Bachelor Thesis

Selfie – RISC-V to x86-64 binary translation

Alexander Kollert

29 February 2020

Advisor: Christoph M. Kirsch

Department of Computer Sciences

University of Salzburg Austria

Contents

Abstract *Selfie* is a project of the Systems Group of the Paris-Lodron University in Salzburg which I was part of during the creation of this work. It implements *starc* that compiles a subset of the C Language into a subset of RISC-V machine code and *mipster* which executes a binary generated by *starc*. In this work I present an implementation of a binary translator that statically translates RISC-V machine code produced by starc into x86-64 machine code. The requirements are that the binary translator should be able to translate *Selfie* itself and the usage of a minimal subset of x86-64 instructions. As it is in the nature of *Selfie* the implementation should also be as minimal as possible. There are a few challenges to overcome to make this work. First the x86-64 ISA (Instruction Set Architecture) is an variable length instruction set which makes calculation of target addresses of jump instructions more difficult. Further the x86-64 instruction set mostly consists of two-operator instructions and even some single-operator instructions in contrast to three-operator instructions in RISC-V. And last but not least we will see that for some RISC-V instructions it is not possible to make an translation entirely without context. The result is a binary translator that is able to generate an x86-64 selfie binary with functioning self-compilation in x86-64. The difference with the translated binary is that there is no longer a need to link against a library, since it uses its own implemantations for the library routines.

1 List of Features and Limitations

This is an overview of features and limitations of the implementation of the binary translator. Selfie itself has a number of limitations which are not discussed here.

1.1 Features

- self-compilation in x86 fixed point
- additional section headers allows easy reading of machine code with `objdump`

1.2 Limitations

- Selfie allows an maximum of seven temporaries. For shortage of registers in x86 the binary translator can only translate correctly when maximum five temporaries are used.
- The binary translator has a crude way of recognizing global pointer initialization in Selfies machine code. Bigger changes to this particular code can result in incorrect code translation for global pointer initialization.

2 Introduction

2.1 Problem Description

Binary Translators generate binaries for a target platform dynamically or statically. The Binary Translator in this work is implemented as an static Translator, because the implementation is easier. The binaries compiled by *Selfie* always rely on some emulator (*QEMU*,*mipster*,*spike*) to be executed. Translating RISC-V binaries into x86-64 binaries makes it possible to execute them without the need for emulation, which will give it a performance boost. Also it would be possible to analyze the behavior of binaries more directly on the system.
Examples of x86-64 assembly code are in AT&T syntax.

2.2 Challenges

There are a few challenges to overcome for this project. The translator needs to generate variable length machine code. There are fewer registers available on x86-64 and are used differently. The x86-64 ISA encodes two operators and sometimes even one operator in instructions instead of three. That means the translator needs extra instructions to move values around registers just to make sure the value ends up in the correct register. Also the translator should use minimal context while translating.

2.3 RISC-U

Selfie uses a subset of RISC-V instructions for its code generation. In this section all 14 instructions are listed and shortly explained. A more detailed explanation is found in section 3.5.

`lui rd,imm`	Loads an immediate value shifted by 12 bits to the left into register `rd`
`add rd,rs1,rs2`	Adds `rs1` and `rs2` together and puts the result in `rd`
`addi rd,rs1,imm`	Adds `rs1` and `imm` together and puts the result in `rd`
`sub rd,rs1,rs2`	Subtracts `rs1` from `rs2` and stores result into `rd`
`mul rd,rs1,rs2`	Multiplies `rs1` and `rs2` and puts the result in `rd`
`divu rd,rs1,rs2`	Divides `rs1` by `rs2` and puts the **quotient** in `rd`
`remu rd,rs1,rs2`	Divides `rs1` by `rs2` and puts the **remainder** in `rd`
`sltu rd,rs1,rs2`	Sets `rd` to value 1 if `rs1` is less than `rs2`
`ld rd,offset(rs1)`	Loads value from memory address `rs1 + offset` into `rd`
`sd rs2,offset(rs1)`	Stores `rs2` to memory address `rs1 + offset`. This command does not modify any registers
`beq rd,rs2,imm`	Jumps relative `imm` instructions if `rd` is equal to `rs2`
`jal rd,imm`	Writes address of following instruction into `rd` and jumps relative `imm` instructions
`jalr rd,offset(rs1)`	Writes address of following instruction into `rd` and jumps to address `rs1 + offset`
`ecall`	Calls an operating system routine which can be selected with register arguments

3 Implementation

3.1 General

This implementation uses Selfie's existing functions and structures as much as possible. Selfies emulator *mipster* loads a binary into its memory, fetches the instruction where the program counter points to, decodes and executes it. The execution of that instruction will set the program counter to a new address and repeat the process. The translator works in a very similar way. One difference is that calling `execute()` in translating mode calls an function that translates the last decoded RISC-V instruction, instead of the function that emulates the instruction. That means for every emulated instruction in *mipster* a translating function must be implemented. Another difference is that translating functions always increment the program counter by 4 bytes, except in a few cases where the binary translator skips instructions. These cases are discussed in detail later. In that way the RISC-V binary is read from first instruction to last in sequence while immediately emitting x86-64 machine code. After this step it is needed to go over the freshly generated x86-64 code and modify all jump and branch offsets. Why this needs to happen is explained in detail in the section Addresses. Then the data segment and the ELF header of the RISC-V binary is copied over. At last a few ELF header fields need to be modified. Now translation is finished and everything can be written into a file.

3.2 x86-64 Registers

RISC-V has 32 Registers of which one is a zero register in contrast to 16 Registers in x86-64. This means there is no one-to-one mapping possible. The simplest solution is to just map multiple RISC-V Registers to one x86-64 Register and to hope that this many registers are never used. This is also the solution used in this project. It turns out that this is enough to translate *Selfie* and so the goal is met. However this implies that not every binary generated by *starc* can be successfully translated into an x86-64 binary. The following mapping of registers is used.

sp $\rightarrow$ rsp
fp $\rightarrow$ rbp
gp $\rightarrow$ rbx
ra $\rightarrow$ r15
t0 $\rightarrow$ rcx
t1,a3 $\rightarrow$ r10
t2 $\rightarrow$ r11
t3 $\rightarrow$ r13
t4 $\rightarrow$ r14
a0 $\rightarrow$ rdi
a1 $\rightarrow$ rsi
a2 $\rightarrow$ rdx
a4 $\rightarrow$ r8
a5 $\rightarrow$ r9
a7 $\rightarrow$ rax

The register `ra` is generally used to store the return address before it is pushed onto the stack. Normally on x86 this is accomplished with one instruction that is `enter`, but this would require the binary translator to be aware of the context the instruction is in. To emulate this behavior the register `r15` is used. The regsiters `a0..a5` are used to hold syscall arguments and `a7` holds the syscall number and are mapped to the corresponding registers on x86.

3.3 x86-64 Instruction Format

The x86-64 instruction set has variable length instructions which can range from 1 byte to 15 bytes. Opcodes can be 1 to 2 bytes long. The first byte of an two byte opcode is always 0x0f. Some single operand instructions encode the register operand in the opcode byte itself. Otherwise operands are encoded in the so called MODRM byte which would follow after the opcode. The MODRM byte sets the addressing mode, the operands and if required the opcode extension. The SIB (Scaled Indexed Byte) and the Displacement field is not used in this project. There are a few pfixes with different meanings, but for this work there is only one important prefix. To get access to a 64 bit register or if more than three bits are needed to encode a register the instruction needs the REX prefix.

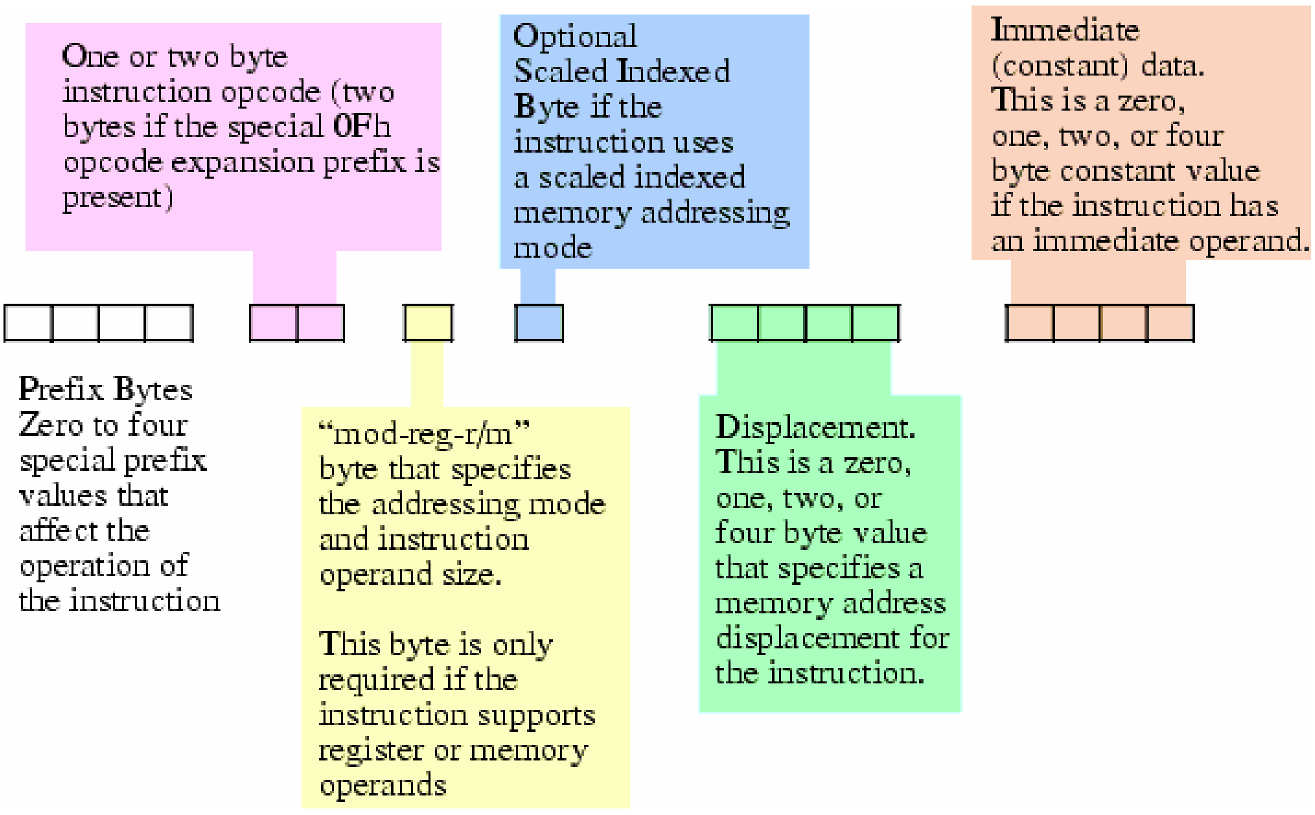
One or two byte instruction opcode (two bytes if the special 0Fh opcode expansion prefix is present)
Optional Scaled Indexed Byte if the instruction uses a scaled indexed memory addressing mode
Immediate (constant) data. This is a zero, one, two, or four byte constant value if the instruction has an immediate operand.
Prefix Bytes Zero to four special prefix values that affect the operation of the instruction
"mod-reg-r/m" byte that specifies the addressing mode and instruction operand size.
This byte is only required if the instruction supports register or memory operands
Displacement. This is a zero, one, two, or four byte value that specifies a memory address displacement for the instruction.

Figure 1: [1]

7	6	5	4	3	2	1	0
0	1	0	0	W	R	X	B

Figure 2: REX Prefix byte

The upper half of the REX prefix has a fixed bit pattern. The other bits have special meanings and need to be set correctly to encode an x86-64 instruction. The meanings of the bit fields are the following:

W → if set register size of 64 bits is used, otherwise instruction dependent default operand size is used
R → extension to the MODRM.reg field
X → extension to SIB.index field (not relevant for this project)
B → extension to MODRM.rm field

For the sixteen x86-64 registers 4 bits would be needed to encode a register. But the MODRM byte only has space for three bits per operand, so that these extension bits in the REX prefix are used. The function `x86GetPrefix(uint64_t operand1, uint64_t operand2, uint64_t wide)` computes the REX prefix based on the provided operands. The `wide` parameter also needs to be provided which sets the W bitfield of the MODRM and as of writing there was no way to avoid this parameter. The caller of this function needs to know what the default operand size of an instruction is which is not desirable.

3.4 System Calls

The binary translator needs to recognize when a system call value is getting set since the values are different for the Linux operating system running on the x86 platform. The registers that Selfie's *starc* uses for setting RISC-V system calls are only used for that matter. Hence every time a value is moved to one of this register set `a0..a5,a7` it must be a system call value. The binary translator just needs to map that value to an x86 system call. The following table shows the mapping of system calls:

System Call	RISC-V	x86-64
EXIT	93	60
READ	63	0
WRITE	64	1
OPENAT	56	257
BRK	214	12

3.5 Selfie's RISC-U Instructions

3.5.1 ADD

Case 1: All three register operands are different.

```
                        mov %r10,%rcx
add $t0,$t1,$t2 →
                        add %r11,%rcx
```

The x86 add instruction adds together two registers and writes the result into the second register. Before the addition one of the operands has to be moved into the target register.

Case 2: Target register and one of the operands are the same.

```
add $t0,$t0,$t1 →  add %r10,%rcx
```

In this case there is no need to move around operands before addition.

Case 3: One of the operands is the zero register.

```
add $t0,$zero,$t1 →  mov %r10,%rcx
```

This is just moving a the value of a register into another. RISC-V has no explicit instruction for this.

3.5.2 ADDI

Case 1: The register operand is the zero register.

```
addi $t0,$zero,1234 →  movabs 1234,%rcx
```

Very similar as in the ADD instruction this is loading a value into the register. The x86-64 `movabs` instruction loads a 64 bit immediate value into a register.

Case 2:

```
                       mov %r10,%rcx
addi $t0,$t1,1234 →
                       add 1234,%rcx
```

Case 3: This is a special case. On RISC-V there are no explicit instructions to push onto the stack or pop from the stack. The push operation for instance is achieved by incrementing the stack pointer by 8 and then storing the desired value to this new address. On the x86 platform there are the `push` and `pop` instructions that do this job. It is not possible to translate the RISC-V sequence directly because loading from and storing to register `rsp` on x86 is only possible with these two instructions. In other words, the x86 ISA does not allow encoding an instruction like this: `mov %rcx,0(%rsp)`.

```
addi $sp,$sp,-8
                 → push %rcx
  sd $t0,0($sp)
```

```
   ld $t0,0($sp)
                 → pop %rcx
 addi $sp,$sp,8
```

This means the binary translator needs to be aware of this sequence of instructions. Every time the binary translator sees `addi $sp,$sp,-8` it does not translate immediately but checks if the next instruction is `sd $t0,0($sp)`, otherwise translation is performed as usual. The same procedure is followed when a potential `pop` instruction is translated.

3.5.3 SUB

Case1 1: The first operand is the zero register.

```
                        mov %r10,%rcx
sub $t0,$zero,$t1 →
                        neg %rcx
```

If something is subtracted from zero than it is an negation. The binary translator will use the `neg` instruction for translation.

Case 2: The second operator of the subtraction is the same as the target register.

```
                      neg %rcx
sub $t0,$t1,$t0 →
                      add %r10,%rcx
```

Subtracting two numbers from each other is the same as a addition with the negation of the second operator.

Case 3: The general case.

```
                      mov %r10,%rcx
sub $t0,$t1,$t2 →
                      sub %r11,%rcx
```

3.5.4 MUL

```
                      mov %r10,%rcx
mul $t0,$t1,$t2 →
                      mul %r11,%rcx
```

For multiplication the binary translator does not differentiate between cases.

3.5.5 DIVU/REMU

```
                      push %rdx
                      push %rax
                      movabs 0,%rdx
                      mov %r10,%rax
divu $t0,$t1,$t2 →    div %r11
                      mov %rax,%rcx
                      pop %rax
                      pop %rdx
```

The division instruction on x86 is very different than the other instruction discussed in this work. First of all only one operator is given by this instruction, namely the divisor. The dividend is hardwired to the `rax` register. And then the computed quotient is placed into `rax`. The instruction also simultaneously computes the remainder and places the result into `rdx`. For the translation this means saving the two registers `rax, rdx` to the stack since they possibly holding needed values. Moving the dividend to `rax` and now the division can be executed. Now the result is moved to the target register, depending if its `divu` of `remu` the result is in `rax` or `rdx` respectively.

3.5.6 LD

Case 1: Loading from stack pointer address using `$sp` register. As described above for the `ADDI` instruction this will be translated into a `pop` instruction. A slight difference here is that the following instruction could not only deallocate 8 bytes but the whole stack frame. The following example shows this situation:

```
  ld $t0,0($sp)  →  pop %rcx
addi $sp,$sp,16     add 8,%rsp
```

Case 2: Loading from address using any register except `$sp`.

```
ld $t0,0($t1) → mov 0(%r10),%rcx
```

3.5.7 SD

```
sd $t0,0($t1) → mov %rcx,0(%r10)
```

3.5.8 LUI

On RISC-V to set a register to a value that exceeds 12 bits the `lui` instruction is used. It loads the upper 20 bits of a 32 bit word. The `mov` instruction on x86 can move 32 bit values and even 64 bit values to a register. The binary translator just has to shift the immediate value by 12 bits to the left.

```
lui $t0,0xFF → movabs 0xFF000,%rcx
```

Calculating Global Pointer:
Selfie's *starc* generates code where the global pointer initialized to a value that is the memory address right after where the binary ends in memory. Here is an example how this code looks:

```
lui $t0,62
addi $t0,$t0,144
addi $gp,$t0,0
```

Binary translation will certainly yield a binary of different size as the RISC-V binary. This means the global pointer in the generated x86 binary differs from the one in the RISC-V binary. Somehow the binary translator needs to recognize when it is translating the code for the global pointer initialization and use another value for its own global pointer initialization. A very simple approach is to check if after a `lui` instruction the value is eventually moved to `gp`. If that is the case the binary translator only generates a instruction with a dummy value like this:

```
movabs 0xFFFFFFFFFFFFFFFF,%rbx
```

The binary translator saves the exact location of this dummy 8 byte value in memory. After translation is finished the dummy value is overwritten with the now known actual value. And the end result looks something like that:

```
movabs 0x54E73,%rbx
```

3.5.9 SLTU

Here the target register is set to 1 if first operator is less than the second register or zero otherwise. On x86 there exists the `setb` (set below) instruction which sets a register based on the `eflags` register. The `eflags` are not computed by the `setb` instruction. It must be done by another instruction which is the `cmp` instruction.

```
                      cmp %r10,%r11
sltu $t0,$t1,$t2 →
                      setb %cl
```

3.5.10 BEQ

Similar to the `sltu` instruction the `beq` instruction compares two registers and performs a relative jump if they are equal. For x86 there is the `je` instruction for a relative jump based on the `eflags` register. Here too first the comparison has to be made before the conditional jump.

```
                 cmp %r10,%r11
beq $t0,$t1,4 →
                 je 16
```

The offset for the `je` instruction is in bytes and the `beq` offset in four byte words. That explains the different offsets in this example. In reality the binary translator calculates a new value as offset. This is explained more detailed in section 3.7.

3.5.11 JAL

This instruction stores the current program counter + 4 to a register and jumps to a relative offset. Selfie's *starc* uses it not only for function calls but also loops, returns etc. The x86 instruction set offers `call` and `ret` for function calls which handle return addresses. In this project the decision was made to use the `jmpq` instruction instead, because it mirrors more exactly the RISC-V `jal` instruction and more x86 instructions would be needed otherwise. But before the actual jump we need to save the return address. This is accomplished with the `lea` (load effective address) instruction.

```
                  lea 5(%rip),%r15
jal $ra,4 →
                  jmpq 16
```

The return address comes right after the `jmpq` instruction which is 5 bytes long. When the `lea` instruction accesses the program counter it points the following instruction, which in this case is the `jmpq` instruction. Adding 5 bytes offset is necessary to get the address to the instruction after the `jmpq` instruction.

3.5.12 JALR

This instruction jumps to an absolute address that is read from an register. Selfie's *starc* uses this for returning from function calls, hence there is no need to store a return address.

```
jalr $zero,0($ra) →  jmpq %r15
```

3.5.13 ECALL

This instruction can easily just be translated to its x86 eqivalent `syscall`. After the system call on x86 the return value will be in the `rax` register which needs to be moved into the `rdi` register to match RISV-V behavior.

```
          syscall
ecall →
          mov %rax,%rdi
```

3.6 Overview of used x86 instructions

In this section a short overview is given over all discussed x86 instructions so far in this project. Operands are called `rd` for register destination, `md` for memory destination `rs` for register source and `ms` for memory source. An Operand sometimes can be either a register or memory operand.

`add rd/md,rs`	Adds `rd/md` and `rs` together and puts the result in `rd/md`
`addi rd/md,imm`	Adds `rd/md` and `imm` together and puts the result in `rd/md`
`sub rd/md,rs`	Subtracts `rd/md` from `rs` and stores result into `rd/md`
`mul rd,rs/ms`	Multiplies `rd` and `rs/ms` and puts the result in `rd`
`div/neg rs/ms`	Divides `rax` by `rs/ms` and puts the **quotient** in `rax`. Simultaneous the **remainder** is put into `rdx`
`cmp rs1,rs2/ms2`	Compares `[rs1` and `rs2/ms2` and sets the internal status register accordingly
`setb rd/md`	Sets `rd/md` to value 1 if the first operand of the last comparison was below the the second operand
`push rs`	Stores the register `rs` into address the stack pointer `rsp` points to and increase `rsp` by the operand size
`pop rd`	Loads value from memory address `rsp` points to into `rd` and decrease `rsp` by operand size
`mov rd/md,rs`	Moves register `rs` to `rd/md`
`mov rd,rs/ms`	Moves `rs/ms` to `rd`
`mov rd,imm`	Moves `imm` to `rd`
`lea rd,offset(rs)`	Moves `rs + offset` to `rd`
`jmp offset`	Jumps relative by `offset` bytes
`jmp rd/md`	Jumps to absolute address provided in `rd/md`
`je offset`	Jumps relative by `offset` bytes if the last comparison yielded two operands were equal
`syscall`	Calls an operating system routine which can be selected with register arguments
`nop`	No operation

3.7 Addresses

Since the translation of an instruction could yield any arbitrary amount of translated instructions with different sizes (for x86 binaries) any jump or branch offset needs also to be adjusted. Lets consider following RISC-V code and its translation so far:

```
beq $zero,$zero,3        cmp %r13,%r14
sd $a0,-8($gp)           je 3
addi $a0,$t1,0      →    mov %rdi,-8(%rbx)
jalr $zero,0($ra)        mov %r10,%rdi
                         add 0,%rdi
                         jmpq %r15
```

The 3×4 relative offset in the ”`je 3`” instruction is wrong and would throw an `Illegal instruction` error if attempted to execute, because it would jump into the middle of the `add` instruction. The correct offset to jump over the two move's and add instructions here should be 14 bytes. To calculate the correct offset the binary translator needs two things: the offset to the target of a jump in the RISC-V code and the address in the x86 code where exactly this instruction was translated to. The latter piece of information is stored in an list with the size of the number of RISC-V instructions in the binary. For every RISC-V instruction the binary translator sees it makes an entry in that list of the address where it will emit its translation. The position of the RISC-V instruction in the code is the index into the list. For simplicity lets assume the code snippet above is the whole program, then the list for that example would look like this:

Index	Address
0	0x00
1	0x0a
2	0x0e
3	0x18

Now if the binary translator wants to know where the translated code for the ”`jalr $zero,0($ra)`” instruction is, it looks in the list at index 3 because its the fourth instruction in the RISC-V code. The target offset of the jump is stored in the translated instruction itself as it can be seen in the ”`je 3`” instruction,

which is just copied from ''`beq $zero,$zero,3`''. After every RISC-V instruction is processed the binary translator iterates over the addresses in the just generated address list. If a jump instruction is seen in the x86 code at the current address from the list the offset of the jump instruction is read. Now the correct offset can be computed which is the address at the target index of the list subtracted by the current address. Applied to the current example the process would look like this: The binary translator starts looking for any jumps starting at address from index 0 in the address list which is 0x0. The ''`je 3`'' instruction is found. The target address at index 0+3=3 is looked up which is 0x18. Offsets for jumping do not include the instruction itself. The address right after the ''`je 3`'' instruction is 0x0a. In other words the jump goes from address 0x0a to 0x18. To get the relative offset the current address 0x0a is subtracted from target address 0x18 which is 0x0e or 14 in decimal. Now the binary translator overwrites the old value with 14. Keep in mind that `je` expects the offset in bytes whereas the RISC-V counterparts expect offsets in number of instructions.
The implementation reuses the memory where the RISC-V binary is stored for the address list since the translator only needs one pass for translation. If the RISC-V binary is needed in memory after translation it must be loaded again.

4 Results

4.1 General

All the discussed concepts until here were enough to implement a binary translator that can translate Selfie and fully replace the binary compiled by gcc. At this moment of writing the RISC-V binary generated with Selfie of Selfies sourcecode is 193 Kilobytes big. The binary translator generated a binary of 286 Kilobytes, which is an increase of 24 % in size.
On the other side of course an increase in Performance is expected since the translated binary is running natively on the processor instead of emulation. Without binary translation to actually execute Selfie generated code something like this needs to be run: `selfie -l selfie.m -m 2`. With binary translation this situation changes and there is no need for help by an binary generated from another compiler. A simple run of the translated binary is enough to actually run Selfie generated code like this: `selfie.x86`. In this case one layer of emulation is removed and has a huge impact on performance. One test with this setup showed an speedup of more than 200 times. At this point it should be noted that an direct comparison is not possible and really not necessary. The whole idea of an binary translator exactly is to remove the need for emulation.
Since x86 provides less registers, it can hold less variables simultaneous in registers than a program running on the RISC-V architecture. This is an limit for this implementation of an binary translator. In most cases this limit should not be a problem.

4.2 Running the binary translator

The binary translator can be invoked with the `-x86 [`*`outputfile`*`]` option. The binary translator will translate the content that is mapped in Selfies memory space. That means before invoking the binary translator a RISC-V binary must be compiled with the `-c` option or loaded with the `-l` option. For example to compile

Selfies source code and then translate it into a x86 binary the following command needs to be executed: `./selfie -c selfie.c -x86 selfie.x86`. The new binary is saved in the file `selfie.x86`. It might be necessary to give it execution permission. Now the binary can be executed directly on any x86 architecture.

References

[1] http://www.c-jump.com/CIS77/CPU/x86/lecture.html, *c-jump*.

[2] http://ref.x86asm.net/coder64.html.

[3] System V Application Binary Interface, 3rd Edition, The Santa Cruz Operation Inc.

[4] rv8: a high performance RISC-V to x86 binary translator, Michael Clark and Bruce Hoult, 2017