;  // number of program header / section header
  *(ELF_header + 12) = 196612;   // number of section headers/ SHDR string table

  // Program Header
  createELFProgramHeader(1, ELF_HEADER_LEN+4, ELF_ENTRY_POINT,
                         0, binaryLength, binaryLength, 7,
                            4096);

  // Section Header 0 (Zero-Header)
  createELFSectionHeader(21, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);

  // Section Header 1 (.text)
  createELFSectionHeader(31, 1, 1, 7,  ELF_ENTRY_POINT,
    ELF_HEADER_LEN + 4, codeLength, 0,  0, 0, 0);

  // Section Header 2 (.data)
  createELFSectionHeader(41, 7, 1, 7, ELF_ENTRY_POINT + 4 + codeLength,
    ELF_HEADER_LEN + 4 + codeLength, binaryLength - codeLength, 0, 0, 0, 0);

  // Section Header 3 (.shstrtab)
  createELFSectionHeader(51, 13, 3, 0, 0, ELF_HEADER_LEN - stringBytes,
    codeLength, 0, 0, 0, 0);

  // String table
  *(ELF_header + 61) = 1702112768;  // 0.te
  *(ELF_header + 62) = 771781752;   // xt0.
  *(ELF_header + 63) = 1635017060;  // data
  *(ELF_header + 64) = 1752378880;  // 0.sh
  *(ELF_header + 65) = 1953657971;  // strt
  *(ELF_header + 66) = 25185;       // ab
}
```

```

void createELFProgramHeader(int type, int offset, int vaddr,
                            int paddr, int fsize, int memsize,
                            int flags, int align)
{
  *(ELF_header + 13) = type;    // Type of program header (LOAD)
  *(ELF_header + 14) = offset;  // Offset to 1. byte of segment
  *(ELF_header + 15) = vaddr;   // Virtual address
  *(ELF_header + 16) = paddr;   // Physical address
  *(ELF_header + 17) = fsize;   // File size
  *(ELF_header + 18) = memsize; // Memory size
  *(ELF_header + 19) = flags;   // Flags (Read, Write, Execute)
  *(ELF_header + 20) = align;   // Alignment of segments
}

void createELFSectionHeader(int start, int name, int type,
                            int flags, int addr, int off,
                            int size, int link, int info,
                            int align, int entsize)
{
  *(ELF_header + start)     = name;
  *(ELF_header + (start+1)) = type;
  *(ELF_header + (start+2)) = flags;
  *(ELF_header + (start+3)) = addr;
  *(ELF_header + (start+4)) = off;
  *(ELF_header + (start+5)) = size;
  *(ELF_header + (start+6)) = link;
  *(ELF_header + (start+7)) = info;
  *(ELF_header + (start+8)) = align;
  *(ELF_header + (start+9)) = entsize;
}
```

Listing A.4: Full ELF header generation code within selfie.

List of Figures

Listings

List of Tables

Nomenclature

CISC Complex instruction set computing is a processor design where instructions are capable of executing several low level operations

`.COM` Very simple binary format used in the MS-DOS operating system

API Application Programmer Interface. Usually, it describe a interface between a software layer and another application

ASCII American Standard Code for Information Interchange

COFF Common Object File Format

CPU Central Processing Unit

ELF Executable and Linkage Format

errno Error variable used by the `libc` to provide detailed reasons why a system call failed

FP Frame Pointer: a pointer to the current execution frame of a function

GB Abbreviation for Gigabyte, which is 1024MB

gcc Refers to either the GNU Compiler Collection or the `gcc` compiler program that compiles code to machine instructions

GNU Acronym for *GNU is not Unix*. Is is a collection of software that is licensed under the GPL license. The GNU project was invented by Richard M. Stallman.

GP Global Pointer: Pointer to the beginning of the local variable sections. Local Variables are addressed by adding an offset to this variable.

ISA Abbreviation for Instruction Set Architecture, describing a particular set of instructions which are provided by a CPU

JAL Jump and Link operation

KB Abbreviation for Megabyte, which is 1024 Bytes

ld GNU `ld` command to link object files

LLVM Low Level Virtual Machine is a collection of reusable compilers and related technologies. It is used to develop front- and back ends.

LSB 2's complement with least significant byte at the lowest address

macOS Operating System that is developed by Apple Inc. and based on a MACH microkernel and FreeBSD and partly open source.

MB Abbreviation for Megabyte, which is 1024KB

MSB 2's complement with most significant byte at the lowest address

OS Operating System

PC Program Counter: a pointer to the current instruction in memory

PHDR Program Header Table of the ELF binary format

PIC Position Independent Code: Code that does not rely on fixed/absolute addresses

pk Proxy Kernel provided by the RISC-V foundation

RISC Reduced instruction set computing

RV32i RISC-V 32 Bit instruction set

SHDR Section Header Table of the ELF binary format

SVR4 UNIX System V, one of the first commercial Unix systems developed by AT&T and released in 1983.

X86 A microprocessor based on Intel 8086. It implements a CISC instructions.

Bibliography

[1] Simone Oblasser. Porting Selfie to RISC-V: State-of-the-Art ISA Support, 2017. URL http://selfie.cs.uni-salzburg.at/.

[2] The Selfie Project. URL http://selfie.cs.uni-salzburg.at.

[3] The RISC-V Foundation, . URL https://riscv.org.

[4] Software Tools - RISC-V Foundation, . URL https://www.riscv.org/software-tools.

[5] Waterman A. et al. The RISC-V Instruction Set Manual: Volume I - User-Level ISA, 2016. URL https://www.riscv.org/specifications.

[6] GNU Binutils - a collection of binary tools. GNU Website, 2017. URL https://www.gnu.org/software/binutils/.

[7] TIS Commitee. Tool Interface Standard (tis): Executable and Linkage Format (ELF) specification, ver12, 1995. URL http://refspecs.linuxbase.org/elf/elf.pdf.

[8] Richard Chang. C Function Call Conventions and the Stack, 2001. URL https://www.csee.umbc.edu/~chang/cs313.s02/stack.shtml.

[9] a.out - assembler and link editor output. URL http://man.cat-v.org/unix-6th/5/a.out.

[10] a.out. Wikipedia, the free encyclopedia, 2016. URL https://en.wikipedia.org/wiki/A.out.

[11] Dimitri van Heesch. Mips COFF Specification: coff.h, 2002. URL http://www-scf.usc.edu/~csci402/ncode/coff_8h-source.html.

[12] G. Gircys. *Understanding and Using COFF*. O'Reilly, 1988. ISBN 978-0-937175-31-6.

[13] Wikipedia: Executable and linkable format. Wikipedia, the free encyclopedia, . URL https://en.wikipedia.org/wiki/Executable_and_Linkable_Format.

[14] Frequently Asked Questions for FreeBSD 2.x: Why use (what are) a.out and ELF executable formats?, . URL https://docs.freebsd.org/doc/2.2.7-RELEASE/usr/share/doc/FAQ/FAQ204.html.

[15] J.R. Levine. *Linkers & Loaders*. Morgan Kaufmann, 2000. ISBN 978-155-860-4964.

[16] Web: Linkers and Loaders, 2016. URL `http://www.iecc.com/linker/`.

[17] Palmer Dabbelt. GNU Compiler Mailinglist: Palmer Dabbelt: New Port for RISC-V, 2017. URL `https://gcc.gnu.org/ml/gcc-patches/2017-01/msg00776.html`.

[18] Palmer Dabbelt. GNU Compiler Mailinglist: Palmer Dabbelt: RISC-V Port Commit, r245224, 2017. URL `https://gcc.gnu.org/ml/gcc-cvs/2017-02/msg00147.html`.

[19] Jeff Duntemann. *Assembly Language Step by Step - Progrmaming with Linux*. Wiley Publishing, Inc, 2009. ISBN 978-0-470-49702-9.

[20] Peter v.d. Linden. *Expert C Programming - Deep C Secrets*. Prentice Hall, 1994. ISBN 978-0131774292.

[21] MIPS Technologies. MIPS Architecture for Programmers Volume II: The MIPS32 Instruction Set, 2001. URL `http://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS_Vol2.pdf`.

[22] Simone Oblasser. Git commit 981a344: replaced all emits, minor bugfixes, 2016. URL `http://bch.at/?id=NVSDS2615`.

[23] Andrew Waterman. riscv-pk, ELF Loading Mechanism, elf.c, 2015. URL `http://bch.at/?id=LYYFG3811`.

[24] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. System V Application Binary Interface, AMD64 Architecture Processor Supplement, 2010.

[25] mmap manual page from OpenBSD, 1994. URL `http://man.openbsd.org/mmap`.

[26] M. McKusick and G. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, Inc., 2005.

[27] brk manual page from OpenBSD, 1979. URL `http://man.openbsd.org/brk`.

[28] brk manual page from Linux. URL `https://linux.die.net/man/2/brk`.

[29] Christian Barthel. Git commit 4a37572/riscv-pk: Add ftruncate syscall, 2016. URL `http://bch.at/?id=SXRJS2917`.

[30] Hisham Muhammad. htop - an interactive process viewer for unix, 2006. URL `http://hisham.hm/htop/`.

[31] Apple Inc. sbrk implementation on macOS: brk.c, 1999. URL `https://opensource.apple.com/source/Libc/Libc-763.12/emulated/brk.c?txt`.

[32] Christian Barthel. Git commit 67e0a38: implemented feedback, remove pkcompile flag, 2016. URL http://bch.at/?id=JIYBW2914.

[33] Simone Oblasser. Git commit 9be4c5/selfie running on all platforms incl. spike/pk, 2016. URL http://bch.at/?id=EWYLI3036.

[34] Christoph Kirsch. Git commit 8bdf07/fixed and improved console argument handlink, linker is done, 2016. URL http://bch.at/?id=CLIYB3039.

[35] Simone Oblasser. Git commit 981a34/replaced all emits, minor bugfixes, 2016. URL http://bch.at/?id=MDXPQ3038.

[36] Christian Barthel. Git commit 67e0a3/implemented feedback, remove pkcompile flag, 2016. URL http://bch.at/?id=VTNHL3040.

[37] Christian Barthel. Git commit 5552e4/introduce elf header with proper program headers and section headers, 2016. URL http://bch.at/?id=IKGRN3044.

[38] Dwarf debugging format. Wikipedia, the free encyclopedia, 2017. URL http://en.wikipedia.org/wiki/DWARF.

[39] Open virualization format. Wikipedia, the free encyclopedia, 2016. URL https://en.wikipedia.org/wiki/Open_Virtualization_Format.

[40] Risc-v/selfie VM, 2017. URL http://student.cosy.sbg.ac.at/~cbarthel/sr5.

[41] Debian - The Universal Operating System, 2016. URL https://www.debian.org.

[42] VMware Workstation for Windows, 2016. URL https://www.vmware.com/products/workstation.html.

[43] Fibonacci numbers. Wikipedia, the free encyclopedia, 2017. URL http://de.wikipedia.org/wiki/Fibonacci-Folge.