Recycle, Reduce, Reuse - Surveying Instruction Set Architectures

Philipp Erhardt, Henning Stubbe*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: philipp.erhardt@tum.de, stubbe@net.in.tum.de

Abstract—**Code size can play an important role when implementing emulators for an instruction set architecture (ISA), as performance can in part depend on whether cache misses occur frequently. Additionally, implementation complexity is heavily correlated with the complexity of the ISA to emulate. In this paper we take a look at different ISAs and compare their complexity as well as the object code size of an example function. A git repository is provided, allowing reproduction and adaptation for analysis of other functions. While disassembling executables, we also found a possible size optimization in a *RISC-V* compiler.**

Index Terms—**instruction set architectures, code size**

1. Introduction

When creating emulation software, the complexity of the implementation is tightly coupled with that of the emulated ISA. High complexity can increase effort needed for testing and verification. Furthermore, reducing the size of the input machine code can improve performance due to fewer cache misses during interpretation as well as reducing memory, storage and transmission overhead.

Both properties show a tradeoff when designing ISAs: one can either use simple and usually small instructions of which more will be needed to express complex programs, or add more instructions that express the program at a more direct level. Nowadays, there are many ISAs that have taken different design decisions. Some have become symbols for high complexity due to thousands of supported instructions, others aim to provide only a minimal set. Variants for microcontrollers have been introduced to minimize code size, improving usage of limited storage.

This paper aims to describe and compare different ISAs to evaluate their fitness for the implementation of an emulator, with focus being on code size reduction and decreased implementation complexity. Section 2 explains concepts of ISAs and introduces selected ones. Section 3 discusses related work on ISA complexity and usage, Section 4 compares ISAs with regard to their complexity and size.

2. Background

The first part of this section will present common properties used to differentiate ISAs. The second part presents selected ISAs that are compared in Section 4.

2.1. Instruction Set Architecture

The ISA of a processor defines operations for loading, storing and manipulation of data, as well as how these instructions are encoded and executed [1, 5.3 Instruction Set Architecture]. ISAs can be compared by means of their properties, of which selected ones will be presented.

Internal Storage Type The differentiation between stack, accumulator or register machines is fundamental when analyzing ISAs.

In a stack machine, an instruction operates by taking zero or more arguments from the stack and pushing its result to the top of the stack. For most instructions, operands are implicit and need not be specified.

In an accumulator architecture an instruction operates on the accumulator and zero or more explicit arguments, with results being stored in the accumulator.

Instructions of register machines explicitly specify both source and destination operands. In a *register-register machine*, all operands must be registers, except for operands of load and store instructions. These architectures are called *load-store* architectures. In a *register-memory* architecture, most instructions can access memory operands [2, A.2 Classifying Instruction Set Architectures].

Instruction Encoding The instruction encoding of an architecture defines the executable format. The exact binary values of instructions differ widely between architectures. We can however differentiate between fixed-length and variable-length encodings [2, A.7 Encoding an Instruction Set].

Complexity ISAs can be classified by the complexity of their operations. The most common categories are *Reduced Instruction Set Computer* (*RISC*) and *Complex Instruction Set Computer* (*CISC*).

CISC processors are typically able to perform hundreds of instructions of differing complexity with memory operands and many addressing modes available for most. These instructions can take many clock cycles to execute.

In comparison, *RISC* processors include mostly basic instructions, more complex operations must be expressed using them. Many *RISC* architectures aim to execute one instruction per clock cycle [1, p. 91]. Often memory access is only possible via load and store instructions [3, Chapter 3].

Endianness The endianness of a processor defines the byte order when accessing data from memory. With little endian, the most significant byte is stored at the highest address. With big endian byte order, the most significant byte is stored at the lowest address [4, Section 2.4].

Addressing Modes ISAs typically include different ways of accessing operands, either as immediate values, in registers or in memory. Memory addressing modes can express the address as *displacement* relative to a known location like the stack, as *register indirect* with an address already being stored in a register or *indexed* addressing where an offset is added to a base address (for accessing data in arrays), as well as many more addressing modes not mentioned here [2, Section A.3].

2.2. Selected ISAs

The following section briefly describes the ISAs that will be compared in Section 4. These architectures were selected for comparison because they either claim reduced object code size, have a small number of instructions or feature a design that is vastly different from the others.

ARM T32 The *ARM T32* instruction set, previously called *Thumb-2*, is a superset and successor of the *ARM Thumb* instruction set [5] featuring 301 instructions [6, p. 5-13]. Encoded instructions have variable length of either 16 or 32 bits [7]. It is in the family of *RISC* architectures. Since memory access is only possible using load and store instructions, *T32* can be classified as a load-store architecture [8, p. 36]. Registers are 32 bits in size [8, p. 38]. Processors with this architecture can switch between big-endian and little-endian mode for memory access using the `SETEND` instruction [8, p. 7569, F5.1.182].

RISC-V Compressed *RISC-V* is a free and open family of *RISC* ISAs. While any *RISC-V* ISA includes the base integer ISA, it is possible to add optional extensions [9, Section 1.3]. The size of encoded instructions is fixed to 32 bits for the base ISA, extensions can however use a multiple of 16 bits to encode further instructions [9, Section 1.5]. One extension providing 16-bit encodings for common instructions is the standard "C" ("Compressed") extension [9, Chapter 16]. The *RV32I Base Integer Instruction Set* defines 40 instructions [9, p. 31] and is little endian [9, p. 8]. *RV32I* is load-store as memory access is only possible using load/store instructions. All other instructions use registers as operands, so these ISAs can be classified as register machines [9, p. 42].

WebAssembly WebAssembly is an open standard for a "virtual instruction set architecture" based on a stack machine [10, p. 5-7]. Instruction opcodes are one or two bytes long, but since immediate arguments can follow these opcodes, the instruction set is variable-length [10, Section 5.4]. Memory access is possible via load-store instructions, the byte order is little endian [10, p. 22]. There are 437 valid opcodes, with an additional 69 being currently reserved [11].

x86-64 The *x86-64* instruction set is a backwards-compatible successor of the *Intel 8086*'s architecture [12, p. 37]. Instruction size is variable and ranges from 1 to 15 bytes [12, p. 3058], the byte order is little endian [12, p. 32]. The basic ISA provides 16 general purpose registers, each having a width of 64 bits [12, p. 76]. The internal storage type for *x86* is a register-memory machine [2, Figure A.3], with some instructions like `MUL` and `DIV` using implicit accumulator registers as operands and destinations [12]. Heule et al. count at least 981 mnemonics and 3,684 instruction variants [13], allowing classification as *CISC*.

Z80 The Zilog *Z80* is a microprocessor introduced in 1976, with later versions still being used today [14]. It features 158 instructions of which 78 are adapted from the Intel 8080 CPU, making the instruction set backwards-compatible [15, p. 46]. Instruction size is variable ranging from 1 to 4 bytes [15, p. 57]. They can operate on register or memory operands, with some storing results in accumulator registers [15, p. 40-47]. Featuring instructions like `LDIR` that can occupy the CPU for many clock cycles [15, p. 41], this instruction set can be classified as *CISC*.

3. Related Work

In [16], Davidson and Vaughan analyze the relation between instruction set complexity and program size. A technique called "instruction set subsetting" is used to eliminate biases that could arise when comparing different architectures. Three subsets of the rather complex *VAX* instruction set are created with decreasing complexity: while *MAXVAX* supports 16 addressing modes both in source and destination operands for almost all instructions, *MIDVAX* supports only eight addressing modes and restricts destination operands to registers only. Some more complex instructions are not available at all. The *MINVAX* instruction set further reduces available instructions and addressing modes, memory access is only possible via load-store instructions. Their comparison of the object code size of ten different programs shows an increase in average code size with reduced architectural complexity: compared to the baseline *MAXVAX* instruction set, programs compiled for *MIDVAX* are on average 1.54 times the size, while the average size of programs compiled for *MINVAX* grows to 2.48 times. They also note that average instruction sizes are 4.10 bytes for *MAXVAX*, 3.71 bytes for *MIDVAX* and 3.61 bytes for *MINVAX*, showing that the compiler is able to use more complex and large instructions when they are available.

To reduce implementation complexity, only a subset of an instruction set could be implemented. In [17], Akshintala et al. analyze the distribution of instruction opcodes in Linux packages for the *x86-64* architecture. A table [17, Table 5] shows the number of instructions needed to support a given percentage of available packages. They find that an emulator aiming to run 80% of available packages would have to implement 189 instruction mnemonics, while for 90% of packages 230 mnemonics are required. 611 additional instructions are needed for full compatibility. They also recommend a sequence of instructions that can be used for such an implementation based on the popularity of packages using these instructions.

Another approach for reducing complexity or effort when implementing an emulator is using a very small ISA, like *LC-3* described by Yale N. Patt and Sanjay J. Patel in [18, pp. 520-545]. With a total of 16 opcodes, one of them reserved for the future, the instruction set is very small. It features eight 16-bit general purpose registers and is a load-store architecture [18, p. 553]. All instructions are 16 bits wide, with the upper 4 bits defining the opcode. Originally it was planned to include this architecture in the comparison in Section 4. However, due to a lack of functioning compilers from *C* to *LC-3*, it did not end up being included.

4. Comparison

This section compares the different ISAs introduced in Section 2.2 with regard to their complexity and code size.

4.1. Complexity

In order to assess the complexity required for an emulator implementation, we can look at the complexity of the target ISA. The distinction between *CISC* and *RISC* is on a high level, but can be helpful for estimating the complexity of operations before implementing them.

A typical difference is the concept of a load-store architecture (usually used in *RISC* ISAs) versus memory operands. In *CISC* architectures, it is often possible to directly operate on memory operands, requiring only one instruction for manipulation. Load-store architectures however must explicitly load and store data for manipulation in a register. Even if the encoding for the *CISC* instruction were larger in size than a typical *RISC* instruction, encoding only a single instruction instead of three can still result in smaller code. In fact, as shown in Figure 1, incrementing a value at a given memory address stored in a register (`r0` for *ARM T32*, `rdi` for *x86-64*) needs triple the amount of bytes to encode for the *RISC* architecture in this case.

```
03 68 ldr r3, [r0, #0]  |
01 33 adds r3, #1       |  ff 07 incl   (%rdi)
03 60 str r3, [r0, #0]  |
```

Figure 1: Comparing *ARM T32* (left) and *x86-64* (right)[1]

Another method for comparing the complexity of different ISAs is comparing the number of operations that are possible, including addressing modes. However, getting an accurate and up to date count of available opcodes for different architecture variants is challenging: as Heule et al. note in [13], as well as Mahoney and McDonald in [19], getting accurate data on *x86-64* opcodes is hard. Similar problems with other architectures, especially the inaccessibility of PDF files for programmatically counting opcodes, have prevented further analysis in this paper.

4.2. Code Size

To compare code size we compiled a simple *C* function for different architectures. The code is shown in Figure 2. A git repository containing all code, including Makefiles for generating the results for all architectures, is available online at https://github.com/xarantolus/iitm-surveying-isas.

Tools This section will outline the tools used for compiling and measuring program size for each architecture. The installation steps are also available in the git repository linked above.

All compilations were done on an *Ubuntu 20.04.4 LTS x86_64* system. Where available, the `-Oz` option is passed to compilers to "optimize aggressively for size rather than speed", else the `-Os` flag is used to "optimize for size" [20]. The following lists the tools used for compiling the C program for each architecture. More detailed installation instructions can be found within the git repository.

1. Adapted from [2, Figure A.2]

```c
int fib(int n) {
    if (n <= 1) { return n; }
    int prev = 0; int current = 1; int tmp;
    for (int i = 2; i <= n; i++) {
        tmp = current;
        current += prev;
        prev = tmp;
    }
    return current;
}
```

Figure 2: Fibonacci function written in C

ARM T32 The compiler used is *arm-linux-gnueabihf-gcc 9.4.0* from the *gcc-arm-linux-gnueabihf* package.

RISC-V Compressed The *riscv64-unknown-linux-gnu-gcc (g5964b5cd727) 11.1.0* compiler from the *RISC-V* toolchain available at [21] was used to compile the program. An install script is available in the `riscv` directory of the git repository.

WebAssembly Instructions for installing the *emcc 3.1.8* compiler are available at [22].

x86-64 For compilation *gcc 9.4.0* from the *gcc* package is used.

Z80 The *clang 12.0.0* compiler is built from the project at [23]. Installation steps were adapted from [24].

Results Table 1 shows the object code size of the compiled `fib` function from Figure 2.

The *ARM T32* object code includes 14 instructions with a size of 2 bytes each, making it the smallest result. No 4-byte instructions were needed, showing that the compiler was able to take advantage of the 16-bit instruction variants.

Similarly, the compiled output for *RISC-V Compressed* uses mostly 16 bit instructions, except for two 4-byte `BGE` instructions used to implement branches. One interesting detail is that the compiler generates two `RET` instructions at the end of the function, adding an extra two bytes while only one instruction would have sufficed. A maintainer of the *RISC-V* toolchain responded to our inquiry, calling this behavior "a missed optimization opportunity" [25].

The *WebAssembly* version uses mostly two-byte instructions. Despite the smaller average instruction size, its comparatively high number of instructions results in the largest output size.

The output compiled for *x86-64* contains 14 instructions with a length of 1 to 5 bytes. Nine instructions have a length of 2 bytes. Two `MOV` instructions are 5 bytes long as they encode a 4-byte immediate value. Additionaly, there is one single-byte, one three-byte and one four-byte instruction present. The small instruction count compared with *WebAssembly* and *Z80* leads to the overall smaller code size, despite the larger average instruction size.

While the instructions for the *Z80* version have an average size of only 1.88 bytes, it needs 34 instructions to implement the function, resulting in the second largest code size.

TABLE 1: Object code size of the fibonacci function from Figure 2 compiled for different architectures

Architecture	Compiler	Size	Instruction Count	Avg. Instruction Size
ARM T32	*arm-linux-gnueabihf-gcc 9.4.0*	28B	14	2.00B
RISC-V Compressed	*riscv64-unknown-linux-gnu-gcc 11.1.0*	30B	13	2.31B
WebAssembly	*emcc 3.1.8*	71B	37	1.92B
x86-64	*gcc 9.4.0*	36B	14	2.57B
Z80	*clang 12.0.0*	64B	34	1.88B
ARM A32	*arm-linux-gnueabihf-gcc 9.4.0*	52B	13	4.00B
RISC-V	*riscv64-unknown-linux-gnu-gcc 11.1.0*	52B	13	4.00B

In addition to the ISAs presented in Section 2.2, Table 1 also shows the code size of *ARM A32* and *RISC-V* without the "C" extension. Both ISAs have 32-bit wide fixed-length instructions [9, p. 25] [26].

ARMs' claim of code size reduction of the variable-length *ARM T32* ISA compared to *ARM A32* [7] holds true in this case with a reduction by almost half.

The code size for *RISC-V Compressed* is approximately 40% smaller than the size for its fixed-length *RISC-V* counterpart, exceeding the "25%-30% code-size reduction" claim for the compressed extension in this case [9, p. 115].

Limitations This comparison is rather limited as only one small program is compared using only one compiler for each architecture. In addition, different compilers might support different optimizations regarding the program size.

Comparison with Previous Work As mentioned in Section 3, Jack W. Davidson and Richard A. Vaughan found in [16] that reduced architectural complexity leads to larger overall code size.

If we classify *WebAssembly* as closer to *RISC* than *CISC* due to it being a load-store architecture, it fits this observation. However, the *Z80* instruction set also produces an overall large code size despite it being classified as *CISC*. Additionally, contrary to the expectations one might have on the basis of the mentioned study, the *RISC* architectures *ARM T32* and *RISC-V Compressed* result in the overall smallest code size.

One explanation for these findings is the limited example function. Its small amount of required variables makes it possible for all data to be stored in registers, foregoing the need to access memory. As mentioned in Section 4.1, requiring more instructions to read and write memory in *RISC* ISAs could lead to larger code size. Since no explicit memory accesses are necessary for the `fib` function, the disadvantages of *RISC* ISAs do not come into consideration in this case.

5. Conclusion and Future Work

In this paper we analyzed the object code size of a program when compiled for different architectures, creating a system of Makefiles that can be adapted for analyzing other *C* functions as well. While doing so, we found variations in the code size when compiling for different architectures and a possible size optimization in a *RISC-V* compiler.

Our example code reached the smallest sizes when compiled for the *ARM T32* and *RISC-V Compressed* architectures, indicating that they can be suitable for reducing overhead and cache misses in an emulator implementation.

However, the selection of ISAs is not the only factor when reducing code size. Future work can explore reducing complexity by limiting the set of opcodes and addressing modes available to a compiler when translating a program. The effect of register spilling on code size, especially for load-store architectures that need more instructions to operate on memory operands, can also be explored. Additionally, analyses similar to [17] by Akshintala et al. can be done for other ISAs before implementing an emulator.

References

[1] C. Douglas, *Essentials of Computer Architecture.* Chapman and Hall/CRC, 2017, vol. Second edition.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A quantitative approach*, 5th ed. Morgan Kaufmann, 2011.

[3] J. Ledin, *Modern Computer Architecture and Organization.* Packt Publishing, 2020.

[4] J. Catsoulis, *Designing embedded hardware*, 2nd ed. O'Reilly, 2005.

[5] Arm Limited, "About Thumb-2," https://developer.arm.com/documentation/ddi0308/d/Introduction-to-Thumb-2/About-Thumb-2?lang=en, [Online; accessed 31-March-2022].

[6] ——, *Arm® A32/T32 Instruction Set Architecture*, December 2021.

[7] ——, "T32 Instruction Set," https://developer.arm.com/architectures/instruction-sets/base-isas/t32, [Online; accessed 24-March-2022].

[8] ——, *Arm® Architecture Reference Manual for A-profile architecture*, 2022.

[9] Editors A. Waterman and K. Asanović, RISC-V Foundation, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121*, December 2019.

[10] WebAssembly Community Group, *WebAssembly Specification, Release 1.1 (Draft 2022-03-21)*, https://webassembly.github.io/spec/core/_download/WebAssembly.pdf, March 2022, [Online; accessed 1-April-2022].

[11] ——, "Index of Instructions – WebAssembly 1.1 (Draft 2022-03-21)," https://webassembly.github.io/spec/core/appendix/index-instructions.html, [Online; accessed 1-April-2022].

[12] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2021.

[13] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, "Stratified synthesis: Automatically learning the x86-64 instruction set," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 237–250. [Online]. Available: https://doi.org/10.1145/2908080.2908121

[14] D. M. G. Preethichandra, "Z80—the 1970s microprocessor still alive," *IEEE Micro*, vol. 41, no. 6, pp. 156–157, 2021.

[15] Z80 Microprocessors, *Z80 CPU User Manual UM008011-0816*, http://www.zilog.com/docs/z80/UM0080.pdf, [Online; accessed 30-March-2022].

[16] J. W. Davidson and R. A. Vaughan, “The effect of instruction set complexity on program size and memory performance,” *ACM SIGOPS Operating Systems Review*, vol. 21, no. 4, pp. 60–64, Oct. 1987. [Online]. Available: https://doi.org/10.1145/36204.36184

[17] A. Akshintala, B. Jain, C.-C. Tsai, M. Ferdman, and D. E. Porter, “X86-64 instruction usage among c/c++ applications,” in *Proceedings of the 12th ACM International Conference on Systems and Storage*, ser. SYSTOR ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 68–79. [Online]. Available: https://doi.org/10.1145/3319647.3325833

[18] Y. Patt and S. Patel, *Introduction to Computing Systems: From Bits and Gates to C and Beyond*, 2nd ed. McGraw-Hill Higher Education, 200.

[19] W. Mahoney and J. T. McDonald, “Enumerating x86-64–it’s not as easy as counting.”

[20] GCC team, “Optimize Options (Using the GNU Compiler Collection (GCC)),” https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, [Online; accessed 4-April-2022].

[21] RISC-V Software Collaboration, “GNU toolchain for RISC-V, including GCC,” https://github.com/riscv/riscv-gnu-toolchain, [Online; accessed 8-April-2022].

[22] The Emscripten project, “Emscripten SDK,” https://github.com/emscripten-core/emsdk, [Online; accessed 8-April-2022].

[23] Retro Computing at Georgia Tech, “The LLVM Compiler Infrastructure,” https://github.com/gt-retro-computing/llvm-project, [Online; accessed 8-April-2022].

[24] The IMSAI Gang, “LLVM Z80: Building,” https://imsai.dev/posts/build_llvm/, [Online; accessed 8-April-2022].

[25] “Duplicate ret instruction at end of function,” https://github.com/riscv-collab/riscv-gnu-toolchain/issues/1048, [Online; accessed 8-April-2022].

[26] Arm Limited, “A32 Instruction Set,” https://developer.arm.com/architectures/instruction-sets/base-isas/a32, [Online; accessed 5-April-2022].