Bachelor Thesis

Porting Selfie to RISC-V: State-of-the-Art ISA Support

Simone Oblasser

simone.oblasser@stud.sbg.ac.at

Department of Computer Sciences
University of Salzburg
Austria

May 9, 2017

Advisor

Professor Christoph Kirsch

ck@cs.uni-salzburg.at

Contents

Abstract

The following thesis describes a newly developed alternative version of Selfie, a self-referential platform containing a compiler, an emulator and a hypervisor for teaching computer systems. Instead of MIPS32 code, this version supports a minimal subset of RISC-V (RV32I plus "M" standard extension), an open-source ISA developed at University of California, Berkeley.

The port to this different architecture included the adaption of all major parts of Selfie, especially of the compiler and emulator. The key changes made involved register usage, encoding and decoding of the new instruction formats, and handling of immediate values. The resulting platform still fulfills all the self-referentiality requirements the original project did.

Furthermore, the new support for RISC-V instructions paved the way for an additional project to make binaries generated by Selfie (and, in turn, Selfie itself) executable on official RISC-V emulators and hardware chips.

1 Introduction

1.1 The Selfie Project

Selfie is a project of the Computational Systems Group at the Department of Computer Sciences of the University of Salzburg in Austria. It combines a self-compiling compiler (`starc`), a self-executing emulator (`mipster`), a self-hosting hypervisor (`hypster`) and a tiny library (`libcstar`) into one educational platform which is fully self-referential. This means that the compiler can compile its own source code, the emulator can emulate itself and the hypervisor can host itself, and every form of cascading the components is possible.

The project's source code is written in a subset of C called C* which only supports 32-bit integers, 32-bit pointers and character and string literals, but is Turing-complete. Selfie's original version compiles to and executes assembly code of a subset of MIPS32 called MIPSter.

All parts of Selfie are kept minimal and are therefore implemented in a single file. They are mainly designed to teach both undergraduate and graduate students the construction of compilers, operating systems and virtual machine monitors and to help them understand the problem of self-referentiality in such systems [1], [2].

1.2 MIPS

MIPS is a reduced instruction set computer (RISC) architecture developed by MIPS Technologies Inc. (now Imagination Technologies) that is available for licensing. Since its introduction in 1985, multiple 32- and 64-bit versions of the instruction set architecture (ISA) and even a compressed version supporting 16-bit instructions called microMIPS have been developed [3]. Selfie's original version utilizes less than 20 instructions of the MIPS32 ISA, all of which are listed in Section 2.

1.3 Problem Description

Since Selfie is a platform that is mainly used for teaching, it is important for it to support an ISA that is easy to teach and to understand, but that is also up-to-date and that has the potential of being used in projects of other organizations as well. For students, working with an ISA that is actively being improved and that affordable hardware chips are built for could have huge benefits. On the one hand, MIPS instruction formats are straightforward and easy to memorize, but on the other hand, the ISA is proprietary, limiting who can build chips for it. This created the need for transitioning to an instruction set architecture that increases the possible ways Selfie can be used to experiment with. Also, having two versions of Selfie offers the opportunity to really comprehend the differences between two ISAs and where they affect the platform in which way.

1.4 RISC-V

The most promising candidate for being used in the new version of Selfie was RISC-V. In contrast to MIPS, RISC-V is a relatively new RISC ISA designed to support architecture research and education and was originally developed at the University of Berkeley. Its origin in education has caused many projects containing RISC-V to be realized by different Universities. One of its biggest advantages is that it is an open ISA with a small but powerful base instruction set. It is also highly extensible and supports highly-parallel multicore implementations and is fully virtualizable, which could be used to extend Selfie in future projects [4].

RISC-V's clean, modular design has attracted partners like Google and Oracle and has also been selected as the Best Technology of 2016 in the Linley Group's Analysts' Choice Awards [5].

2 Comparison of MIPS and RISC-V Instruction Set Architectures

As both MIPS and RISC-V are RISC instruction set architectures, they are very similar in their core features. For example, both implement a load-store or register-register architecture which supports arithmetic and logic operations only between registers and requires load and store instructions to access memory.

Additionally, they both support 32-bit versions, which is essential in order to keep as much of Selfie's original code as possible, making it easier to comprehend the key differences between both architectures.

One of these differences is the encoding of instructions, which is (in principal) variable in RISC-V, but fixed in MIPS [6], [7]. However, all RISC-V instructions relevant to Selfie are either in the base ISA (RV32I) or the "M" Standard Extension for Integer Multiplication and Division, which both contain only instructions with fixed-length encoding.

In the following, differences and similarities between the two instruction set architectures which are relevant to Selfie are discussed further.

2.1 Calling Convention and Register Usage

The MIPS ISA utilizes 32 32-bit general-purpose registers for storing both addresses and data needed for executing operations. Most of these registers can be used freely, although there are conventions to use them in a certain way. Table 1 lists all general-purpose registers of MIPS. Since Selfie does not suppport floating point numbers, there is no need to discuss MIPS floating point registers.

In addition to register names and their conventional purposes, the table shows whether a register shall be saved by the caller or callee in case of a function call. For the most part, Selfie's code follows these conventions. Examples where this is not the case are the registers `s0 - s7`, which are dedicated to store temporaries that are needed to be preserved across procedure calls, or the registers `k0` and `k1`, which are reserved for the OS kernel. These registers are not used at all, so there is no immediate need to save them [8, chapter A.6].

Another special case are the registers `a0 - a3`, which are only used to store arguments for system calls but not for other function calls. Instead, all parameters are pushed onto the stack for reasons of simplification.

Like MIPS, RISC-V also provides 32 32-bit general-purpose registers with roughly the same conventions (see Table 2). There is an always-zero register

Number	**Name**	**Description**	**Saver**
0	zero	Hard-wired zero	-
1	at	reserved for assembler	-
2-3	v0-1	Return values	-
4-7	a0-3	Function arguments	Caller
8-15	t0-7	Temporaries	Caller
16-23	s0-7	Saved registers/temporaries	Callee
24-25	t8-9	Temporaries	Caller
26-27	k0-1	reserved for OS kernel	-
28	gp	Global pointer	-
29	sp	Stack pointer	-
30	fp	Frame pointer	Callee
31	ra	Return address	Callee

Table 1: MIPS register usage and calling convention

with the number zero, but registers for the stack, global and frame pointer as well as for the return address have differing register numbers across the ISAs. Another difference is the amount of temporary registers (ten in MIPS, but only seven in RISC-V). Conversely, RISC-V provides more saved temporary registers than MIPS, although this offers no advantage for use in Selfie [6, chapter 20].

Apart from the general-purpose registers, MIPS has additional user-visible registers: the program counter (`pc`) for indicating which instruction to execute next and registers `HI` and `LO`, which store the highest and lowest 32 bits of a multiplication or the quotient and remainder of a division [7, chapter 3]. The RISC-V ISA also uses the program counter, but has no need for the other two registers since there are dedicated instructions for obtaining the results that MIPS usually stores in them [6, chapter 2.1]. These instructions are described in detail in Section 2.3.

2.2 Instruction Formats

There are three principal formats for encoding MIPS32 instructions: the R-format for register instructions, the I-format for instructions handling immediates and the J-format for dealing with jumps across the assembly code. A graphical illustration can be seen in Figure 1 [8, p. 82/83, 113].

Number	Name	Description	Saver
0	zero	Hard-wired zero	-
1	ra	Return address	Caller
2	sp	Stack pointer	Callee
3	gp	Global pointer	-
4	tp	Thread pointer	-
5-7	t0-2	Temporaries	Caller
8	s0/fp	Saved register/frame pointer	Callee
9	s1	Saved register	Callee
10-11	a0-1	Function arguments/return values	Caller
12-17	a2-7	Function arguments	Caller
18-27	s2-11	Saved registers	Callee
28-31	t3-6	Temporaries	Caller

Table 2: RISC-V register usage and calling convention

These formats or types include a varying amount of fields according to their respective purpose, but instructions of all types are always limited to a total size of 32 bits. For example, R-type instructions need to execute the operation described through the fields `opcode` and `funct` on two general-purpose registers (`rs` and `rt`) and store the result in a third register `rd`. Bitwise operations also need a field for storing the shift amount `shamt`, which is never used in Selfie.

I-type instructions operate on a base register `rs` together with an offset of `immediate` bits and store the result in register `rt`. The reduction of required fields leads to a bigger field for the immediate value, which has 16 bits. Additionally, I-type instructions can be identified solely through their opcode.

Lastly, instructions of the J-type do not need to encode any registers at all. They just jump (or set the program counter) to the specified `address`, which can be in a range between 0 and 2^{28}.

In contrast to MIPS, the RISC-V ISA encodes instructions using a few more formats. The four core instruction formats are the R-type and I-type, which are similar to the corresponding MIPS variants, and the S-type and U-type. There are also two further variants, namely the SB-type and UJ-type based on the S- and U-type respectively (see Figure 2).

Similarily to MIPS, the individual instruction fields are kept in the same positions as far as possible for all instruction formats in order to simplify decoding.

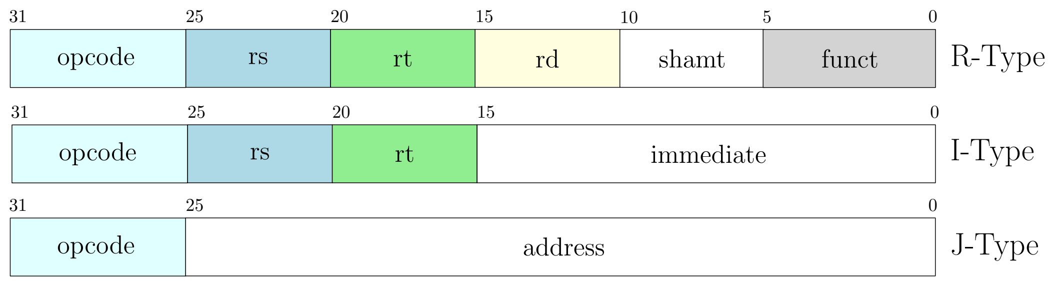


Figure 1: MIPS instruction formats

However, the order of the fields seems to be reversed and immediates are always packed as far left as possible, placing their sign bit in bit 31.

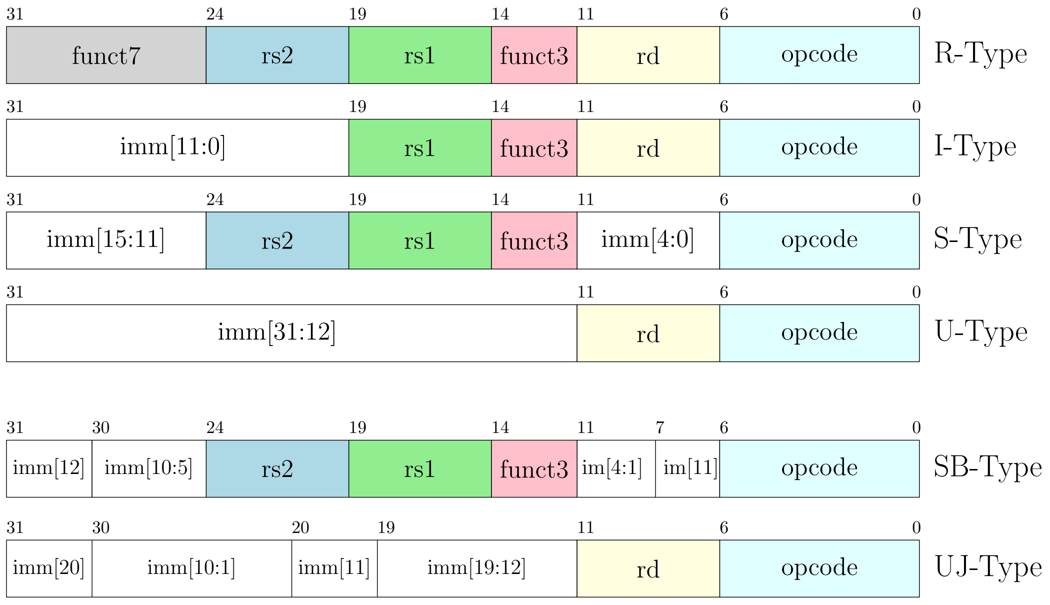
A key difference can be observed between the immediates of both ISAs. While they have a length of 16 bits in MIPS, there are usually only 12 bits for immediates in RISC-V formats. This was designed in favor of small immediates and those requiring all 32 bits, making an asymmetric split of 12 bits plus 20 bits with an extra instruction the consequence [6, p. 11].

The additional two RISC-V instruction formats were created for practical reasons. For example, the SB-type is used for branch instructions, which shall only encode offsets in multiples of two. The SB-type makes use of this and allows for including an extra bit, doubling the possible range for the offset. The UJ-type is useful for creating even immediates for jumps, while the U-type (in form of a load upper immediate) is used to create a 32-bit immediate out of a 12-bit and 20-bit immediate [6, chapter 2.3].

Although there are six instruction formats, the following section will only describe instructions based on the five types R, I, S, SB and UJ, since U-type instructions are not necessary to support all features of Selfie.

2.3 Instructions Used in Selfie

The original MIPS32-version of Selfie aims at requiring the smallest subset of instructions possible to function as described. This aim is also targeted when replacing the superset with RISC-V. While many of the used MIPS instructions can also be found in the other ISA, some need to be replaced with one or two similar instructions, and others can be removed entirely. In the following all instructions from the MIPS32 ISA previously used by Selfie are listed and possible RISC-V replacements are discussed [7, 6].

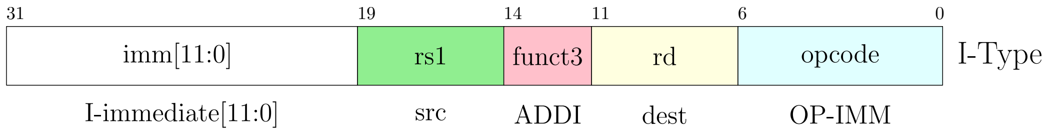


31 24 19 14 11 6 0
funct7 rs2 rs1 funct3 rd opcode R-Type
imm[11:0] rs1 funct3 rd opcode I-Type
imm[15:11] rs2 rs1 funct3 imm[4:0] opcode S-Type
imm[31:12] rd opcode U-Type
imm[12] imm[10:5] rs2 rs1 funct3 im[4:1] im[11] opcode SB-Type
imm[20] imm[10:1] imm[11] imm[19:12] rd opcode UJ-Type

Figure 2: RISC-V instruction formats

ADDIU The *Add Immediate Unsigned Word* instruction is used to add a signed 16-bit immediate value to the value of a general-purpose register. A possible overflow is not being trapped due to the unsigned nature of the operation.

As there is generally no support for overflow checks on integer arithmetic operations in RISC-V, this instruction can be replaced with a RISC-V `ADDI`:



31 19 14 11 6 0
imm[11:0] rs1 funct3 rd opcode I-Type
I-immediate[11:0] src ADDI dest OP-IMM

Figure 3: RISC-V ADDI instruction format

ADDU and SLT The *Add Unsigned Word* instruction is similar to `ADDIU`, but uses the value of a second register instead of an immediate to perfom the addition on. Again, the word unsigned refers to possible overflows being ignored and has nothing to do with adding unsigned values.

The same reasoning as with the `ADDIU` substitution applies here, so a RISC-V `ADD` can be used instead. *Set On Less Than* and `ADDU` have almost the same encoding in MIPS but also in RISC-V, which can be seen in Figure 4.

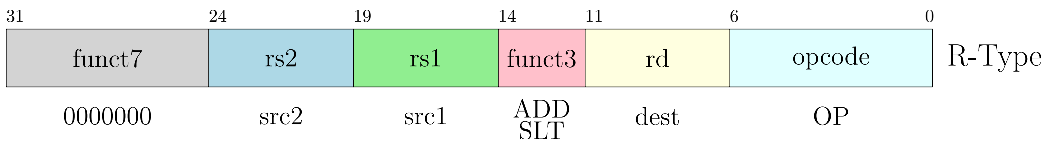


Figure 4: RISC-V ADD and SLT instruction format

SUBU Subtract Unsigned Word works the same way as `ADDU`, but for subtractions. Similarily, a RISC-V `SUB` is the right substitution.

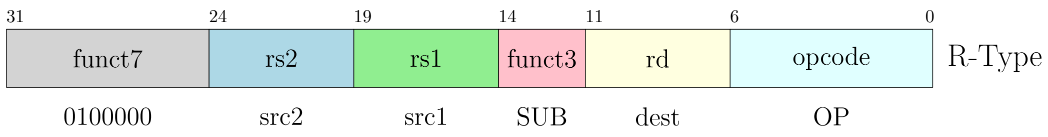


Figure 5: RISC-V SUB instruction format

LW and SW Since both ISAs are load-store architectures, they both support *Load Word* (Figure 6) and *Store Word* (Figure 7) instructions which are used to load data from memory into a specified general-purpose register or store data from a register to a memory address, respectively.

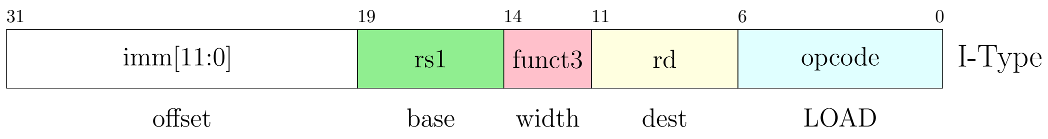


Figure 6: RISC-V LW instruction format

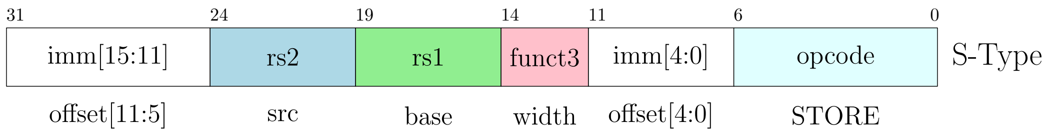


Figure 7: RISC-V SW instruction format

JAL Selfie uses *Jump And Link Register* instructions when jumping to subroutines. The address of the next instruction after returning from the subroutine, which equals the current program counter plus four bytes is (by convention) placed into the return address register. RISC-V provides a JAL instruction in its base instruction set that fulfills the same function, but needs an explicit declaration of the register where the return address shall be placed into (Figure 8).

It is important to note that JAL instructions in Selfie's MIPS version require absolute addresses (which is not entirely conform to the specification), while RISC-V JAL instructions are pc-relative.

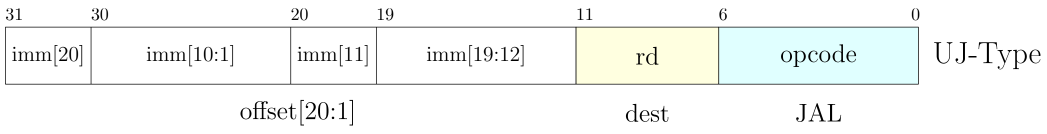


Figure 8: RISC-V JAL instruction format

J *Jump* instructions without linking are used in Selfie for jumping directly to the end of a procedure when a return statement has been parsed. In this case, there is no need to save the position of the next instruction since it will never be executed.

There is no dedicated *Jump* instruction in RISC-V. Instead, a JAL instruction where the zero register is used as destination register serves as the replacement.

JR *Jump Register* acts as a counterpart to *Jump And Link*. When a subroutine is finished, a JR instruction jumps back to the caller, i. e. to the address that is in the conventional return address register.

Just like with JAL, there is a RISC-V instruction called *Jump and Link Register* (Figure 9) which does the same as the original MIPS instruction, but requires an explicit return register to be set as destination. Since the original JR instruction did not additionally link anywhere, the destination can be set to the zero register. Also, the same pitfall as with JAL applies here: JR in Selfie's MIPS version does absolute jumps, while those of JALR in RISC-V are pc-relative.

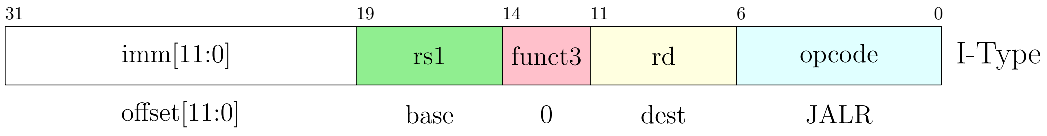


Figure 9: RISC-V JALR instruction format

BEQ and BNE *Branch on Equal* and *Branch on Not Equal* instructions exist in both ISAs and are very similar, apart from different branching ranges. In MIPS the branch can be in a range of ± 128 KiB of the program counter, in RISC-V the range is only ± 4 KiB due to fewer bits available for encoding the offset.

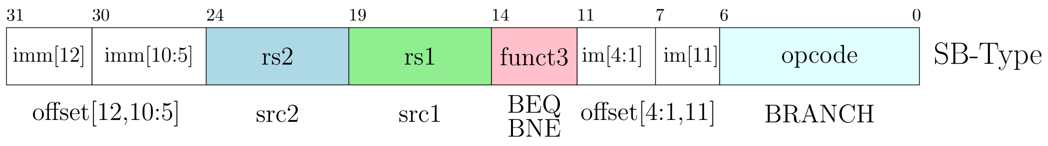


Figure 10: RISC-V BEQ and BNE instruction format

NOP Unlike MIPS, RISC-V instructions for unconditional jumps and conditional branches have no architecturally visible delay slots. This means the `NOP` instruction is no longer required to be implemented in the RISC-V version of Selfie.

MULTU *Multiply Unsigned Word* multiplies two unsigned integers stored in general-purpose registers. The result is usually an unsigned 64-bit value whose low-order 32 bits are placed in the special register `LO` and whose high-order 32 bits are placed in `HI`.

In RISC-V, this operation is covered by two separate instructions, `MUL` and `MULHU`, making the registers `HI` and `LO` obsolete. However, Selfie only supports 32-bit types and cannot handle results bigger than this, so only the lower 32 bits of the result are stored, meaning the correct replacement for `MULTU` is only the `MUL` instruction (Figure 11).

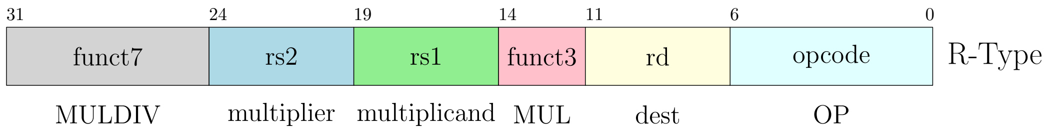


Figure 11: RISC-V MUL instruction format

DIVU Similarily to `MULTU`, *Divide Usigned Word* makes use of the two special registers. It divides two unsigned integer values and stores the 32-bit quotient into register `LO`, while the 32-bit remainder is stored in `HI`.

Again, this is achieved by using two separate instructions, `DIVU` and `REMU` (Figure 12), making the special registers unnecessary. Selfie supports both the division and the remainder operators, so both instructions are required to be used by the RISC-V version.

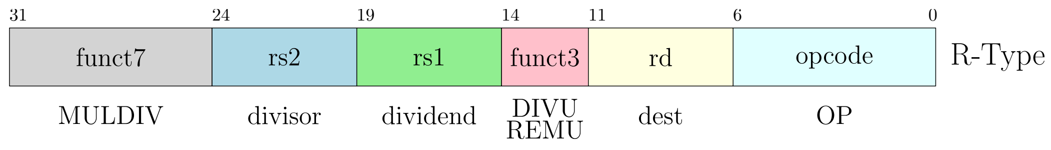


Figure 12: RISC-V DIVU and REMU instruction format

MFHI and MFLO *Move From HI Register* and *Move From LO Register* are MIPS instructions required to move the results of a multiplication or division from the `HI` and `LO` registers to general-purpose registers. Since neither are available or used in RISC-V, these instructions are obsolete.

SYSCALL In MIPS, the *System Call* instruction is a special form of the R-type. Its purpose is to request the execution of a system call that is specified through storing the desired system call number in a special register, namely `v0`, beforehand. The required system call parameters must be stored in the registers `a0 - a3` in advance of the instruction.

In RISC-V this instruction is called `ECALL` (environment call) but fulfills the same purpose. System call numbers must be stored in register `a7` before invocation.

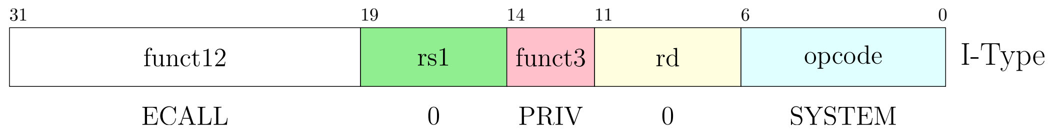


Figure 13: RISC-V ECALL instruction format

Summary To conclude the observations made in this Section, Table 3 provides an overview of all MIPS instructions that are emitted by Selfie's `starc` and their corresponding RISC-V substitutions.

MIPS	**RISC-V**	**Notes**
ADDIU	ADDI	no overflow checks available
ADDU	ADD	no overflow checks available
SUBU	SUB	no overflow checks available
SLT	SLT	
LW	LOAD	
SW	STORE	
JAL	JAL	
J	JAL	equals JAL with rd = reg0
JR	JALR	equals JALR with rd = reg0
BEQ	BEQ	
BNE	BNE	
SYSCALL	ECALL	
MULTU	MUL	result: lower 32 bits of overall result
DIVU	DIVU REMU	result: quotient of division result: remainder of division
MFHI	-	not necessary/available
MFLO	-	not necessary/available
NOP	-	not necessary

Table 3: Summary of MIPS instructions used by Selfie and their respective replacements

3 Porting Process

In Section 2, all major differences between the existing MIPS version and the future RISC-V version of Selfie have been discussed, including register usage, instruction formats and necessary instructions. The following section now describes the details of the changes needed to be made to Selfie's compiler, emulator and hypvervisor to fully support the RISC-V ISA according to the specification. Some parts of this process are straightfoward consequences of the different architectures as mentioned before, other parts included changes that were not as obvious and needed additional care to be executed successfully. The source code of Selfie's RISC-V version can be found at [9].

3.1 Adapting Registers

The first step in the process of porting Selfie to RISC-V was adapting the general-purpose registers' names and usages and the corresponding debugging strings, especially for the human-readable assembly output.

This is a task of little complexity because although registers dedicated to the same purpose have different numbers across both ISAs, all components of Selfie use constants to address individual registers. These constants are offsets in a block of allocated memory dedicated to the GRPs. Therefore, changing the names of these constants automatically means Selfie is using the right registers, as long as their names and purposes still fit. For example, the registers for the global, stack and frame pointers were declared and used as follows:

```
        int REG_GP = 28;
        int REG_SP = 29;
        int REG_FP = 30;

        // ...

        stack_pointer = *(registers + REG_SP);
        global_pointer = *(registers + REG_SP);
        frame_pointer = *(registers + REG_FP);
```

Listing 1: Register declarations in the MIPS version of Selfie

Now, the usage of these registers stays exactly the same, only their constant values changed (the global pointer is register number three now, for instance).

For some registers, their usage changed as well. This is the case for the MIPS register v0 which is used to store system call numbers and return values, and v1, which is an addtional register for storing return values. The former has two different replacement registers, depending on the function. If it was used to store a system call number, it is replaced by a7, for all other purposes the replacement is a0. The latter is replaced by a1, which is the second return value register in RISC-V.

Doing this introduced a bug that was difficult to find. The only instance where v1 originally appeared was in order to copy the value of register v0 so it could later be reused. Incidentally, a1 is also used to store system call arguments, which led to a race condition where in some cases, Selfie would not self-compile normally. This could be avoided by choosing one of the unused saved temporary registers (s1) instead of v1 respectively a1.

Lastly, the usage of temporary registers had to be adapted. Starc only uses the first eight temporary registers of MIPS which all have consecutive register numbers. In RISC-V, the available seven temporaries are split into two consecutive parts. Selfie implements several procedures for managing temporary registers, like getting the next "free" and the previously and currently used register (Listing 2).

```
int currentTemporary() {
  if (allocatedTemporaries > 0) {
    return *(temporary_registers + (allocatedTemporaries - 1));
  } else {
    syntaxErrorMessage((int*) "illegal register access");
    exit(-1);
  }
}
```

Listing 2: An example where consecutive register numbers are required

These procedures need registers with consecutive numbers because they count up and down. To achieve such a thing with RISC-V registers, the code shown in Listing 3 was added.

```
void initRegister() {
  temporary_registers =
      malloc(maxNumberOfTemporaries * SIZEOFINT);

  *(temporary_registers + 0) = REG_T0;
  *(temporary_registers + 1) = REG_T1;
  *(temporary_registers + 2) = REG_T2;
  *(temporary_registers + 3) = REG_T3;
  *(temporary_registers + 4) = REG_T4;
  *(temporary_registers + 5) = REG_T5;
  *(temporary_registers + 6) = REG_T6;
}
```

Listing 3: Putting all temporary register numbers into a block of memory so they can be used as if the were consecutive numbers

After this first step was completed, the `make test` target of Selfie's Makefile (which includes self-compilation, self-execution and self-hosting tests) still finished execution successfully, meaning that no existing functionality had been severely broken by the changes.

3.2 Removal of Unnecessary Instructions

Although the next step is not the most important one, it helped making the rest of the porting process easier. In Section 2.3, we already observed that `NOP` instructions used by Selfie were no longer needed in the RISC-V version due to the jumps and branches not supporting visible delay slots. Removing instances of this and the two other obsolete instructions `MFLO` and `MFHI` simplified the next steps by cleaning up code that was no longer needed.

Another reason why this was chosen to be the second step is that all other steps would break Selfie's functionality temporarily, so postponing the removal would have increased the sources of errors later.

This measure included not only the removal of corresponding opcodes, debug information tied to the instructions, and emulator procedures necessary to execute correct operations, but of course also the postitions in the compiler where the instructions were emitted. Another important aspect are the changes made to instructions which appeared together with `NOP`s. Branch offsets as well as link addresses of jumps had to be recalculated to compensate for the missing instructions.

3.3 Adapting Emulator Functions and Sign-Extension

After the easier parts of the porting process had been finished successfully, the emulator was the next target to be adapted. Here, the actual execution of instructions takes place, and altough some RISC-V instructions need to be handled exactly like their existing MIPS equivalents by the emulator, all instructions had to be adapted in some way. The reasons for this are the differing immediate lengths, field names (such as `rs, rt` versus `r1, r2`) and other naming issues. Also, the execution procedures for some previously not supported instructions had to be added at this point (e. g. for `REMU`).

From here on, Selfie's functionality could not be tested to be correct anymore right until the end of the process, increasing the possibility of creating bugs that could be hard to identify and resolve.

A problem that was encountered during this adaption phase involves the handling of immediates. In MIPS, immediates have a fixed length of 16 bits in contrast to RISC-V where they can be 12 or 20 bits long. However, depending on the instruction an immediate of 12 bit can correspond to a 12 or 13-bit integer, which is also true for 20 and 21-bit immediates. This required a dynamic approach to sign-extending these values after decoding them, which was done as shown in Listing 4.

```
int signExtend(int immediate, int n) {
  // sign-extend from "n"-bit to 32-bit two's complement
  if (immediate < twoToThePowerOf(n-1))
     return immediate;
   else
    return immediate - twoToThePowerOf(n);
}
```

Listing 4: Sign-extension for variable-length immediates

Similarly, before an immediate value can be encoded properly, it must be sign-compressed. This means it is converted from 32-bit to n-bit two's complement (where n is the number of bits in the immediate value) in order to preserve the sign in the encoding (see Listing 5).

```
int signCompress(int immediate, int n) {
  // sign-compress from 32-bit to n-bit two's complement
  if (immediate < 0)
```

```
    return immediate + twoToThePowerOf(n);
  else
    return immediate;
}
```

Listing 5: Sign compression for variable length immedates

3.4 Encoding and Decoding of Instruction Formats

Now that the sign-extension and procedures for executing the individual instructions in the emulator had been adapted, Selfie's interface containing the encoders and decoders of instructions needed to be modified. This step was chosen to be next because the substition of the MIPS instructions by RISC-V instructions would be easier after the new instruction formats were available in code.

Firstly, the interface's encoding procedures were adapted and added to. There is one of these procedures for every instruction format that is supported, which builds a 32-bit instruction out of all individual fields required for this particular instruction format by shifting and adding in the according values.

Such an encoding procedure for the RISC-V S-type would look like described in Listing 6. Lines 1 - 5 show a schematic depiction of the S-type. The immediate value that will be encoded is a signed integer which is sign-compressed in line 10. Because the S-type needs the immediate to be split into two parts, it is shifted to extract the bits 11 to 5, labeled `imm1`, and shifted to extract the remaining bits 4 to 0, labeled `imm2`. Lines 15 to 17 show the last step, where all the individual parts of the instruction are shifted and added accordingly to form the complete instruction.

```
1  //       7          5        5        3         5       7
2  // +----------+--------+--------+---------+--------+----+
3  // |    imm1    |   rs2   |   rs1   | funct3 | imm2   | op |
4  // +----------+--------+--------+---------+--------+----+
5  //   imm[11:5]                              imm[4:0]
6
7  int encodeSFormat(int immediate, int rs2, int rs1, int funct3,
       int opcode) {
8    int imm1;
9    int imm2;
10
11   immediate = signCompress(immediate, 12);
```

```

  imm1 = rightShift(leftShift(immediate, 20), 25);
  imm2 = rightShift(leftShift(immediate, 27), 27);

  return leftShift(leftShift(leftShift(leftShift(leftShift(
      imm1, 5) + rs2, 5) + rs1, 3) + funct3, 5) + imm2, 7)
      + opcode;
}
```

Listing 6: The encoding procedure used to build a S-type instruction

Selfie's decoding procedures look a bit different than their counterparts because they rely on other procedures to extract the individual fields of an instruction. This is done to make the code more readable and reusable, since other decoder procedures need to extract the same fields.

Listing 7 shows the decoder for S-type instructions. Fields that are not present in this format, like `funct7` and `rd`, are set to 0 to avoid mistakes. The values of other fields are extracted by procedures like `getRS2` and `getImmediateSFormat`. To obtain the value of a field, the original instruction is shifted so all other fields are shifted "away" or out of the 32-bit space to clear these bits. An example can be seen in Listing 8, where actually two fields are extracted and combined to form an S-type immediate.

```
void decodeSFormat() {
  funct7    = 0;
  rs2       = getRS2(instruction_reg);
  rs1       = getRS1(instruction_reg);
  funct3    = getFunct3(instruction_reg);
  rd        = 0;
  immediate = getImmediateSFormat(instruction_reg);
}
```

Listing 7: The decoding procedure used to break apart an S-type instruction

```
int getImmediateSFormat(int instruction) {
  int imm1;
  int imm2;

  imm1 = rightShift(instruction, 25);
  imm2 = rightShift(leftShift(instruction, 20), 27);

  return leftShift(imm1, 5) + imm2;
}
```

Listing 8: The extraction of an S-immediate

In addition to the individual decoding procedures, there is a central decoder which extracts the opcode and then decides which format the binary instruction is in and which procedure must decode it. After the decoding has taken place, a special execute procedure has enough information (opcode and sometimes function codes) to decide which RISC-V instruction respectively which of the emulator procedures shall be executed. To be able to do this, opcodes and function codes must be declared according to the specification.

3.5 Handling of Integers

A task that became aparent while starting to substitute MIPS instructions with RISC-V instructions in `starc` is again related to the differing immediate lengths in both architectures. When `starc` parses an integer, it must emit several instructions to load it into a register, since immediates can only be 16 bits long in MIPS. This means that for bigger integers, their value must be split into parts which are loaded separately, then reassembled inside of registers. Since the *Add Immediate* instruction in RISC-V has fewer bits for immediates, the procedure loading integers had to be adapted. Before, integers had to be split into three parts at most if they used all bits except for the sign bit. The first and second part could each have a length of 14 bits, the third part were the remaining three bits.

This is because these bits needed to be manually shifted with instructions, meaning the first 14 bits had to be multiplied by 2^{14} to be shifted 14 bits to the left. Then, the next 14 bits could be added to the number which could again be shifted by 14 bits, and lastly the least significant 3 bits could be added. The restriction of 14 bits was due to the capacity of MIPS immediates: A shift by 15 bits would have meant that a 2^{15} number would have to be loaded into a register to perform the multiplication with the number to be shifted, but 2^{15} is already a number with 16 bits which would have been sign extended and therefore would have caused a wrong solution.

Since RISC-V immediates offer only 12 bits to be used, the maximum number of bits per part decreased to ten, making loading a number a lot more complex in terms of the number of instructions required. After the adaption, an integer using all 31 bits needs to be separated into four parts.

3.6 Full Substitution and Fixup of Instructions

At this point, the encoding and decoding of the new instruction formats was fully supported, but Selfie's components could not be tested to be working correctly since `starc` still emitted instructions formatted to fit MIPS standards. This means that now, every single instruction had to be revisited and rewritten according to the substitution Table 3. This is a task that is mostly straightforward but involves lots of manual editing.

However, when dealing with jump and branch instructions, additional care had to be taken. One example where the use of RISC-V instructions differs from that of MIPS instructions is when unconditional branches are used. In MIPS, this is simply done by checking if zero equals zero, then branching to the desired destination. In RISC-V, a jump should be used instead.

Another difficulty lies within the process of fixing up jump and branch instructions where the destintion address is not known at the time when the instruction is generated. This happens either when the condition of a loop or if-else statement evaluates to false or when procedure calls are parsed earlier than the definition of the procedure in question. In these cases, the proper branch or jump address is only known when the end of the loop or if-else statement or the procedure definition has been reached by the parser. At this point, previously emitted instructions which do not contain the right destination address have to be extracted from the binary, modified and written back. If multiple calls to the same procedure have been parsed before its definition, a so-called fixup chain has to be built in order to remember all locations in the binary that have to be revisited once the correct address becomes available.

In `starc`'s MIPS version, branch and jump fixups have been dealt with in separate procedures. This is because branch instructions need to use `pc`-relative offsets to their targets while jump instructions need `pc`-region offsets. However, the latter has been simplified to use only absolute addresses since this is enough to target binaries even bigger than Selfie itself (see Listing 9).

```
void fixup_relative(int fromAddress) {
  int instruction;

  instruction = loadBinary(fromAddress);

  storeBinary(fromAddress,
    encodeIFormat(getOpcode(instruction),
      getRS(instruction),
 9       getRT(instruction),
10       (binaryLength - fromAddress - WORDSIZE) / WORDSIZE));
11 }
12
13 void fixup_absolute(int fromAddress, int toAddress) {
14   storeBinary(fromAddress,
15     encodeJFormat(getOpcode(loadBinary(fromAddress)),
16         toAddress / WORDSIZE));
17 }
```

Listing 9: The MIPS version of procedures responsible for fixing up branch and jump instructions

In RISC-V, on the other hand, both branches and jumps use `pc`-relative offsets. This means that the previous "absolute" fixup for jumps is obsolete. Instead, a very similar approach to the relative fixup of branches can be used, and the two separate procedures can be combined into one (Listing 10). The division by `WORDSIZE` as seen in Listing 9 (lines 10 and 16) is not necessary anymore since offsets are not shifted in RISC-V.

```
void fixup(int fromAddress, int toAddress) {
  int instruction;
  int currentOp;

  instruction = loadBinary(fromAddress);
  currentOp = getOpcode(instruction);

  if (currentOp == OP_BRANCH) {
    storeBinary(fromAddress,
    encodeSBFormat((toAddress - fromAddress),
        getRS1(instruction),
        getRS2(instruction),
        getFunct3(instruction),
        currentOp));
  } else if (currentOp == OP_JAL) {
    storeBinary(fromAddress,
        encodeUJFormat((toAddress - fromAddress), getRD(
            instruction), currentOp));
  } else
    // error handling
}
```

Listing 10: The fixup procedures in Selfie's RISC-V version

3.7 Renaming

Finally, after ensuring that all Selfie functionalities are behaving as expected by running `make test` and executing additional tests, the last thing to be done was to rename parts of the software that had a MIPS flavor to them. For instance, the MIPS emulator `mipster` was renamed to `rocstar`, the supported subset of MIPS (MIPSter) was renamed to RISCY, and some internal flags and names for hypercalls were changed.

3.8 Possible Optimizations

There is still potential to make Selfie's components more realistic, efficient and the code more readable. In particular, the calling convention used by RISC-V is not fully supported yet. The registers dedicated for saving temporaries are not used at all, and `starc` does not make use of registers for passing arguments to procedures, but rather pushes them all onto the stack.

Since one goal behind Selfie is to keep it minimal, another optimization could be to replace instructions with other already supported ones, so the instruction set becomes even smaller. For example, this could be achieved by replacing *Branch on Not Equal* instructions with *Branch on Equal* instructions, making the former obsolete. This could possibly also be done for other instructions, although removing too many instructions could make it harder to understand the assembly code produced by Selfie.

Another minor opportunity for optimization is in the assembly output that Selfie's emulator `rocstar` generates. This output can be used for debugging purposes, so the more information it contains, the easier this is. For some instructions, this output does not match up with how it would be generated by other utilities like `objdump` of the official RISC-V toolchain, mostly in favor of printing additional information. To make `rocstar`'s assembly output more realistic, this information could be removed.

4 Performance

With two versions of Selfie now available, it seems natural to try to compare them against each other to further analyze possible benefits of the port.

Firstly, three different C* source files where compiled with both the MIPS and RISC-V version of Selfie's `starc` compiler, and their respective assembly outputs were generated. One of these files is the Selfie source file itself, the other two implement a recursive procedure (`recursion.c`) and a simple "Hello World" output (`hello-world.c`). These programs are both described in [2, chapter 3].

	`selfie.c`	`recursion.c`	`hello-world.c`
Number of instructions in assembly file			
MIPS	28,779	206	145
RISC-V	26,360	180	126
Reduction	8.40 %	12.62 %	13.10 %
Binary size			
MIPS	121,660 B	828 B	600 B
RISC-V	112,016 B	724 B	524 B
Reduction	7.92 %	12.56 %	12.67 %

Table 4: Total amount of instructions emitted by `starc` for different source codes

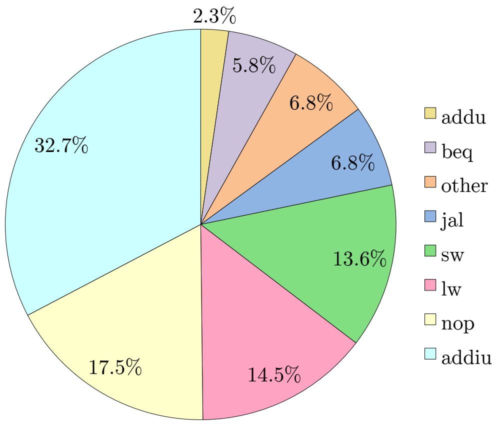
Table 4 shows the number of instructions that the respective assembly files contained and the sizes of the different binaries. For all three source files, the RISC-V binaries and assembly files are smaller than their MIPS counterparts. Depending on the source file size, the differences vary - for files containing little code, the reduction of the number of instructions emitted is higher than for those with more code. The reduction is mostly caused by the removal of `NOP` instructions, but more instructions are needed for loading integers (see Section 3.5) which are more likely to occur in bigger source files. Therefore, the overall reduction of instructions is bigger for small programs.

As a next step, the distribution of individual instructions occuring in assembly files produced by both Selfie versions are analyzed. For this purpose, `selfie.c` was used again. The results are shown in Figure 14.

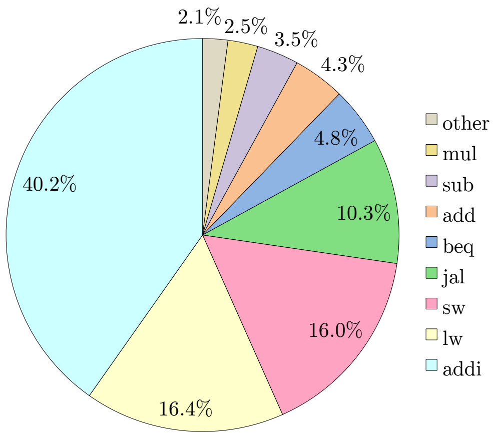
One can observe that the amount of `ADDI` instructions is around a third of the total MIPS instructions emitted for `selfie.c`. Together with NOP instructions, they make up around half of the assembly file. In the RISC-V version more

`ADDI`s are emitted, which is because of the more difficult loading of integers as mentioned before.

The other instructions are distributed roughly the same way in both versions, with the individual RISC-V instructions having slightly larger percentages than their MIPS counterparts because the overall amount of instructions is smaller. The only exceptions are `BEQ` and `JAL` instructions, where the former appears less and the latter more often than the MIPS assembly file. This is because there are no individual jump instructions anymore as they are replaced by `JAL`, as well as the unconditional branches were.



(a) MIPS instruction frequencies



(b) RISC-V instruction frequencies

Figure 14: Relative frequencies of individual instructions in a Selfie assembly file

Lastly, the absolute frequency of instructions is investigated in Figure 15 using one of the smaller source code files. In accordance with the previous findings, most of the instructions are used as often in the RISC-V as in the MIPS assembly file, with the exception of ADDI, JAL, BEQ and MUL. The latter is used in combination with ADDI for loading integers, so both are used more often in RISC-V, whereas NOP and MFLO are not used at all.

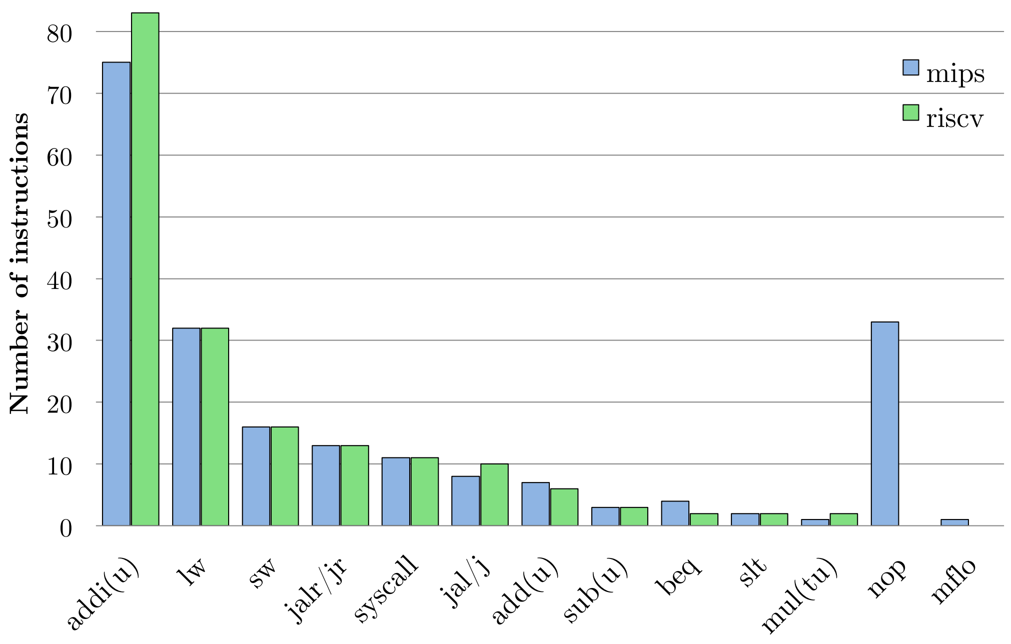


Figure 15: Amount of MIPS and RISC-V instructions emitted by rocstar for recursion.c

5 Conclusion and Related Work

A new version of Selfie, an educational platform for teaching the constructions of compilers, operating systems and virtual machine monitors, has been successfully developed. It utilizes a subset of the open-source ISA RISC-V, which provided opportunities to minimize the amount of instructions that need to be supported and to demonstrate different encoding and decoding schemes from the original MIPS version.

In addition to the educational benefit of using a state-of-the-art ISA, the changes made to Selfie's code resulted in a reduction of the binary and the assembly code size, making binaries generated by `starc` smaller, easier to debug and the assembly code easier to comprehend.

Based on the work described in this thesis, there have been made further changes to Selfie's compiler `starc` and emulator `rocstar` in order to become more realistic. The specific aim of this follow-up project was to modify Selfie in such a way that binaries generated by `starc` are able to be executed by a native RISC-V emulator called `Spike`, together with a minimal kernel (proxy kernel or `pk`). Additionally, a `starc` binary containing Selfie being executed on *Spike* should in turn be able to be used as compiler, emulator and hypervisor with full self-referentiality.

The steps taken to achieve this included replacing Selfie's own raw binary format by a minimal version of the ELF binary format and adapting the native toolchain and various parts of Selfie. The resulting Selfie platform now offers the benefits of potential execution on real RISC-V hardware chips and of creating binaries that are more easily debuggable by using GNU's `binutils` [10].

A Appendix

A.1 References

[1] Selfie. http://selfie.cs.uni-salzburg.at/, 2015-2017.

[2] Christoph Kirsch. *Selfie: Computer Science for Everyone*. Leanpub, 2017.

[3] MIPS Architectures. Imagination Technologies Limited. https://www.imgtec.com/mips/architectures, 2017.

[4] The RISC-V Foundation. https://riscv.org/risc-v-foundation/, 2017.

[5] RISC-V chosen as Best Technology of 2016. https://riscv.org/2017/01/risc-v-chosen-best-technology-2016/, 2017.

[6] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanovic. *The RISC-V Instruction Set Manual Volume I: User-Level ISA, Version 2.1*. CS Division, EECS Department, University of California, Berkeley, May 2016.

[7] MIPS Technologies, Inc., 1225 Charleston Road, Mountain View, CA 94043-1353. *MIPS32 Architecture For Programmers Volume II: The MIPS32 Instruction Set*, January 2009.

[8] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., 5th edition, 2013.

[9] Selfie's RISC-V version on GitHub. https://github.com/cksystemsteaching/selfie/tree/riscv, 2015-2017.

[10] Christian Barthel. *Porting Selfie to RISC-V: Native Toolchain Support*. Bachelor Thesis, Department of Computer Sciences, University of Salzburg, Austria, 2017.

A.2 List of Figures

A.3 List of Listings

A.4 List of Tables