Evaluation of ARM Assembly Generation for a Graduate Compiler Course Compiler

Per Filip Hedén
D18, Lund University, Sweden
fi6468he-s@student.lu.se

Fritjof Bengtsson
D18, Lund University, Sweden
eko15fbe@student.lu.se

Abstract

This paper presents an evaluation of using ARM assembly instead of x86 for code generation in a basic compiler, written in a introductory compiler course at LTH. The new ARM assembly generator specifically targets Raspberry Pi hardware [1]. The evaluation is based on the assembly languages usability for students learning about compilers. First a code generator for ARM was implemented. Then the language was extended with functions for interacting with *GPIO* hardware. Final conclusions from comparing x86 and ARM programs generated by the compiler was that they were almost identical. However, the Raspberry Pi gives students the opportunity to more directly interact with the hardware which other studies has found to be increasing student engagement and results.

1 Introduction

An important step in learning how software works and interacts with the hardware of a computer is learning about the compiler. Jalal Kawash et al. suggest using a Raspberry Pi for teaching students about compilers. Using a Raspberry Pi led to students comprehending the course material better. In addition, they received higher grades for the course [7].

Our paper compares building an assembly generator for a Raspberry Pi using ARM to a previously built x86 assembly generator, to evaluate which works better for teaching purposes. The specific Raspberry Pi model used in this project was a *Raspberry Pi 3 B+*, which can be seen from above in figure 1.

The starting point for the project was a compiler developed using *JastAdd* [3], and where parsing, semantic and name analysis were already implemented.

To determine if the ARM instruction set was suitable for the *Compilers* course, the complexity of the generated ARM code was analyzed and especially how it compares to the x86 counterpart.

1.1 Research Questions

- How can we get our *EDAN65* compiler to produce ARM assembly code?

[1]https://www.raspberrypi.com

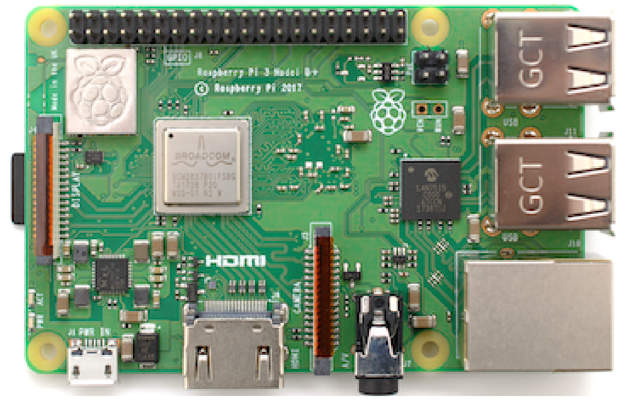

Figure 1. Picture of the *Raspberry Pi 3 B+*, the system used in this project. (*Image by Gareth Halfacree, press sample by Raspberry Pi Foundation, used under Creative Commons License*)

- Can we add instructions for using *GPIO* on a Raspberry Pi and run simple *SimpliC* programs that use this feature?
- How does ARM assembly compare to x86 in the *EDAN65* compiler, is the code still understandable? Could it be suitable for the course to change in the future?

2 Background

2.1 EDAN65 and SimpliC

EDAN65 is an introductory course in compiler implementation taught at Lund University for graduate students. During the course students use the *JastAdd* [3] meta compilation system to create their own compiler for a language called *SimpliC*, which is a subset of the *C programming language*. In the course students create a code generator for the compiler which targets x86 machines as the final step in the compilation pipeline [4].

2.2 ARM & RISC

ARM computers are a type of *RISC* computers. *RISC* stands for *Reduced Instruction Set Computer*, and what it means is not that *RISC* includes very few instructions, but instead that every instruction is conceptually simple and often uses only one clock-cycle. Longer, more complex instructions are not implemented in the hardware and is instead implemented in software [6].

```
// x86 assembly
movq %rax, $3
movq %rbx, $2
imulq %rbx, %rax

// ARM assembly
mov r0, 3
mov r1, 2
mul r1, r0
```

Figure 2. Code for multiplying 2 by 3 in both x86 and ARM assembly

2.3 CISC

The other popular computer architecture is *CISC*, or *Complex Instruction Set Computer*. In this instruction set there are no limitations on the instructions, and highly optimized hardware instructions can be implemented for more complex functionality. Some such instructions could have to be split into multiple operations on *RISC* hardware [6].

2.4 Computers Today

Today, the most common computer people use everyday, the smartphone, is usually a *RISC* machine [10]. Not only portable power-efficient computers use *RISC* architecture, even some of the most powerful computers in the world use reduced instruction set architectures [8]. This said, CISC computers are also in use today, mostly for high performance where energy efficiency is not as much of a concern and in some supercomputers. It is worth to mention that RISC and CISC have grown more similar over the years and that the differences are not large today as when RISC was first introduced.

3 Implementation

3.1 Writing basic ARM assembly

Just like in most other programming languages there exists multiple ways to write a program solving a given problem. The code generation in this project mostly follows the ARM assembly guidelines and examples presented in the book *Introduction to Computer Organization: ARM Assembly Language Using the Raspberry Pi* [9] in combination with the x86 syntax used in *EDAN65*. This means that ARM conventions are not always used, as the ARM code was translated line for line from the x86 version completely unoptimized. An example of this is that even though programmers in ARM have access to many general purpose registers and can use these for faster argument passing, we pass arguments on the stack as is done the *EDAN65* compilers generated x86 code.

For small programs with basic functionality, the assembly instructions are almost identical. This makes it trivial to translate between x86 and ARM for these types of programs. As one can see from the example in figure 2, for basic operations there is practically no difference between the instruction sets.

One thing that could differ is the amount of bytes for storing the integer values. In the x86 version of the compiler integers were assumed to use 8 bytes, while their ARM counterpart used only 4 bytes. This can be changed by the programmer while retaining functional code.

In the original x86 code generation, assembly was generated for a basic print and read functions. This code was not written by students of the course but was instead given to them. In our ARM version of the code generation we used a similar approach, having the compiler generating two helper functions `read` and `print`, but instead of using the system calls read and write we used the *C* library functions `scanf` and `printf`. This was done because the x86 `read` and `print` helper functions were quite long, about 25% of the original code generation source code, and it was therefore deemed outside of this projects scope.

The implementation of the ARM assembly generator was done by using Robert G. Plantz book to translate the generated code line by line, at least for more basic operations. For the more advanced operations we used the book first, then for programs that crashed we compared this assembly code to the unoptimized assembly code generated by *GCC* [2] using the `-S` and `-O0` flags. Finally we debugged anything still not working using *GDB* [3].

3.2 Implementing new SimpliC language features for GPIO on the Raspberry Pi

With the additional feature of *GPIO* ports on the Raspberry Pi system board we created language constructs for accessing these pins, or rather controlling the LED panel on the Raspberry Pi Sense HAT. To implement these features we looked through the official documentation. This however only included documentation for the language Python. Instead we found *C* code to control the LEDs and used this together with *GCC* to understand how assembly code for accessing the LEDs could look. Finally we added a new function to the *SimpliC* language called `led(int index, int color)`, which takes two integers to set a LED lights color. This included making a corresponding code generation function in the ARM code generation aspect to produce working assembly programs.

In addition to the `led` function a basic sleep function was added to the language to enable animations on the LED panel. The function was more or less a wrapper for the standard *C* library function `sleep` and takes a integer as argument for how many seconds to wait with the signature `wait(int seconds)`.

3.3 Comparing ARM and x86 assembly code

To showcase some of the similarities and differences between the ARM and x86 assembly code a simple program from the

[2]https://gcc.gnu.org

[3]https://sourceware.org/gdb/

```
int f(int a, int b) {
    return a*b;
}
```

Figure 3. Example *SimpliC* function.

```
 1  f:
 2      pushq %rbp
 3      movq %rsp, %rbp
 4      movq 8(%rbp), %rax
 5      pushq %rax
 6      movq 16(%rbp), %rax
 7      pushq %rax
 8      movq 16(%rbp), %rax
 9      pushq %rax
10      movq 24(%rbp), %rax
11      movq %rax, %rbx
12      popq %rax
13      imulq %rbx, %rax
14      pushq %rax
15      movq %rbp, %rsp
16      popq %rbp
17      ret
```

(a) Example of x86 assembly code.

```
 1  f:
 2      push {lr}
 3      push {fp}
 4      add fp, sp, 0
 5      sub sp, 12
 6      ldr r1, [fp, 8]
 7      str r1, [fp, -8]
 8      ldr r1, [fp, 12]
 9      str r1, [fp, -12]
10      ldr r1, [fp, -8]
11      push {r1}
12      ldr r1, [fp, -12]
13      mov r0, r1
14      pop {r1}
15      mul r1, r0
16      mov r0, r1
17      sub sp, fp, 0
18      pop {fp}
19      pop {lr}
20      bx lr
```

(b) Example of ARM assembly code.

Figure 4. Comparison between x86 and ARM assembly code generated from the compiler for the *SimpliC* function from figure 3.

compiler's test suite was used. The *SimpliC* code can be seen in figure 3 and the comparison of the generated assembly programs can be seen in figure 4. As noted in the figure text one should keep in mind that the full assembly programs are not showcased in the figure, as the helper functions and static data inhabit many lines of code, instead the full programs can be found in the repository under the name *\testfiles\genCode\parameter0.s* [4].

One difference between ARM and x86 is how the `return` or `ret` instruction works, namely ARM does not have it, see figure 4 line 17 (a) and line 20 (b). Instead of calling return we manually save the link register on the stack with a push at the beginning of a procedure, see line 2 figure 4b to then pop it at the end, see line 19 4b before using a regular branch instruction to go to the correct instruction in the caller. There are multiple other ways to do this using the ARM instruction set, for example *GCC* pops the link registers into the program counter to immediately return in one instruction replacing line 19 and 20 in figure 4b.

Another difference between ARM and x86 are how values from the stack are read to integers. As can be seen in figure 4*a* the procedure f reads a value on the stack using the operation `movq` (line 4), same as for copying values between registers. In ARM we can see that another instruction is used, `ldr`, which *loads* the value from an address in memory (see line 6 4b). These are of course quite similar in that they use an offset from the frame pointer to find the value in memory, but its still notable that they use different syntax in this case.

[4]https://bitbucket.org/edan70/arm-filip-fritjof/src/master/

	ARM (LoC)	**x86** (LoC)
total generated	3367	2881
JastAdd aspect	415	431
static generated	115	119

Table 1. Lines of code measurement and difference between ARM and x86 assembly programs and *JastAdd* generator aspect. Example programs are the test cases used for the compiler. Static code is the helper functions and headers included in every assembly program generated by our compiler.

Another difference encountered in the work was the mathematical operation of division. The division instruction in ARM is a rather new introduction to the instruction set, first a standard in the *ARMv7* architecture [5]. It did not make a difference in this project as the computer that was used, a *Raspberry Pi 3 B+*, uses the newer *ARMv8* architecture [6].

There are many more differences between the instruction sets, especially if the goal is to create optimal performance, but for most instructions relevant to this project the differences are just syntactic and very simple to translate. Some of these syntactic differences are; using curly brackets around register when calling pop or push, or emitting the dollar sign before primitives. The names of registers are also different from x86. In ARM the general purpose registers are called *r0-r30* (*r0-r12* for 32-bit) instead of *rax* and *rbx*, the frame pointer is called *fp* instead of *rbp* and the stack pointer is called *sp* instead of `rsp` [4] [1].

Finally, as for length of programs, there wasn't a huge difference between generated x86 and ARM assembly for the compiler used in this project. However, the difference grows larger for each procedure as the ARM procedures use 3 lines for return instead of 1, and 1 additional line at the start of a procedure. In table 1 our measurements for lines of code in can be seen. These measurements were taken from our test programs and test all different language features except for the newly added `led` and `sleep`. As there were 21 generated assembly programs in the test we chose to use include to total lines of code instead of measurements for each file, but as stated earlier the difference generally grows larger the larger the program is.

[5]https://developer.arm.com//media/Arm%20Developer%20Community/PDF/Cortex-A%20R%20M%20datasheets/Arm%20Cortex-M%20Comparison%20Table_v3.pdf?revision=a2b3e330-d417-49cc-8037-7f034a19197e&la=en&hash=BF9752AB2044B1FDB7EAEF957A1D92F2943FA265
[6]https://www.raspberrypi.com

4 Evaluation

A relevant evaluation of the project could be to compare lines of code in our new *JastAdd* aspect for ARM code generation with the previous x86 aspect. Our measurement shows that the difference between the length of the *JastAdd* aspects are just a few lines, and that the same is true for the part of the generated assembly that is required for helper function and setup. There is a quite significant difference in length between the lines of assembly code though, where the ARM programs are about 15% longer than their x86 counterparts. As mentioned in the previous section this is mostly due to procedure initialization and return requiring additional operations.

When it comes to evaluating the number of lines of code for the generated code for each instruction set using the compiler, its not really important which one uses the fewest. Instead whats important is that no architecture forces the students to write lots of so-called 'boilerplate' code that is not relevant to acquiring more knowledge of the implementation of compilers. In this aspect neither x86 nor ARM assembly provides and advantage over the other as, as previously stated, they can be translated between each other almost line for line. One possible advantage of the ARM assembly generated in this project is the inclusion of pushing, popping and branching to the link register. This addition does make the ARM assembly programs two lines longer per function, but could provide students better understanding of what a 'return' instruction actually does. Something similar could be done in x86, so its still not a real advantage for ARM.

The issue we considered will be most difficult for students, assuming they will still be given the assembly code for a working print procedure, is generating code for integer division. Having multiple procedures to accomplish this is of course more difficult than simply writing 'div', but we also believe this could be easily remedied by including a section on how integer division could be implemented with shifts in the assignments instruction manual or appendix. This is of course not a problem for newer ARM architectures, *ARMv7* and above. This could however be a problem for implementing a more general ARM compiler and especially if students of a potential course would be writing for older hardware. For example the first generation of the Raspberry Pi and the low power versions Pico and Zero use the *ARMv6* architecture [7].

Another issue that is more practical is where to run the code. Many students have personal computers running x86, and so does LTH:s computers that are available to students. Emulators exist but installing and configuring these could be more complicated for students. Every students probably has a computer running ARM in their pockets, but this cannot easily be used to compile code on as both Android and iOS are locked from this. The option we had was a Raspberry Pi, and we believe this was a good option but purchasing enough of these microcomputers for all students taking the compiler course would be costly. A good solution could be to just have a few and let students use SSH to connect to these computers. This would of course lead to the students also having to get acquainted with SSH but this could be considered sufficiently simple as it is pre-installed on most computers and is a single program. Also this could prove difficult with using GPIO as only one pair of students could access these at a time on the same Raspberry Pi.

5 Related work

The work by Javal Kawash et al. is of course related and we base much of our research on their paper. In their study, they conducted a trial of teaching undergraduate students compiler concepts in a course using Raspberry Pi's and ARM assembly. Their results showed that students learnt the concepts better when having hands on experience with a physical computer hardware such as the Raspberry Pi, instead of developing assembly for their usual personal computers which resulted in higher grades and higher student satisfaction [7].

Another recent paper concluded that the Raspberry Pi could be used in a wide range of subjects; from higher level programming to lower level hardware fundamentals. Furthermore, many of the students that worked with the single board computer also made the choice of doing their graduating work projects with a Raspberry Pi [2].

Yet another study released in 2015 states the belief that using single board computers in their university curriculum has had many advantages, both for the curriculum and students. The authors also discuss some of the challenges one may encounter when incorporating these computers in a course [5].

It is worth to mention that the two later papers discuss the use of single board computers in the broader sense of computer science and electronics, not just for compiler technology.

These papers conclude that students learn about software better when working close to hardware, and since we have concluded the feasibility of of using a Raspberry Pi as a target for the EDAN65 course compiler it could be beneficial to trial the usage of Raspberry Pi:s in this course in a future iteration.

6 Conclusion

In many ways, writing ARM assembly is quite similar to x86 for small basic programs. Almost all code-generation from the two different instruction sets could in this compiler be translated almost line for line. In some ways ARM assembly might be easier to understand than x86, but this is mostly due to syntax and not due to lower complexity and only our opinion. In most aspects related to the *EDAN65* course

[7]https://en.wikipedia.org/w/index.php?title=Raspberry_Pi&oldid=1065303384

and in implementing a basic compiler with assembly code generation the instruction sets are almost identical. Thus the answer to 'how' to change a x86 assembly code generating compiler to produce ARM code, the answer for basic operation in *SimpliC* is to almost line for line translate, using the appropriate instruction set.

A benefit of programming on a Raspberry Pi is the access to GPIO pins, enabling students to visually interact with hardware through their code. Students interacting directly with system calls to the operating system using assembly might get a better understanding of how hardware, operating systems and drivers work.

As stated in the first paragraph of the conclusion, for basic programs and without optimization the assembly code for ARM and x86 are very similar. However both authors found the ARM code slightly more readable and understandable with the choices of register names, use of primitives and operation names. If all students in future iterations of the course could get access to ARM computers, either through actual hardware or emulators, we believe it could be suitable to use ARM assembly and that the transition would be easy.

7 Continued Work

An extension of the compiler that could be interesting is the use of a generic `write` system call instead of `printf`, which would not require any extra libraries and thus making the use of GCC for linking obsolete. This would require writing an `itoa` or *integer to ascii* function in assembly as `write` cannot naively print numbers. It could also be helpful to hide less steps of the compilation process for students so they have an easier time understanding it. The same can of course be done in the case of using `read` instead of the C library function `scanf` that is currently in use. To use `read` one would need to implement the reverse of an `itoa` function, an `atoi` function or *ascii to integer* to enable arithmetic operations on the values read by the programs.

On a RISC computer the programmer has access to many general purpose registers. This means that in general function parameters do not need to be passed on the stack, but can instead be passed through some of the general purpose registers. How many registers are allowed for argument passing is dependent on the architecture, but the number for the ARM architecture on the *Raspberry Pi 3 B+* board, which was used in this study, is 31 if run in 64-bit mode [9]. A future version of this compiler could then take advantage of using registers for argument passing instead of the stack.

Acknowledgments

Alfred Åkesson, our supervisor. Who guided us through the project with great advice and feedback.

Görel Hedin who teaches the introductory compiler course at LTH and made us excited about the subject.

Jonas Skeppstedt who helped us understand linking, and continually improves the authors understanding on all things compilers and low level computational systems.

References

[1] Arm. 2019. ARM® Compiler for µVision® armasm User Guide. (2019). https://www.keil.com/support/man/docs/armasm/armasm_deb1353594352617.htm

[2] Branko Balon and Milenko Simić. 2019. Using Raspberry Pi computers in education. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 671–676. https://doi.org/10.23919/MIPRO.2019.8756967

[3] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd Extensible Java Compiler. *OOPSLA 2007,Montreal, Canada,– ACM Sigplan Notices* 42 (2007), 1 – 17. https://doi.org/10.1145/1297105.1297029

[4] Görel Hedin. 2021. EDAN65 Compilers. (2021). https://fileadmin.cs.lth.se/cs/Education/EDAN65/2021/web/index.html

[5] Peter Jamieson and Jeff Herdtner. 2015. More missing the Boat—Arduino, Raspberry Pi, and small prototyping boards and engineering education needs them. In *2015 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–6. https://doi.org/10.1109/FIE.2015.7344259

[6] T. Jamil. 1995. RISC versus CISC. *IEEE Potentials* 14, 3 (1995), 13–16. https://doi.org/10.1109/45.464688

[7] Jalal Kawash, Andrew Kuipers, Leonard Manzara, and Robert Collier. 2016. Undergraduate Assembly Language Instruction Sweetened with the Raspberry Pi. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16)*. Association for Computing Machinery, New York, NY, USA, 498–503. https://doi.org/10.1145/2839509.2844552

[8] Don Monroe. 2021. Fukagu takes the lead. In *Communications of the ACM*. ACM, 16–18. https://doi.org/10.1145/3433954

[9] Robert G. Plantz. 2021. Introduction to Computer Organization: ARM Assembly Language Using the Raspberry Pi. (2021). https://bob.cs.sonoma.edu/IntroCompOrg-RPi/frontmatter-1.html

[10] Manoj Pratap Singh, Mahendra & Kumar. 2014. Evolution of Processor Architecture in Mobile Phones. In *International Journal of Computer Applications*. IJCA. https://doi.org/10.1145/3433954