FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Tiny86 Debugger
Student:	Bc. Filip Gregor
Supervisor:	Ing. Petr Máj
Study program:	Informatics
Branch / specialization:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2023/2024

Instructions

Familiarize yourself with the architecture and instruction set of the Tiny86 virtual machine used in the NI-GEN course. Investigate native debuggers on the Windows and UNIX platforms and their principles. Improve the debugging capabilities of Tiny86 so that they are comparable to modern native debuggers. Design a debugging interface and implement a debugger as a reference client. The debugger must support native and source level debugging, but be extensible for intermediate level debugging (such as IR).

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 29 December 2022 in Prague.

Master's thesis

TINY86 DEBUGGER

Bc. Filip Gregor

Faculty of Information Technology
Department of theoretical computer science
Supervisor: Ing. Petr Máj
March 31, 2023

Czech Technical University in Prague
Faculty of Information Technology
© 2023 Bc. Filip Gregor. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Gregor Filip. *Tiny86 Debugger*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

Acknowledgments **vii**

Declaration **viii**

Abstract **ix**

1 Introduction **1**
1.1 Debugging . . . 3
1.2 Teaching Compilers . . . 5
1.3 Goals of the Thesis . . . 5
1.4 Structure of the Thesis . . . 6

2 Debugging support **7**
2.1 CPU level support . . . 7
2.1.1 Interrupts . . . 10
2.1.2 Superscalar CPUs . . . 11
2.2 Operating system support . . . 11
2.2.1 Linux . . . 12
2.2.2 UNIX systems . . . 18
2.2.3 Windows . . . 18
2.3 Source Level Debugging . . . 23
2.3.1 Executable file formats . . . 23
2.3.2 DWARF . . . 23
2.4 Compilers . . . 32
2.4.1 Modularity of compilers . . . 34
2.4.2 IR Debugging information . . . 34
2.5 Existing debuggers and disassemblers . . . 37
2.5.1 GDB . . . 37
2.5.2 IDA Free . . . 37
2.5.3 Microsoft Visual Studio Debugger . . . 37
2.5.4 Case study - LLDB . . . 39

3 Tiny x86 **41**
3.1 The T86 Instruction Set Architecture . . . 41
3.2 T86 Virtual Machine . . . 42

4 Implementation **45**
4.1 T86 ISA Extensions . . . 45
4.2 T86 Debugging Support . . . 46
4.3 Native Debugger . . . 48

4.4 Debugging Source Code 51
4.4.1 Line Information 51
4.4.2 Debugging Information Format 51
4.4.3 Location Expressions 54
4.4.4 Frontend 54

5 Evaluation **55**
5.1 The development process 55
5.2 Usage and user testing 55
5.3 Performance 56

6 Conclusion **59**
6.1 Summary 59
6.2 Future Work 59

Contents of enclosed media **65**

List of Figures

1.1 An example of a program that checks if a number is positive, shown in both assembly language and machine code. . . . 2
1.2 A function written in the C programming language that check if a number is positive. . . . 2
1.3 Binary search algorithm written in the C programming language. . . . 3
1.4 Example of debugging the binary search algorithm in the LLDB [6] debugger. 4
1.5 Example of debugging a program in the LLDB debugger without debugging information generated by the compiler. . . . 5

2.1 Compiled C program with GCC 9.4 compiler as x86-64 assembly. . . . 9
2.2 Linux - Debugger initialization. . . . 14
2.3 Linux - Reading and writing one byte using ptrace. . . . 14
2.4 Linux - Setting one register using ptrace 15
2.5 Code with and without a breakpoint. . . . 15
2.6 Enabling breakpoints, abbr. . . . 16
2.7 Continue sketch 16
2.8 Stepping over breakpoint, abbreviated. . . . 17
2.9 A diagram representing the communication between the debuggee, the Windows operating system and the debugger [22]. . . . 21
2.10 A structure which contains an information about debugging event. . . . 21
2.11 An abbreviated example of a Windows Debugger loop, taken from [24]. . . 22
2.12 A program in the C programming language and the generated machine code by a GCC compiler. . . . 26
2.13 A DWARF description of mapping source lines to addresses for program in figure 2.12. . . . 27
2.14 A part of DWARFs DIEs for program in figure 2.12 28
2.15 A prologue and epilogue of a function in assembly. . . . 30
2.16 A DWARF stack frame information about the program in 2.15. Top part contains the encoded program for virtual machine, bottom part contains the decoded table from interpreting the program. Abbreviated. . . . 31
2.17 Basic stack layout after function prologue, disproportional. . . . 31
2.18 Example of subroutine type and pointer to function in DWARF, abbreviated. 32
2.19 Simplified structure of a compiler, some parts were left out, like optimizations. 33
2.20 Simplified example of an abstract syntax tree. . . . 33
2.21 Simplified example of LLVM IR. . . . 34
2.22 An example of LLVM IR with debugging information, abbreviated. . . . 36
2.23 An example of a factorial program disassembled with IDA. . . . 38
2.24 An example of debugging factorial program with MSVC debugger. . . . 39

3.1 A small piece of a T86 program. . . . 42

3.2 Simple example of how to create and run a simple program in the T86 VM. 42
3.3 Example of an debug instruction in T86 VM. 43
3.4 Small debugger implementation using T86 BRK instruction, abbreviated. . 43

4.1 Example of an T86 program which prints ”Hello, World!”. 46
4.2 A sequence diagram for the communication between the virtual machine and the debugger. 49
4.3 Debugger code in `Native` class to enable a breakpoint. 50
4.4 Debugging function information for the T86 debugger. 52
4.5 Debugging type information for the T86 debugger. 53
4.6 Debugging type information for the T86 debugger. 53
4.7 Example of the CLI reporting a breakpoint hit. 54

List of Tables

5.1 Performance comparison when using various features of the debugger. The Quicksort had 674 breakpoints hits, while prime numbers had 101266 hits. Each case was run five times and average was taken. 56

First and foremost, thanks to my parents, who have always supported me and enabled me to pursue my dreams, whatever they may be.
I would also like to thank my supervisor, who spent an ungodly amount of time and patience to break through my stubbornness and make this thesis what it is.
Last but not least, thanks to my girlfriend, Jana. Without her support, this work would never have come to be.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

In Praze on March 31, 2023 .....

Abstract

Many university courses teach compiler design. One such course is taught at FIT CTU. The students are tasked to create a compiler from a C-like language called TinyC into the Tiny x86 architecture. This architecture does not have any processor that implements it. Instead, it uses a virtual machine.

To make a programming language usable, it needs proper tooling, like debuggers. For debuggers to work, the compiler must emit debugging information describing the connections between generated machine code and the source code. The virtual machine has almost no debugging support. Hence the students do not have any reason to generate debugging information.

That is about to change. This thesis describes how debugging support is provided at various levels, like the central processing unit and the operating system. It also describes how to create a very simple debugger on Linux. We then add debugging support to the Tiny x86 virtual machine itself, which is partially inspired by the studied Linux debugging interface. Finally, we create an assembly and source-level debugger inspired by real-world debuggers' design.

Keywords Debugging, Debugger, Debug, Debugger implementation, Compiler, LLVM, Linux, Windows, Tiny x86, Debugging support, Errors in programs

Abstrakt

V mnoha univerzitních kurzech se vyučuje návrh překladače. Jeden takový předmět se vyučuje na FIT ČVUT. Studenti mají za úkol vytvořit kompilátor z jazyka TinyC podobnému jazyku C do architektury Tiny x86. Tato architektura nemá žádný procesor, který by ji implementoval. Místo toho používá virtuální stroj.

Aby byl programovací jazyk použitelný, potřebuje vhodné nástroje, například ladicí programy. Aby ladící program fungovaly, musí překladač emitovat ladicí informace popisující vazby mezi generovaným kódem a zdrojovým kódem. Virtuální stroj pro Tiny x86 nenabízí téměř žádnou podporu pro ladění. Studenti tedy nemají žádný důvod generovat ladicí informace.

Tato práce popisuje, jak je podpora ladění poskytována na různých úrovních, jako je procesor, operační systém a překladač. Popisuje také, jak vytvořit velmi jednoduchý ladicí program za použití rozhraní systému Linux a Windows. Dále je popsána podpora ladění virtuálního stroje Tiny x86, která byla v rámci práce přídána. Ladící podpora byla inspirována rozhraním které nabízí systém Linux. Nakonec je vytvořen debugger na úrovni assembleru a zdrojového kódu inspirovaný návrhem ladících programů z reálného světa.

Klíčová slova Ladění, ladící program, Překladač, Implementace ladícího programu, LLVM, Linux, Windows, Tiny x86, Podpora pro ladění, Chyby v programech

Chapter 1
Introduction

At the core of every computer program lies the Central Processing Unit (CPU), which is responsible for executing programs. The CPU excels at performing very primitive operations very fast. These operations are called instructions, they can perform simple arithmetics, move values from and to memory, and change the control flow of the program. The instructions are encoded as a sequences of binary numbers which are easy for the CPU to understand, but for humans they are rather incomprehensible.

To help programmers better understand written programs, a text mapping was created, called *assembly language*. Each instruction is now assigned an text representation, as are the operands of the instruction. The control flow instructions now doesn't have to jump to address offset, but can instead use labels. Example of a simple program in assembly language can be seen in figure 1.1. If a programmer knows the instruction set architecture [1] of the processor he can easily recognize the instructions the program is made of.

But, as the computers became increasingly more powerful, so did the programs became bigger and more complex. When programming in the assembly language, the programmer must have an extensive knowledge of the processor's internal workings.

To spare the programmers from this, high level programming languages were created. These are designed to abstract from the specific machine the program will run on, allowing programmers to focus more on their task. As shown in figure 1.2, even a simple program written in the C programming language, one of the oldest programming language around, provides a clear understanding of the functionality. In contrast, examining the equivalent program in assembly language, as seen in figure 1.1, requires knowledge of the specific architecture of the machine.

However, we previously said that processors can only understand machine code and high level languages are far from it. Therefore, a translation of the high-level language into machine code is necessary, this is a task for a *compiler*. A compiler is a program that reads some source code of high-level language and produces machine code. Compilers are a very complicated piece of software, and we will talk about them in detail in chapter 2.3. For now, it is important to uderstand that the computer cannot run the source code of a high-level language directly, and that it is translated into machine code. Additionaly, compilers often take advantage of special features of the architecture to make the programs faster [2].

```
positive:
        push    rbp
        mov     rbp, rsp
        mov     -4[rbp], edi
        cmp     -4[rbp], 0
        jle     neg
        mov     eax, 1
        jmp     pos
neg:
        mov     eax, 0
pos:
        pop     rbp
        ret
```

```
01010101 01001000
10001001 11100101
10001001 01111101
11111100 10000011
01111101 11111100
00000000 01111110
00000111 10111000
00000001 00000000
00000000 00000000
11101011 00000101
10111000 00000000
00000000 00000000
00000000 01011101
11000011
```

Figure 1.1 An example of a program that checks if a number is positive, shown in both assembly language and machine code.

```
bool positive(int n) {
    if (n > 0) {
        return true;
    } else {
        return false;
    }
}
```

Figure 1.2 A function written in the C programming language that check if a number is positive.

```c
int binary_search(int* arr, int len, int n) {
    int lo = 0;
    int hi = len;
    while (lo < hi) {
        int i = (lo + hi)/2;
        if (arr[i] < n) {
            lo = i;
        } else if (arr[i] > n) {
            hi = i;
        } else {
            return 1;
        }
    }
    return 0;
```

Figure 1.3 Binary search algorithm written in the C programming language.

In the figure 1.3 we present a more complicated example of a program written in high level programming language. This is an implementation of the binary search algorithm. As an input, it receives sorted sequence of numbers and some number n and checks if that number n is in the sequence. This algorithm is widely used when searching in sorted sequence because of its $\mathcal{O}(\log_2(n))$ complexity [3].

1.1 Debugging

Programs are mostly written by humans, who tend to make mistakes [4]. We are no exception, as we have also made a mistake in the binary search program. Let us try to run the program with a `[1,2,3]` sequence and search for the number 4. This number is not in the sequence, so the expected output would be 0. Instead, if we ran the program it would run forever, because of a mistake we made in the source code. Such mistakes are called a *bug*[1]. The process of finding these mistakes and correcting them is called *debugging* [5].

We should fix this bug. Otherwise, users of our program will probably not be happy if their program gets stuck. We could try to look at the source code and find the mistake this way. This is a valid strategy. Here we could assume that the condition `lo < hi` never comes to be since it is the most obvious place where we could get stuck forever. Now, it would be helpful if we could see the states of `lo` and `hi` in each iteration of the cycle. We could resort to print statements, but that is not very flexible. If we changed our minds and wanted to also see the value of variable `i`, we would have to recompile the program and rerun it. The output can also quickly get overwhelming, especially in an infinite loop. However, all is not lost, as a special program was created to inspect running programs, called a *debugger*.

Debugger is able to inspect the state of another program, like the values of its variables. It is also able to control the flow of the program. They allow *breakpoints* to be set at each

[1]The term *bug* actually comes from a real bug that got stuck in relays back when computers were made from relays. They literally had to debug the machine by taking the bug out.

```
   5                int hi = len;
   6                while (lo < hi) {
   7                    int i = (lo + hi)/2;
-> 8                    if (arr[i] < n) {
   9                        lo = i;
   10                   } else if (arr[i] > n) {
   11                       hi = i;
Target 0: (a.out) stopped.
> p lo
(int) $0 = 0
> p hi
(int) $1 = 3
```

Figure 1.4 Example of debugging the binary search algorithm in the LLDB [6] debugger.

line of the source code[2]. When the program is about to execute the line of code with the breakpoint, the control is passed back to the debugger and the user can inspect the state of the program at that line. There are also conditional breakpoints, which only trigger when some condition holds. Example of such condition can be that the breakpoint gets activated only when `i == 3`.

Finally, debuggers also allow *stepping*. This also modifies the control flow of the program.

- *step in* - Executes current statement and stops on the next one. If the current statement is a function call then it will be executed and the program will be paused on the first statement in that function.
- *step over* - Same as a step in, but if the current statement is a function call then the program will be paused on the next statement after the call.
- *step out* - Executes as much as needed to return from the current function. Stops on the next statement that should be executed after the function returns.

Now, back to our program. Let us put a breakpoint on line 8 after i is set. We will monitor how the `lo` and `hi` change. If you run the program with the debugger attached, you will see an output similar to what is displayed in figure 1.4. Here, you can see the line on which the execution was stopped. It is also possible to print the state of variables. In each loop, you can print the value of a variable and then continue execution until another breakpoint. There is only one set, so the execution will again be stopped on line 8. The value of `hi` will not change, which is expected. Value of `lo` will gain following values: $0, 1, 2, 2, 2, \ldots$. The value apparently gets stuck at 2 and never change. The value of variable i is computed as $i = (\text{lo} + \text{hi})/2 = (2 + 3)/2 = 2$, because division in C rounds the value down. The fix is to change the line 9 to `lo = i + 1`. With the debugger, it was simple to find out where the error came from and we didn't have to recompile the program.

[2] Advanced debuggers allow breakpoints to be set inside expressions. This is especially important for functional languages, as their functions often consist of one big expression.

```
->  0x100003e3c <+112>: b      0x100003e78               ; <+172>
    0x100003e40 <+116>: ldr    x8, [sp, #0x20]
    0x100003e44 <+120>: ldrsw  x9, [sp, #0xc]
    0x100003e48 <+124>: ldr    w8, [x8, x9, lsl #2]
```

Figure 1.5 Example of debugging a program in the LLDB debugger without debugging information generated by the compiler.

We previously said that processors themselves only understand machine code. So how it is possible that we can debug the program and the debugger knows about lines, variables, etc. when the program itself is just machine code. The compiler has to lend a hand here. It embeds information about the source code. For example, it maps lines of source code to machine code instructions. Thanks to this mapping, the debugger knows that line x corresponds to instruction y in the machine code and can put a breakpoint there. If the compiler doesn't emit any information into the executable the debugger would only work with assembly, as seen in figure 1.5. This is a lot more discomforting than debugging source code directly.

1.2 Teaching Compilers

Many schools about computer science have a compiler course, and the Faculty of Information Technology, CTU is no exception. The course is called *Code Generators* (NI-GEN). cite? In this course, students are tasked to build a simple compiler from a C-like language called TinyC. The target of the compiler is the Tiny x86 (T86) architecture. This architecture does not have a processor that implements it. Instead, a virtual machine, a program that reads the assembly and executes it, was created for it. The architecture is supposed to ease the code generation and let the students focus on the more interesting parts of the compiler, like register allocation or optimization, instead of the nitty gritty detail of real CPU architectures.

There is, however, a problem with using T86, as it has almost non-existing debugging support. So if a compiler of some student generates the code badly, which is frankly inevitable, it takes a non-trivial amount of effort to find the error. T86 has some very light debugging capability, but it is far from real debugging. Also, compiling debugging information is not taught in the NI-GEN course because there is no reason to as of now. If a debugger was provided to the students, it might be incentivizing to compile such information to ease their lives later, when they will need to find errors in their compilers. This way, they will also learn how and what information the compiler needs to embed for the debugger to work.

1.3 Goals of the Thesis

The primary goal is to add debugging support to the T86 and to create a debugger that supports debugging both on the machine code level and on the source code level. The debugger should be extensible enough to also work with an intermediate representation. The debugger should be similar to real-world debuggers, in terms of how it works. This will require non-trivial changes in the T86 virtual machine source code. The students'

compilers will also have to generate debugging information. The format of the debugging information should be so that it is not discouraging for students to generate but also comparable to debugging information generated by real compilers.

1.4 Structure of the Thesis

1. The *Introduction* is the motivation behind the thesis and introduces basic terms with which should the reader be familiar.
2. *Debugging support* describes how are debuggers implemented and what debugging support is provided on various levels (OS, processors, compilers) to make their implementation possible.
3. *Tiny x86* describes the T86 architecture and discusses some of the parts of the virtual machine, mainly its existing debugging capabilities.
4. *Implementation* focuses on extending the T86 instruction set and adding a debugging interface to the virtual machine. It also describes the implementation of the debugger and the format of the debugging information chosen.
5. *Evaluation* evaluates the performance of the debugger and its ease of use.
6. *Conclusions* summarizes the result of the thesis and speaks of possible future work.

Chapter 2
Debugging support

> *Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*
>
> Brian W. Kernighan

To debug a program written in a high level programming language, we need the compiler to emit debugging information. Without this information, we can debug the program at the assembly level. This can still be useful, for example for reverse engineering. And without assembly level debugging, there is no source level debugging, since it depends on it. In this chapter, we describe at which level and what kind of debugging support is provided. First, we will mention what kind of mechanism the CPU itself offers. Going one step higher, we will talk about the API that various operating systems provide. Finally, we'll discuss how are compilers and debuggers able to allow us to debug source code although the program we debug was compiled into a machine code program.

2.1 CPU level support

The CPU can only execute machine code which is made of instructions. It also has several registers to help with computations. Which instructions and registers the CPU has can differ from CPU to CPU. This is specified by *Instruction Set Architecture* (ISA) [1]. It is an abstract interface between the hardware and the lowest level software (machine code). It contains all information needed to write a program in machine code. In general, ISA specifies the following:

- Set of machine code instructions - Specifies instructions the ISA has and what operands each instruction has.
- Register set - Which registers the ISA has[1].
- Addressing modes - Possible methods to refer to memory or register.

[1] Strictly speaking ISA doesn't have to use registers. It's possible to use only stack or accumulator, but most used ISAs use registers, so we'll ignore those architectures.

This list is not exhaustive, but for our purposes it suffices. For each instruction and operand it is specified how they should be encoded into binary. CPU then *implements* some ISA. If two different CPUs implement the same ISA then they should be able to run the same machine code program. For example, most PC use the x86-64 architecture [1], although the ARM architecture is also seeing use in personal computers, for example, the Apple Sillicon is of ARM architecture.

The x86-64 architecture is so called *Complex Instruction Set Architecture* (CISC). It contains many instructions that do many things at once, have varying lengths, and take multiple clock cycles to complete [7]. On the other hand, the ARM architecture is *Reduced Instruction Set Architecture* (RISC). The number of instructions is smaller, they are intended to be small building blocks from which complex operations may be created by using many of them. Each instruction in RISC also has the same length. Both architectures have their pros and cons, although some literature suggests that in modern days the choice of architecture is irrelevant if one is only considering performance and power consumption [8, 9]. Unless specified otherwise the rest of this chapter will be talking about x86-64. This is because the T86 architecture 3 is loosely based on x86-64, so it is most relevant for us.

In the first chapter, we briefly mentioned that machine code programs can instead be written in assembly language. Assembly is almost 1:1 mapping to machine code. When showing programs, we will show them in assembly so that they are readable. In figure 2.1, we present another example of a program that was compiled from the C programming language to assembly of the x86-64 architecture. As seen, instructions have various operands. Most often registers (`rbp, rsp, eax`), memory (`[rbp-4]` is reference to memory at address which is in register `rbp` minus 4), or labels (like `L2`). Labels are not part of machine code, instead memory address has to be provided. This is a small part where assembly and machine code differ. For a detailed overview of the x86-64 instruction set see [7].

Registers

The x86-64 architecture has a set of general purpose registers. Some of these are

- rax - Accumulator for operands and results data,
- rcx - Counter for string and loop operations,
- rsp - Stack pointer,
- rbp - Pointer to data on the stack.

Although there is a special role for each register, it is more of a convention. It is not necessary to use `rcx` only for loop operations. Although the `rsp` and `rbp` are called general purpose they are often only used for pointing at the top of the stack, resp. to the base of the stack. Stack is a special part of program memory with LIFO semantics. It can be used to store the intermediate result, arguments to functions, return address, etc. The `rbp` register is weird in a sense that it has this very special purpose but is still considered part of the general purpose registers [7]. One might use it for storing calculations, but it would make the rest of the instructions that work with stack behave unexpectedly.

The instruction pointer register (`rip`) contains the address of the current instruction to be executed. As we mentioned in the Introduction chapter, programs are executed

```
max:
    push    rbp
    mov     rbp, rsp
    mov     QWORD PTR [rbp-24], rdi
    mov     DWORD PTR [rbp-28], esi
    mov     rax, QWORD PTR [rbp-24]
    mov     eax, DWORD PTR [rax]
    mov     DWORD PTR [rbp-4], eax
    mov     DWORD PTR [rbp-8], 1
    jmp     .L2
.L3:
    mov     eax, DWORD PTR [rbp-8]
    cdqe
    lea     rdx, [0+rax*4]
    mov     rax, QWORD PTR [rbp-24]
    add     rax, rdx
    mov     eax, DWORD PTR [rax]
    cmp     DWORD PTR [rbp-4], eax
    cmovge  eax, DWORD PTR [rbp-4]
    mov     DWORD PTR [rbp-4], eax
    add     DWORD PTR [rbp-8], 1
.L2:
    mov     eax, DWORD PTR [rbp-8]
    cmp     eax, DWORD PTR [rbp-28]
    jl      .L3
    mov     eax, DWORD PTR [rbp-4]
    pop     rbp
    ret
```

Figure 2.1 Compiled C program with GCC 9.4 compiler as x86-64 assembly.

sequentially from top to bottom, with certain instructions having the ability to change the control flow. When an instruction gets executed, the size of the instruction will be added to the value in RIP register. This will advance the instruction pointer to the next instruction. Or, if the instruction changes the control flow, the value in the instruction pointer will be changed to the destination of the instruction. The register can also be changed directly.

Another interesting register is the `eflags` register. The register contains a group of flags, which can alter various behavior of the CPU, or the CPU itself sets them as a result of some instruction. For example the instruction `cmp` compares its two operands and if they are the same the *zero* flag in the `eflags` register will be set.

2.1.1 Interrupts

An interrupt is a special request to the CPU to stop the execution of the current program and to quickly react to the reason that caused the request [10]. An example of such event can be a keyboard press or an error in a program (division by zero). There are two main categories [7]

- An **interrupt** is an asynchronous[2] event that is typically triggered by an Input/Output (IO) device.
- An **exception** is a synchronous event that is generated when the processor detects one or more predefined conditions when executing an instruction. These are further divided into three classes: faults, traps, and aborts.

For the rest of this thesis, we will use the word interrupt interchargebly for both interrupt and exception although we will be mainly talking about exceptions. This is to disambigous the exceptions which are at a programming language level (like C++) and Microsoft structured exceptions about which we will talk later. When an interrupt or exception happens, the processor halts the execution of a current program and switches to a specific interrupt handler. An interrupt handler is just another sequence of instructions that gets executed when the interrupt happens and it is supposed to handle the interrupt.

An example of an exception is the `int3` instruction. When this instruction is executed an interrupt is generated. This instruction is specifically meant to be used as a breakpoint. In theory, we could supply code that will be responsible for handling the breakpoint as the interrupt handler. We could then set the `int3` instruction where we want the breakpoint to occurr and inspect the code through or interrupt handler. However, on modern PCs, an Operating System (OS) is governing the PC and consequently is responsible for setting interrupt handlers. Alas, we cannot touch the interrupt handler directly. Instead, the OS is going to have to provide another layer of support for debugging.

Recall the EFLAGS register mentioned in section 2.1. There is a special flag called the trap flag. When it is set, the CPU will issue an interrupt after every executed instruction. This could be useful if we wanted to inspect execution instruction by instruction.

The CPU may also contain special debug registers. The x86-64 architecture has six of them, named `dr0` to `dr7`, with `dr4` and `dr5` being synonyms for `dr6` and `dr7` on most CPUs, otherwise they are reserved anyway and can't be used. The CPU provides special

[2]Meaning that the interrupt may happen when another instruction is being processed (not on the CPU clock edge).

breakpoints through these registers. Each of the first four registers contains an address of the breakpoint. The debug status register `dr6` contains debug conditions that were sampled at the last debug interrupt. Lastly, the debug control register `dr7` is used to enable and disable breakpoints for each of the address registers and set breakpoint conditions. The conditions are either instruction execution on specific address, or a memory read or write. When the breakpoints are hit, for example the address stored in `dr0` is written into , the breakpoint is enabled and condition is set to memory write in the `dr7` register, an interrupt is issued by the CPU and the `dr6` register contains an information about which register caused the interrupt [7].

2.1.2 Superscalar CPUs

Modern CPUs do not execute instructions strictly one by one. They instead use *pipelines* and *out-of-order execution*. This means that they can execute several instructions at once and can execute them in non sequential order. The observable behaviour of the program has to be the same as if it was run sequentially. The CPU has to be careful about which instruction it can run in parallel and out of order. The execution of an instruction consist of several stages, which together form an *execution pipeline*. The stages can be *Fetch*, *Decode*, *Execute*, *Memory*, and *Writeback*, this may vary depending on architecture, but the general ideas behind it are the same. The instruction must pass all these stages to be successfully executed. To maximize efficiency, all stages are occupied by an instruction. This means that when instruction x leaves fetch stage, it goes to the decode stage and instruction y is put into the fetch stage.

However, when the next instruction is a conditional jump, there may be no way to be sure if the jump will be taken. CPUs deal with it in various ways and try to predict the right branch. Sometimes however, the prediction fails and the whole pipeline needs to be flushed. Same thing must happen when interrupt occurs. The execution changes to the interrupt handler, the pipeline must therefore be flushed because it contains instructions from the previous location before the control flow was switched to the handler. Also, if we hit the breakpoint, we want to make sure that instructions before it were all executed and that none was executed after, otherwise it would be very confusing for the user that tries to debug the program.

2.2 Operating system support

Operating system is a layer between computer components (CPU, memory, input/output devices, …) and software. It is responsible for handling all the resources so programmers do not have to think about it [11, 12]. Managing resources is not only to make writing programs easier but to make sure that the programs are safe from each other. Modern operating system allows to run multiple programs at once (or at least offer the illusion that it can) and they make sure that one program cannot overwrite data or otherwise interfere with other programs. Normal programs run in so-called *user space*, which has limited capabilities. The kernel on the other hand runs in *kernel space*. It has full access to the hardware of the computer, can use all instructions, can permit or mask interrupts, and so on.

However, if programs were kept to user space all the time they would be very limited. Sometimes, they need to escape the confinement of the OS, for example, to read a file or

communicate with other processes. Operating systems provide an interface through which the user space program can leverage a small part of the kernel in form of system calls. They offer a way of requiring some service from the OS. This API is often in form of C and C++ functions [12]. A part of these functions is a special instruction, like `SYSCALL` on x86-64 [7], that switches the mode to kernel space.

The most used operating systems today are Microsoft Windows, Linux, and MacOS. Linux and MacOS systems are somewhat similar, but Windows is very different. To debug programs, we need to be able to read and modify the state of a running program. This is in direct conflict with the encapsulation the operating system is trying to achieve. As such, the operating system must provide an api through which we can do these operations that are needed for debugging.

2.2.1 Linux

Linux offers a special system call which is very handy for debugging. It is called `ptrace` [13] - process trace. It has following signature: `ptrace(request, pid, void* addr, void* data)`. The request is a `PTRACE_COMMAND`, which specifies the behavior of the function (for example `PTRACE_SINGLESTEP`), pid of some process (presumably the debuggee) and two other parameters, whose meaning change depending on the `PTRACE_COMMAND` that was chosen[3]. It allows to observe and control the execution of another process, this process will be the debuggee. In the context of this chapter, we will sometimes use the word tracee for the debuggee, and tracer for the debugger, to be consistent with ptrace documentation.

`ptrace` has many commands, here are some of the most important:

- `PTRACE_PEEKTEXT, PTRACE_PEEKDATA` - Read tracee's memory,
- `PTRACE_POKETEXT, PTRACE_POKEDATA` - Write into tracee's memory,
- `PTRACE_GETREGS` - Read tracee's register values,
- `PTRACE_SETREGSET` - Modify tracee's register values,
- `PTRACE_GETSIGINFO` - Retrieve information about the signal that caused tracee to stop,
- `PTRACE_CONT` - Restart the stopped tracee process,
- `PTRACE_SINGLESTEP` - Restart the stopped tracee but stop it after executing one instruction.

Linux however needs some way of notifying the debugger that the tracee encountered a breakpoint, or that some other event requiring debugger attention happened. To this end, *signals* are used. They serve as an abstraction on top of the CPU interrupts. Interrupts are sent by the CPU and processed by the operating system kernel. Signals are sent by the operating system kernel and received by processes. They can also be sent by a process, that however happens via system calls and the kernel is the one who sends the signal, so the statement still stands.

A signal is used in UNIX and Linux systems to notify a process that a particular event has occurred [12]. When a process receives a signal, it stops its execution and starts the

[3] Peak API design!

execution of a signal handler. There are various signal types. Most signals can have a custom signal handler defined by the process. If no handler is defined then a default one is provided by the OS. However, handlers for `SIGKILL` and `SIGSTOP` cannot be changed [14].

The reason for sending a signal to a process can for example be a CPU interrupt (like division by zero or breakpoint hit) or a system call (`kill(pid, signal)`). One such signal is the `SIGTERM`, it can be sent to a process to "ask it" to exit. The process can handle this request, for example to save some state before exiting. It can also however be completely ignored. For this, a signal `SIGKILL` can be used, which cannot be handled, ignored, or blocked.

To begin tracing a command `PTRACE_ATTACH` may be used for an already existing process, or a `fork` followed with a child calling `PTRACE_TRACEME` and typically `execve`. When the tracee is being traced it will stop each time a signal is delivered to it, even if it chooses to ignore said signal. The tracer will be notified that the tracee received a signal at its next call to the `waitpid` function. When the tracee is stopped the tracer can initiate various ptrace requests listed above to inspect and change the state of the tracee [13].

In the CPU chapter, we mentioned that there are debug instructions and that they issue an interrupt. The Linux kernel however does not permit us to work with interrupt handlers, instead it translates those interrupts into signals. So when the tracee executes the `int3` instruction, it'll receive a signal that we will catch with the `waitpid` function.

2.2.1.1 Debugger implementation

Now, we have all the necessary building blocks to build a simple debugger on the Linux operating system running on the x86-64 platform. Running a program under the debugger is simple. In figure 2.2 you can see the initialization of the debugger. It uses the fork-exec idiom. The `fork` system call creates an exact copy of the process as a child except the following: process ID (PID) of the child is different. In the parent process, `fork` returns the PID of the child. In the child, 0 is returned. In the child, we initiate the `PTRACE_TRACEME` call, which indicates that this process is to be traced by its parent. Then it calls the `execve` system call, which replaces this program with the one we want to debug. The execve causes a `SIGTRAP` signal on completion because the process is being traced [15].

The parent first issues a `waitpid` system call. This waits for the child `execve` to finish. The debugger then has a chance to debug the child immediately, because it is stopped. The `while` loop can then request input from the user and act accordingly. The tracee can be continued by `PTRACE_CONT` command.

Reading and writing to memory is simple. The `PTRACE_PEEKTEXT` and `PTRACE_POKETEXT` are used for this purpose. This call reads or writes a word (32 or 64 bits, depending on the Linux variant) into tracees memory. However, if we want to write an arbitrary amount of bytes, we have to work around the word limitation. We need to write by blocks and for the last one we have to pad the write with already existing data if the block size does not divide the word size. Figure 2.3 is shown how to read and write exactly one byte of memory at a given address, which is simpler to implement but not ideal for writing many bytes.

Registers are similar to a memory in regards to how to read them. Linux contains a predefined structure `user_regs_struct`, which maps all registers on the current architecture. The command `PTRACE_GETREGS` then fills up this structure with the value in registers. To save registers, `PTRACE_SETREGS` can be used. One has to pass the whole structure, so if we want to modify only some registers we first need to fetch the structure,

```
pid_t pid = fork();
if (pid == 0) {
    // Begin tracing
    ptrace(PTRACE_TRACEME, 0, NULL, NULL);
    // Replace the code with the intended tracee code
    execve(executable, argv, NULL);
} else {
    int w;
    waitpid(pid, &w, 0);
    while(...) {
        // Main debugger loop
    }
}
```

Figure 2.2 Linux - Debugger initialization.

```
uint8_t read_memory(pid_t pid, uint64_t address) {
    uint64_t data = ptrace(PTRACE_PEEKDATA, pid, address, NULL);
    return (uint8_t)data;
}

void write_memory(pid_t pid, uint64_t address, uint8_t data) {
    uint64_t old = ptrace(PTRACE_PEEKDATA, pid, address, NULL);
    uint64_t new_data = (old & ~0xFF) | data;
    ptrace(PTRACE_POKEDATA, pid, address, new_data);
}
```

Figure 2.3 Linux - Reading and writing one byte using ptrace.

```
void set_register(pid_t pid, const char* name, uint64_t data) {
    struct user_regs_struct regs;
    ptrace(PTRACE_GETREGS, pid, NULL, &regs);
    uint64_t* reg_data = get_register(&regs, name);
    *reg_data = data;
    ptrace(PTRACE_SETREGS, pid, NULL, &regs);
}
```

Figure 2.4 Linux - Setting one register using ptrace

```
01 d0      add eax,edx
88 45 fd   mov [rbp-0x3],al
ff e0      jmp rax
```

```
01 d0      add eax,edx
cc 45 fd   int3
ff e0      jmp rax
```

Figure 2.5 Code with and without a breakpoint.

fill the registers we want with values, and then finally use `SETREGS`. An implementation can be seen in figure 2.4, the `get_register` functions maps the structure members to register name. Since the address of the current instruction to be executed is also stored in a register, we now have access to it.

Breakpoints are more interesting. Debuggers often allow to enable and disable a breakpoint, so we will distinguish between setting a breakpoint and enabling it. Setting a breakpoint just means that the debugger needs to keep track of where the breakpoint was set and if it is enabled or disabled. Enabling a breakpoint means writing into the program memory the debug instruction (we will use the x86-64 `int3` instruction with opcode `0xCC`). Figure 2.5 shows how code looks with and without enabled breakpoint. The blue color shows the new breakpoint. On line 2, the opcode changes from `0x83` to `0xCC`. Since the breakpoint is set via the `PTRACE_POKEDATA`, which only works data blocks, one has to pad the breakpoint opcode with already existing data, those are marked by the red color. Also, the rewritten data (in this case the `0x88` has to be kept so that the breakpoint may later be deactivated. Abbreviated implementation can be found on 2.6.

When a breakpoint is hit, the control is passed to the debugger, at this point, the instruction pointer would point to the next instruction, which has opcode `0x7D`, not `0xFF` (the `jmp` instruction) as might be expected. This is because `int3` is an instruction with size 1. It gets executed, the instruction pointer is advanced by the size (1) and an interrupt is issued. But after the int3 there were operands to the `mov` instruction (the red text in the figure), we definitely do not want to interpret them as instruction. We found out that on some architectures, advancing PC does not happen when an instruction that issues an interrupt is executed, such as the ARM `BKPT` instruction. This was later confirmed by looking at the LLDB code (under `NativeProcessProtocol.cpp`), as it only moves back the program counter for few architectures.

Additionally, if we just moved back and resumed the program, we would again hit the breakpoint which is still set. We need to temporarily unset it, move one instruction forward, and set it again. The general idea behind it can be found in figure 2.7. With breakpoints, we essentially gained the ability to single step as well.

However, putting a breakpoint on the instruction that will be executed next is not as easy as it might appear. Due to the varying sizes of instructions on CISC architectures,

```
void enable(pid_t pid, struct breakpoint* bp) {
    if (bp->enabled) {
        return;
    }
    bp->backup = read_memory(pid, bp->address);
    write_memory(pid, bp->address, BP_OPCODE);
    bp->enabled = true;
}

void disable(pid_t pid, struct breakpoint* bp) {
    if (!bp->enabled) {
        return;
    }
    write_memory(pid, bp->address, bp->backup);
    bp->enabled = false;
}
```

Figure 2.6 Enabling breakpoints, abbr.

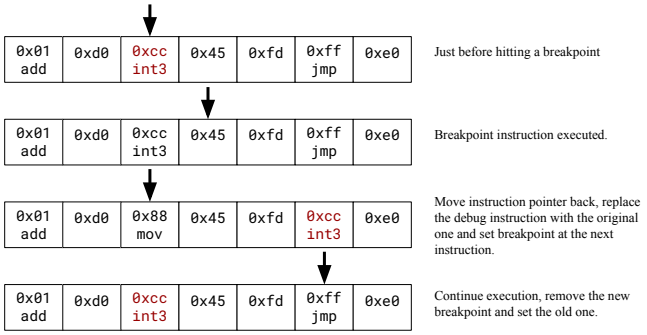

Figure 2.7 Continue sketch

```
void step_over_bp(pid_t pid) {
    uint64_t loc = read_register(pid, "rip");
    struct breakpoint* bp = find_bp(loc);
    if (bp == NULL || !bp->enabled) {
        return;
    }

    disable(pid, bp);
    ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL);
    // Waits until debuggee finishes singlestepping
    wait_for_debuggee(pid);
    enable(pid, bp);
}
```

Figure 2.8 Stepping over breakpoint, abbreviated.

even determining the instruction that follows the current one in the code is challenging. RISC architectures facilitate the identification of subsequent instructions, because they all have the same length. Finding the next instruction to be executed may still pose a challenge if the instruction is a conditional jump. Accurately determining where the jump will go can be an arduous task. To achieve this, some debuggers use an instruction emulators. Fortunately, some architectures have a built in hardware support for single stepping, like the trap flag we mentioned in section 2.1.

The ptrace library contains a special `PTRACE_SINGLESTEP` command, which executes one instruction in the tracee before sending the `SIGTRAP` signal. The Linux kernel (as of v6.1.8) uses the previously mentioned trap flag for x86-64 [16]. If the architecture does not provide a hardware supported single stepping then the call will return an error. In lieu of the breakpoint dance mentioned earlier, the singlestep command will be used in our implementation because x86-64 supports it. Implementation is described in figure 2.8. We presume that the instruction pointer was moved when the breakpoint was hit, eliminating the need to return one byte backward. This is more reasonable anyway since you want to show the user at which address was the breakpoint hit. The LLDB [6] debugger uses hardware support if there is one, else it uses an instruction emulator. For instance for the ARM architecture, LLDB uses an instruction emulator, whereas for the x86-64 it uses the trap flag.

For a step in, we can just use the `PTRACE_SINGLESTEP` command. We however need to check if the current line doesn't have a breakpoint, since that would cause the debugger to return the program counter back before it and we wouldn't be able to step over it. So, if there is a breakpoint use the code from `fig:singlestep`, else just use `PTRACE_SINGLESTEP`.

To perform a step in, the `PTRACE_SINGLESTEP` command can be utilized. However, in the case where the current line contains a breakpoint, the debugger must be cautious, as executing the single step would cause the program counter to revert back before the breakpoint[4], rendering it impossible to step over. Therefore, a check for the presence of a breakpoint must be performed prior to the execution of the single step operation. If a

[4]Remeber that we set the instruction pointer before the breakpoint when we hit it.

breakpoint exists, the code depicted in `fig:singlestep` must be utilized, otherwise the `PTRACE_SINGLESTEP` command may be directly used.

Next is line is the step out operations, which allows to jump out of the current function. When a procedure is entered using the `call` instruction, the address immediately after the call instruction is put onto the stack. Upon entry into the function, the value of `rbp` register is pushed onto the stack and then the register is updated to reflect the current value of the stack pointer. As a result, the return address is located at `rbp + 8`. We can set a breakpoint at that address, continue execution, wait for the debuggee to hit the breakpoint, and then disable and remove it. This has its flaws, for example, it won't work if the step out is used before the rbp is set, or if some library completely bypasses the call instruction.

This concludes the implementation of a very basic assembly level debugger for Linux on the x86-64 platform. The proof of concept implementation is provided as an extension to this thesis and can be found under `/linux-debugger` folder. The implementation was partially inspired by [17] and [6]. It demonstrates the feasibility of building a very basic debugger with the ptrace api.

Hardware versus Software breakpoints

ction:cpu-debug-
pport

The presented implementation has used `software breakpoints`, which are implemented by modifications to the actual code. In section we mentioned that CPU also supports breakpoints, these are called *hardware breakpoints*. The difference between these two are mostly felt when conducting reverse engineering, which involves studying compiled programs in machine code because they do not have an access to source code of the programs, often in order to identify potential malicious activity. The programs themselves however may try to defend from being debugged. Software breakpoints changes the actual code, so a checksum of instruction opcodes can detect them. Hardware breakpoints on the other hand do not modify the code and are therefore more difficult to detect. Additionaly, they can also be used to break on memory access, which is not possible with software breakpoints. However, there are limits amount of hardware breakpoints that can be set, with only four available on the x86-64 architecture.

2.2.2 UNIX systems

There are other systems based on UNIX, however, the implementation is very similar to the Linux one. For example, MacOS, FreeBSD, and OpenBSD also have the `ptrace` system call. The call itself is a little different on each operating system but it can be used to achieve the same functionality as on the Linux operating system.

ratky, asi nema cenu
y to byla subsection

2.2.3 Windows

The Microsoft Windows operating system also offers built-in support for debugging. It is provided at the Win32 API layer [18, 19]. It builds on *debug events* and *debug functions*. Unlike POSIX systems such as Linux, Windows uses a different approach for signaling abnormal conditions, such as breakpoint hits or just interrupts in general. We will explore this mechanism before getting to the debugging API.

Structured Exception Handling

Instead of signals, Windows uses *Structured Exceptions* (SEH) [20]. An exception is an event that requires the execution of code outside the normal flow of control. There are software exceptions, like throwing an exception explicitly or by the operating system, and hardware exceptions that are caused by the CPU (e.g., interrupts). SEH unifies both of these things into one, just like signals do.

When an exception is triggered, control is transferred to the system. It saves all necessary information that can later be used to continue execution from the point where the exception was thrown. It also contains information about which type of exception was thrown, if the execution can continue after handling the exception, the address where the exception occurred, and some other bits of information[5]. The system then searches for an exception handler that will handle the exception. The search is performed in this order:

1. If the process is being debugged the debugger is notified.
2. If the process is not debugged or the debugger does not handle the exception, the frame-based exception handler is to be found[6].
3. If no frame-based handler can be found, or no handler handles the exception, but the process is being debugged, then the debugger gets notified a second time.
4. The system provides a default handler, which often is the termination of the program via `ExitProcess`.

The debugger has two opportunities to handle the exception. The *first-chance* is before the exception gets to the debuggee. This is intended for exceptions for which the debugger is responsible, for example, breakpoint or single stepping. The debugger should handle these because it is responsible for them, while the debuggee is not. The second opportunity, called last-chance, is when a debuggee does not have an appropriate frame-based handler to handle the exception. If no debugger were attached, the system would have already used the default exception handler, which is often program termination. An example of an exception the debugger should not handle on first chance is `EXCEPTION_INT_DIVIDE_BY_ZERO`. It should pass it along to the debuggee. If the debuggee does not have any appropriate handler, the debugger will have a last chance to look at the program's state before it is killed [21].

Two important exception types for debugging, both of which should be handled on first-chance:

- `EXCEPTION_BREAKPOINT` - Raised when a breakpoint was encountered.
- `EXCEPTION_SINGLE_STEP` - Raised when a hardware-supported single step was completed.

Debugging events and functions

Now that we have explained how a breakpoint may be signaled, let us look at other tools that the Windows API gives us to allow debugger implementation. On Linux, we

[5] See [20] for a full list.

[6] For detailed information about the handlers see [20].

had a single function called `ptrace`. The behavior of this function changed based on its arguments. Windows, on the other hand, provides us with many functions, some of which are:

- `DebugActiveProcess` - Attaches the debugger to an active process.
- `DebugBreakProcess` - Causes a breakpoint exception to occur in the specified process. This passes control of the process to the debugger if there is one.
- `WaitForDebugEvent` - Waits for new debugging events (on Linux, we used `waitpid`, which is more a general purpose function).
- `ContinueDebugEvent` - Continue the process execution after processing a debugging event (on Linux, we used the `PTRACE_CONT`).
- `OutputDebugString` - Sends a string from the debuggee to the debugger.
- `ReadProcessMemory` and `WriteProcessMemory` - Read and modify process virtual address space. On linux we used `PTRACE_PEEKTEXT` and `PTRACE_POKETEXT`.
- `FlushInstructionCache` - Flushes the instruction cache of the process. This should be used when modifying the process text section because the old instructions may still be cached.

The figure 2.9 depicts how a communication between the debugger, the Windows operating system and the debuggee can look. The debugger waits for debug events via function `WaitForDebugEvent`. This function has a timeout parameter, so the debugger can also do other things while it is waiting, like GUI updates.

Debugging Events

Debugging events are various incidents in the debuggee that causes the system to notify the debugger [23]. These are stored in special `DEBUG_EVENT` structure, which is received in `WaitForDebugEvent` call initiated from the debugger. This structure contains various information about the event, the internals can be seen in figure 2.10. These events include loading and unloading a DLL, creating and exiting a process, sending debug strings via the `OutputDebugString`, and last but not least, exception occurence. Exceptions include the `EXCEPTION_BREAKPOINT` and `EXCEPTION_SINGLESTEP` mentioned previously, which are vital to the debugger.

Tying it all together

Now we have all the necessary building blocks to create a tiny Windows debugger. The implementation itself would be very similar to the Linux one we presented in section 2.2.1. To provide at least some idea, figure 2.11 contains an abbreviated main loop of the debugger [24]. We need to handle all of the debugging events if we want the debugger to be robust. For example, thread creation should be handled so that we may trace all threads spawned by the debuggee.

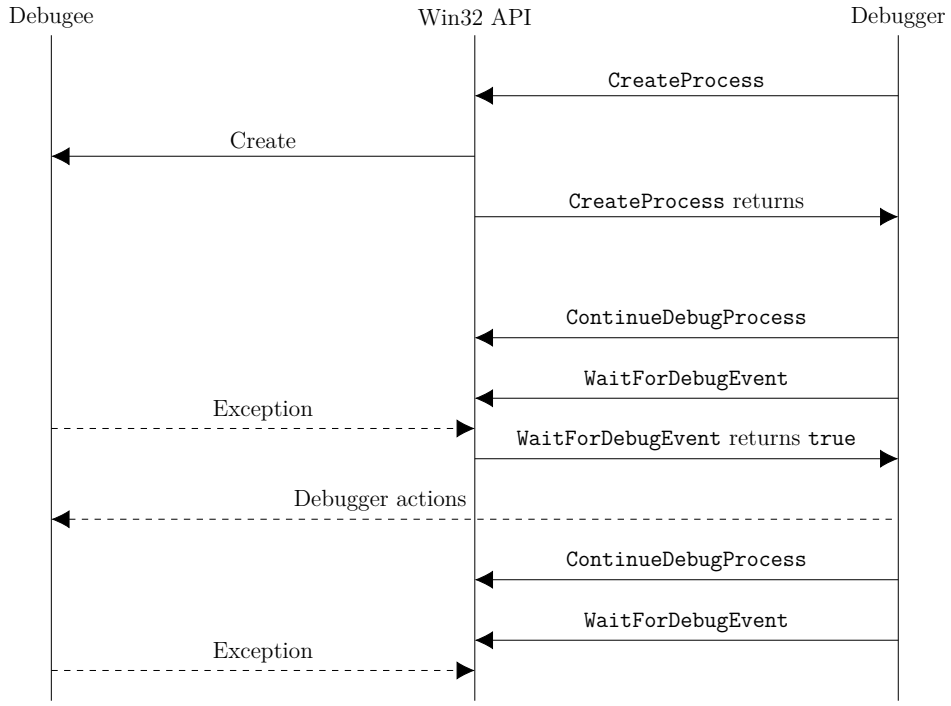

Figure 2.9 A diagram representing the communication between the debuggee, the Windows operating system and the debugger [22].

```
typedef struct _DEBUG_EVENT {
  DWORD dwDebugEventCode;
  DWORD dwProcessId;
  DWORD dwThreadId;
  union {
    EXCEPTION_DEBUG_INFO      Exception;
    CREATE_THREAD_DEBUG_INFO  CreateThread;
    CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
    EXIT_THREAD_DEBUG_INFO    ExitThread;
    EXIT_PROCESS_DEBUG_INFO   ExitProcess;
    LOAD_DLL_DEBUG_INFO       LoadDll;
    UNLOAD_DLL_DEBUG_INFO     UnloadDll;
    OUTPUT_DEBUG_STRING_INFO  DebugString;
    RIP_INFO                  RipInfo;
  } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

Figure 2.10 A structure which contains an information about debugging event.

```
void EnterDebugLoop(const LPDEBUG_EVENT DebugEv)
{
   DWORD dwContinueStatus = DBG_CONTINUE; // exception cont.
   for(;;)
   {
      WaitForDebugEvent(DebugEv, INFINITE);
      switch (DebugEv->dwDebugEventCode)
      {
            case EXCEPTION_ACCESS_VIOLATION:
               // First chance: Pass this on to the system.
               // Last chance: Display an appropriate error.
                  break;

               case EXCEPTION_BREAKPOINT:
               // First chance: Display the current
               // instruction and register values.
                  break;

              // Other exception types like singlestep ...
               ...
          }
          // Other debug events
          ...
      }
   ContinueDebugEvent(DebugEv->dwProcessId,
                      DebugEv->dwThreadId,
                      dwContinueStatus);
   }
}
```

Figure 2.11 An abbreviated example of a Windows Debugger loop, taken from [24].

2.3 Source Level Debugging

If we want to debug at the source code level, we must provide the debugger with additional information because otherwise, it can only work with the generated machine code. For instance, we somehow need to tell the debugger that the instruction at address *0xABCD* belongs to line `5` in the source code so that we may properly set a breakpoint at this line. We also need to provide information about where some variable is located so that we may display its value to the user. Before we get into that, we will briefly examine the layout of executables and where those pieces of debugging information are stored.

2.3.1 Executable file formats

ELF is one of the executable file formats. The abbreviation ELF means Executable and Linkable format. The file itself consists of sections, some of which are

- `.text` - the executable code.
- `.data` - allocated space and values of initialized static[7] variables and other objects.
- `.bss` - allocated space for uninitialized static variables,
- `.rodata` - data that doesn't change for the entirety of the program lifetime, like strings literals in C.
- various debug sections like `.debug_info`, `.debug_line` etc.

For inspecting the ELF files, commands `readelf` and `objdump` may be used on Linux systems. The ELF format also has a header, and all sections have their header too. This contains information like the name of the section, the size of the section, etc. The ELF format is not only used for executables, but also for *relocatable* file, *executable* file and *shared object* file [25]. The sections themselves are often protected. For example, the `.text` section can only be read and executed, but cannot be written into. The `.data` section can be read and written to, but cannot be executed. The ELF also contains various debug sections, so the debugging information is embedded in the executable itself.

ELF is not the only one executable format. Others include the *COFF* and *PE* format, which are used by Microsoft Windows, or the *Mach-O*, which is used by the MacOS operating system. All of these formats use some kind of encoding for the debugging information. ELF and Mach-O use the DWARF encoding.

2.3.2 DWARF

DWARF [26] is a debugging information format used to describe programs written in procedural languages. It is primarily associated with the ELF file format. It aims to support all different types of information that may be needed while debugging. It also strives to encode this information in as much space efficient manner as possible [26]. This can make it somewhat complex.

There are five versions of the standard, we will be mainly talking about version 2, since the others mainly add features to accommodate newer language features. We will

[7] In this context, static means that the variable exists for the entire lifetime of the program.

not get that far even with version 2. The complete changelog can be found in the text describing DWARF version 5 [27]. We provide many examples of the information that DWARF encodes. For reading the DWARF information from object files, we used the `llvm-dwarfdump`, `objdump`, and `readelf` tools to construct these examples. The information DWARF encodes is stored in various debug sections if used in conjunction with the ELF standard. The Mach-O format stores debugging information into separate file. This file is still however split into several sections.

ould I cite all of ese guys?

2.3.2.1 Line Number Information

Line number information needs to convey which line of code corresponds to an address of a machine code instruction. The DWARF standard [26] mentions that encoding this information would be possible in a matrix, which would have one row for each instruction. The columns of the matrix would contain

- the source file name,
- the source line number,
- the source column number,
- whether this instruction is the beginning of a source statement,
- whether this instruction is the beginning of a basic block[8].

They however argue that such a matrix would be very big. The matrix is therefore stripped of redundant information. For instance, instruction whose row would be the same as the previous one is not saved into the matrix. This still was not enough. Instead of a matrix, they provide a virtual machine specification and a bytecode language that one has to interpret to reconstruct this matrix and consequently to get the necessary debugging information.

The virtual machine consists of following registers: `address`, `file`, `line`, `column`, `is_stmt`, `basic_block`, and `end_sequence`. The program itself begins with a prologue. This prologue contains the length of the program, version, length of the prologue, length of instructions (so that it may be better compressed), and similar pieces of information.

Then the *Standard opcodes* are provided, which are instructions of the virtual machine. These are mostly used for manipulating the registers of the machine or to create a new row from the values in the registers and append it to the matrix [26].

It also has several *Special opcodes*, each of which does all of the following operations:

- Add a signed integer to the `line` register.
- Multiply an unsigned integer by the smallest length of instruction and add the result to the *address* register.
- Append a row to the matrix consisting of the current values in registers.
- Set the `basic_block` register to false.

[8] Basic block is a sequence of instructions that is entered only at the first instruction and exited only at the last instruction [26]. In other words, all of the instructions in the basic block are executed sequentially.

All of these special opcodes always do these four operations, they only differ in what values they add to the `line` and `address` register [26]. Those values are calculated from the instruction opcode, which ranges from 10 to 255 in DWARF version 2. This is the most common series of operations the virtual machine must do. Encoding it into single opcode makes the program extremely size efficient.

Observe the program in figure 2.12. A simple greeter-like program that displays the name given to it as an argument. With the `readelf` utility program, we can inspect the DWARF bytecode language describing the locations, it is showed in figure 2.13. First few lines of the example are explained below, we ignore the set column operations which are only used to make the location more precise.

- `0x41` - Jump to the beginning of the `main` function (i.e. `0x1139`).
- `0x4c` - Special opcode - Increment line by one and advance address by zero. Although the advance address was essentially a noop it was still more space efficient to use special opcode, because the `advance_pc` instruction DWARF offers would take up at least two bytes (the opcode and the operand), whereas the special opcode occupies only one.
- `0x4f` - Special opcode - Increment line by 2 to 3 and advance address by 15 to `0x1148`. This essentialy skips the function prologue and advances to instruction `cmpl`, which represents line `if (argc < 2)` of the program.
- `0x52` - Special opcode - advance line by 1 to 4 and advance address by 6 to `0x114e`. This is the `return 1;` line of the program.

Remember that each of the special opcode also creates a line in the matrix. As said, this is extremely space efficient. DWARF stores this encoded information in the `.debug_line` section.

2.3.2.2 Debugging Information Entry

The *Debugging Information Entries*, or just entries, are the building blocks of debugging information in DWARF. Each entry has a tag and a series of attributes. The tag specifies the class to which an entry belongs, like `DW_TAG_subprogram`, which is used for functions, or `DW_TAG_variable`, which is used for variables. A complete list can be found in [26]. Those entries can be found in the `.debug_info` section [26].

The attributes themselves then convey some property of the entry, like the name and the type of a variable or starting and ending address of a function. The entries form a tree-like structure. Each entry is owned by one parent (excluding the top entry) and can own multiple entries. These relations somewhat mimic the relations of the program structures. For instance, variable is owned by a function in which it was defined. There are also other relations among the entries, not only ownership. With those relations in place, the relation is a graph, not just a tree.

An example of DWARF entries can be seen in figure 2.14. The `DW_TAG_pointer_type` is simple, its attributes are only the size of the type and which type it points to, this is encoded as an address to another entry. The type it points to is another entry. It only contains information that its a const type, and again has a link to another entry. Finally, the entry at `0x6b` has a base type. It contains the size of the type, name, and encoding.

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
    if (argc < 2)
        return 1;
    const char* name = argv[1];
    printf("Hello, %s!\n", name);
    return 0;
}
```

```
0000000000001139 <main>:
    1139:   55                      push   %rbp
    113a:   48 89 e5                mov    %rsp,%rbp
    113d:   48 83 ec 20             sub    $0x20,%rsp
    1141:   89 7d ec                mov    %edi,-0x14(%rbp)
    1144:   48 89 75 e0             mov    %rsi,-0x20(%rbp)
    1148:   83 7d ec 00             cmpl   $0x1,-0x14(%rbp)
    114c:   7f 07                   jg     1155 <main+0x1c>
    114e:   b8 01 00 00 00          mov    $0x1,%eax
    1153:   eb 2c                   jmp    1181 <main+0x48>
    1155:   48 8b 45 e0             mov    -0x20(%rbp),%rax
    1159:   48 8b 40 08             mov    0x8(%rax),%rax
    115d:   48 89 45 f8             mov    %rax,-0x8(%rbp)
    1161:   48 8b 45 f8             mov    -0x8(%rbp),%rax
    1165:   48 89 c6                mov    %rax,%rsi
    1168:   48 8d 05 95 0e 00 00    lea    0xe95(%rip),%rax
    116f:   48 89 c7                mov    %rax,%rdi
    1172:   b8 00 00 00 00          mov    $0x0,%eax
    1177:   e8 b4 fe ff ff          call   1030 <printf@plt>
    117c:   b8 00 00 00 00          mov    \$0x0,%eax
    1181:   c9                      leave
    1182:   c3                      ret
```

Figure 2.12 A program in the C programming language and the generated machine code by a GCC compiler.

```
[0x3f] Set column to 34
[0x41] Extended opcode 2: set Address to 0x1139
[0x4c] Special opcode 6: advance Address by 0 to 0x1139 and Line by 1 to 2
[0x4d] Set column to 8
[0x4f] Special opcode 216: advance Address by 15 to 0x1148 and Line by 1 to
    3
[0x50] Set column to 16
[0x52] Special opcode 90: advance Address by 6 to 0x114e and Line by 1 to 4
[0x53] Set column to 17
[0x55] Special opcode 104: advance Address by 7 to 0x1155 and Line by 1 to 5
[0x56] Set column to 5
[0x58] Special opcode 174: advance Address by 12 to 0x1161 and Line by 1 to
    6
[0x59] Set column to 12
[0x5b] Advance PC by constant 17 to 0x1172
[0x5c] Special opcode 146: advance Address by 10 to 0x117c and Line by 1 to
    7
[0x5d] Set column to 1
[0x5f] Special opcode 76: advance Address by 5 to 0x1181 and Line by 1 to 8
[0x60] Advance PC by 2 to 0x1183
[0x62] Extended opcode 1: End of Sequence
```

Figure 2.13 A DWARF description of mapping source lines to addresses for program in figure 2.12.

We also have the `DW_TAG_SUBPROGRAM`, which represents a function. It has many attributes, like name, path to the file where it was declared, on which line and column it is located, at which machine code address the function begins and ends, and so on. This entry also owns some other entries (indicated by indentation). Here, the children are the parameters and variables of the function. Important information about variables is where they are stored, so that we may look up their value. We can also see that the variable `name` has an attribute type, which points to the entry at address `0x8f`, which is the entry we were talking about previously when examining types.

2.3.2.3 Locations of Variables

Locations are one of the most complicated type of debugging information in the DWARF standard. Like with line mapping, described in section 2.3.2.1 a virtual machine is needed to decode this information. Variables are generally stored in one place. This can be a memory, register, or stack. Sometimes, however, the location of an object can change throughout its lifetime, it may not be present at all since it was optimized away, it can be split into several locations, or it might be a reference to some other object. The fact that an object changes location throughout its lifetime seldom happens [28], but DWARF is nevertheless prepared for it [26].

Contrary to the virtual machine described in the line encoding section, which was register-based, this one is stack-based. This means that most operations take their operands from the stack and push the result back onto the stack. More about stack-based virtual machines can be discovered in [29]. DWARF calls those instructions *expressions* [26]. The last value on the top of the stack after the VM runs is considered to be the

```
0x6b: DW_TAG_base_type
        DW_AT_byte_size (0x01)
        DW_AT_encoding (DW_ATE_signed_char)
        DW_AT_name ("char")

0x72: DW_TAG_const_type
        DW_AT_type (0x0000006b "char")

0x8f: DW_TAG_pointer_type
        DW_AT_byte_size (8)
        DW_AT_type (0x00000072 "const char")

DW_TAG_subprogram
  DW_AT_external    (true)
  DW_AT_name        ("main")
  DW_AT_decl_file   ("/home/gregofi1/tmp/main.c")
  DW_AT_decl_line   (2)
  DW_AT_decl_column (0x05)
  DW_AT_prototyped  (true)
  DW_AT_type        (0x00000058 "int")
  DW_AT_low_pc      (0x0000000000001139)
  DW_AT_high_pc     (0x0000000000001183)
  DW_AT_frame_base  (DW_OP_call_frame_cfa)
  DW_AT_call_all_tail_calls (true)
  DW_AT_sibling     (0x000000e0)

    DW_TAG_formal_parameter
      DW_AT_name        ("argc")
      DW_AT_decl_file   ("/home/gregofi1/tmp/main.c")
      DW_AT_decl_line   (2)
      DW_AT_decl_column (0x0e)
      DW_AT_type        (0x00000058 "int")
      DW_AT_location    (DW_OP_fbreg -36)

    DW_TAG_formal_parameter
      DW_AT_name        ("argv")
      ...

    DW_TAG_variable
      DW_AT_name        ("name")
      DW_AT_decl_file   ("/home/gregofi1/tmp/main.c")
      DW_AT_decl_line   (5)
      DW_AT_decl_column (0x11)
      DW_AT_type        (0x0000008f "const char *")
      DW_AT_location    (DW_OP_fbreg -24)
```

Figure 2.14 A part of DWARFs DIEs for program in figure 2.12

resulting value (this can be an address of an object, value of an array bound, length of dynamic string, and so on) [26].

For example in the figure 2.12 there is the `DW_OP_fbreg -24`, this pushes value at offset -24 by the register specified by a descriptor in `DW_AT_frame_base`, which often is a base pointer to the stack. And since there are no more expressions the computed value on the top of the stack is the resulting location of the variable that the debugger will get to use.

There are many types of expressions, like arithmetic operations, stack manipulation, control flow operations, and last but not least, the specifications of the location. We provided already the example with the base pointer to the stack, let us look at some more examples, mostly taken from [26]:

- `DW_OP_reg3` - The value is in register 3.
- `DW_OP_addr 0xABCD` - The value of a static variable is at address `0xABCD`.
- `DW_OP_bregx 54 3; DW_OP_deref` - Adds 54 to value in register 3. This should yield an address. The value at the location to which the yielded address points is then pushed onto the stack. This can be used for recording information about references and pointers.
- `DW_OP_reg3; DW_OP_dup; DW_OP_add` - Fetches value from register 3 and pushes it onto the stack, duplicates that value and pushes it on the stack, and pops two values from the stack, adds them together, and pushes them back onto the stack[9].

With this computational power, we can save any information we need. There are many more instructions, all of which can be found in [26]. An example of an interpreter for this can be found in the LLDB source code in file `source/Expression/DWARFExpression.cpp`.

2.3.2.4 Call frames

Debuggers often need to know through which function calls the executed program went to arrive at a certain state [26]. For each function call, there is a *stack frame* saved on the stack. Stack frame is a part of the stack memory that contains information relevant to the function invocation. An example of a stack frame can be seen in figure 2.17. Stack frames mostly consist of the following:

- Return address - This contains the location from which the program was called so that after the function finishes it may properly return to that location. This fact was also used in our step-out implementation in 2.2.1.
- Base pointer backup - Backup of the value in the base pointer which was set by the previous function. This register needs to be preserved between calls, that's why a backup is made. Other registers might need to be saved for the same reason.
- Allocated space for variables - A space for local variables.

Stack frames are described in more detail in [2], where they are called activation records. We also provide an example of how a function prologue and epilogue look in figure 2.15, so one may get a better feel about how such stack frame comes to fruition.

[9] Artificial example just to show the power of the language.

```
0000000000001183 <foo>:
0x1183: push   %rbp           ; Save old stack frame pointer
0x1184: mov    %rsp,%rbp
0x1187: sub    \$0x18,%rsp    ; Allocate space on the stack
                                 <body ...>
0x11b1: leave                 ; deallocate and restore regs
0x11b2: ret
```

Figure 2.15 A prologue and epilogue of a function in assembly.

Debuggers need to do a so-called stack unwinding. Unwinding essentially means going to a state where the previous stack frame is the most recent. This needs to be done several times to get the whole chain of function calls which led to the current state of the program. We can get the last stack frame by looking at the current value of base pointer. But to get to the penultimate one is not easy. We need to know exactly where was the base pointer and possibly other registers saved so that we may simulate the unwinds [26].

DWARF proposes a table, in each row, there is an address of a location in the program, a location of a stack frame, and values of other registers for that location. However, such table would be very large, so DWARF again uses a virtual machine for the construction of the table, since many columns in one row would be have the same values in the following rows [26]. The location of a stack frame can be described either by a register and a signed offset which are added together, or a DWARF expression may be used (the ones used to describe variable locations).

Figure 2.16 contains an example of the encoding of this information and also the decoded values for the program in figure 2.15. The `advance_location` adds a new row to the matrix, with the only thing changing from the previous row being the location value. The `def_cfa_offset` specifies that the CFA (current stack frame) is at the provided offset from the same register it was offseted in previous row. We can see that in the example on line 2 the offset changes from 8 to 16, but the register stays the same. This is apparent in the decoded information where on the address `0x1184` the `CFA` changed from `RSP + 8` to `RSP + 16`. This happened because the `push %rbp` advanced the stack pointer by eight bytes.

On the contrary, the `def_cfa_register` changes the register but keeps the offset. This happens when the instruction `mov %rsp, %rbp` is executed. From that point the stack frame is at sixteen byte offset from the base pointer. This holds until the epilogue, where the old stack pointer is restored. In the example in the decoded information, many more registers were missing. This is because their value didn't change across the function, so there was no need to save information about them.

The DWARF offers more instructions for manipulation, for a complete overview and another example consult [26]. With this information, DWARF can describe a way to simulate a stack unwinding without touching the state of the debuggee.

2.3.2.5 Other informations

The DWARF standard has information about compilation units, since object files may be derived from multiple compilation units [26]. It also has information about subroutines and inlined subroutines, lexical blocks, try-and-catch blocks, and many other constructs.

```
... FDE cie=00000000 pc=00001183...000011b3
DW_CFA_advance_loc: 1
DW_CFA_def_cfa_offset: +16
DW_CFA_offset: RBP -16
DW_CFA_advance_loc: 3
DW_CFA_def_cfa_register: RBP
DW_CFA_advance_loc: 43
DW_CFA_def_cfa: RSP +8

0x1183: CFA=RSP+8: RIP=[CFA-8]
0x1184: CFA=RSP+16: RBP=[CFA-16], RIP=[CFA-8]
0x1187: CFA=RBP+16: RBP=[CFA-16], RIP=[CFA-8]
0x11b2: CFA=RSP+8: RBP=[CFA-16], RIP=[CFA-8]
```

Figure 2.16 A DWARF stack frame information about the program in 2.15. Top part contains the encoded program for virtual machine, bottom part contains the decoded table from interpreting the program. Abbreviated.

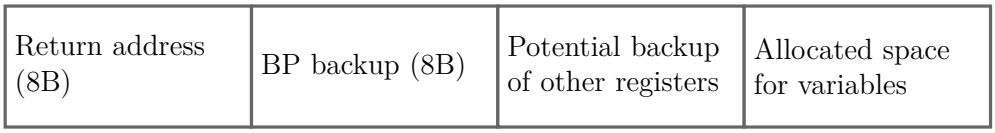

Figure 2.17 Basic stack layout after function prologue, disproportional.

These are not special in any way, we have seen a small example in figure 2.14. They mainly store the beginning and end and some other information. They also contain various information about macros, so that the debugger can work with the original code before the macro expansion.

It also contains many type entries. Those are split into different categories, we present some of the most common:

- Base types - Provided by the language, like `int`, `char`.
- Modifier types - Modifies some existing type, like type qualifiers from C (`const`, `volatile` and `mutable`), or pointers.
- Array types - A sequential storage space containing objects of the same type. Multidimensional arrays are considered a special type. An interesting attribute that a multidimensional array has is ordering - either column-wise or row-wise. This is not relevant to C, since it has no notion of a multidimensional array.
- Structure, Union, and Class types - User-defined types that pack together several different types. Information is provided about the members, especially the bit offset at which they reside in the structure and in which order were they declared. In addition, the size of the type is specified - if it can be determined at compile time (In C, it cannot be determined for forward declarations). For C++, information about inheritance, like deriving, accessibility, and if the inheritance is virtual, is provided.
- Enumeration type.

```
0x138:   DW_TAG_subroutine_type
            DW_AT_type  (0x00000131 "int")
            DW_AT_sibling   (0x14c)

0x141:     DW_TAG_formal_parameter
              DW_AT_type    (0x131 "int")

0x146:     DW_TAG_formal_parameter
              DW_AT_type    (0x14c "double")

0x153:   DW_TAG_pointer_type
            DW_AT_byte_size (8)
            DW_AT_type  (0x138 "int (int, double)")

0x124:   DW_TAG_variable
            DW_AT_name  ("xx")
            DW_AT_type  (0x153 "int (*)(int, double)")
            ...
```

Figure 2.18 Example of subroutine type and pointer to function in DWARF, abbreviated.

- Subroutine type - Used for types of functions in C. However, for the functions themselves, their type is specified as return type. Types of their parameters are specified in the parameters themselves since they are DIEs and are children of the function. We showed this in the figure 2.14. This subroutine type is for example used if one would create a pointer to a function. Then the type of that is a pointer to the subroutine type. An abbreviated example can be seen on 2.18.

We have described the most interesting parts of the DWARF standard. But all this information must be created by a compiler so that the program may be debugged at source level. In the next section, we will briefly explore compilers and how they handle debugging information.

2.4 Compilers

The compiler is a program that transforms the source code into some other form, most often into machine code or assembly. Compilers are very complicated pieces of software. Often, they are made of several parts, the general structure can be seen in figure 2.19 [2]. We will briefly explore these stages.

It all starts with the *lexical analysis*, which groups separate symbols into groups called tokens. For example the code

```
foo = bar(1 + 2);
```

might be translated into tokens like this

```
<id:"foo"> <assignment-operator> <id:"bar">
<left-bracket> <int-number:1> <plus-operator>
<int-number:2> <right-bracket> <semicolon>
```

The *Syntactic analysis* then works with these tokens and transforms them into some other representation. This is most often an abstract syntax tree (abbr. AST, figure 2.20).

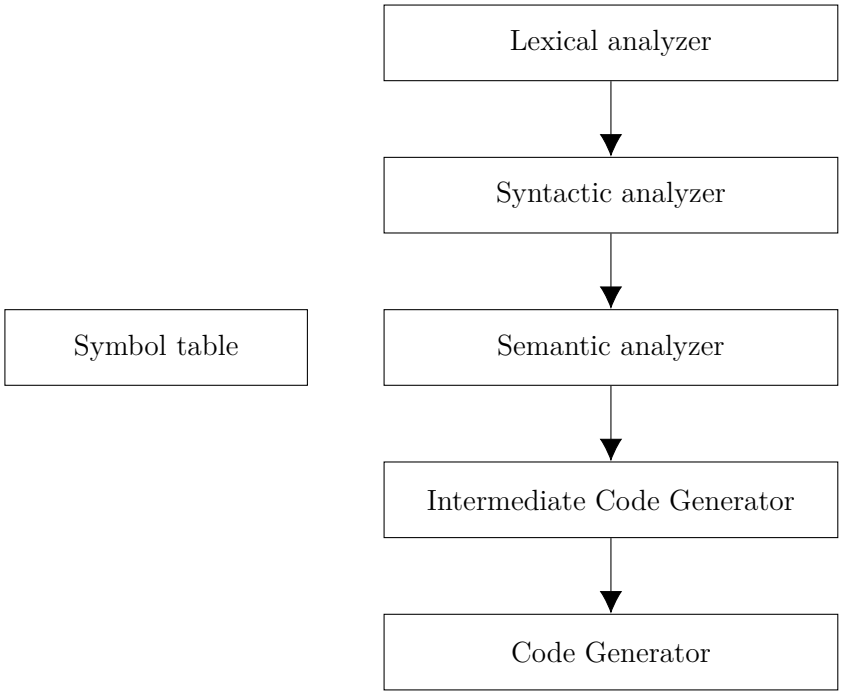

Figure 2.19 Simplified structure of a compiler, some parts were left out, like optimizations.

It also checks that the source code complies with the grammar of the language. With AST, we already lose the locations of the language constructs, so for debugging information to work it needs to be kept in the AST together with the nodes.

The *Semantic analysis* then checks that the program is semantically consistent. For example that used variable has been declared before.

After that, intermediate code generation comes into play. It converts the AST into some other representation, most commonly called IR, which means intermediate representation. IR should be closer to the machine code to be easily translated but is still machine independent. It also should retain some high-level properties that make it easier to work with, like types and functions, but should be language-independent. There are many types of IR. One of the most popular compilers, LLVM, uses single static assignment (SSA) and satisfies all of the above [30]. An example of LLVM IR can be found in figure

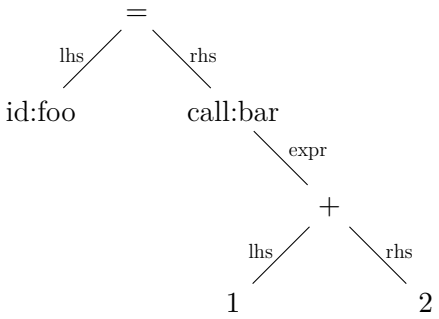

Figure 2.20 Simplified example of an abstract syntax tree.

```
define dso_local i32 @_Z6squarei(i32 %0) {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = load i32, i32* %2, align 4
  %5 = mul nsw i32 %3, %4
  ret i32 %5
}
```

Figure 2.21 Simplified example of LLVM IR.

2.21. Compilers perform most optimizations on this intermediate representation.

Finally, IR is translated directly to the target machine code or possibly assembly. This stage is called *code generation*. Even though IR can seem very similar to assembly, there are still some things to take care of. For example, SSA IR doesn't have registers. It uses an unlimited number of variables. Other architectures might have some other traits that differentiate them from the IR, and they all have to be accounted for when generating code.

2.4.1 Modularity of compilers

The main advantage of using an IR is that there is a common ground for every language. Imagine we write a compiler for the C language. We need to write all five parts from figure 2.19. If we later decided that we also wanted to create a compiler for Haskell, we just need to write everything up to the IR translation. Once we can translate Haskell into the IR, we can reuse the previous part of the compiler to compile to machine code! This also works the other way around. If we compile IR to the machine code that works with the x86-64 architecture and we want to compile to ARM, we need to create the code generation part for the ARM architecture. There is no need to write the whole compiler. Also, most optimizations are done on the IR level, saving a lot of development time. The parts of the compiler which are dependent on the source language are called **frontend** (Syntax, Semantic, and IR translation), and the parts that are dependent on the target are called **backend** (Code generation).

This is widely used in practice. The LLVM [30] project is a compiler backend. It uses its own IR (as was mentioned in figure 2.21). It can compile this IR into many targets, including x86-64, ARM, and Spark. The *Clang* project is a compiler frontend for C, C++, and Objective-C languages. It translates these languages to the LLVM IR. Other frontends for LLVM also include *ghc*, which is a Haskell compiler, or *rustc*, which is a Rust compiler. With LLVM, creating a new programming language comes down to just writing the front end.

2.4.2 IR Debugging information

With IR translation, some of the information from the AST is stripped away. The IR must encode how some of the properties of the AST map onto the IR. We will give a glimpse of how LLVM IR handles this.

Since the LLVM IR is transformed into many existing architectures and many very different existing programming languages are compiled into it, LLVM debug information does not put any restrictions on the flavor of the programming language. A high priority of the LLVM debugging information (LDI) is to make it interact well with optimizations [31]. The LDI guarantees the following:

1. LDI always provides information to accurately **read** the source-level state of the program regardless of which LLVM optimizations have been run [31]. Emphasis has been put on the word read, because it is not always possible to, for example, change the value of a variable because it might have been completely optimized away.

2. LDI does not prevent optimizations from happening.

3. LLVM optimizations may be upgraded to be aware of debugging information. This is sometimes necessary, especially if the optimizations are aggressive [31]. LLVM provides a guide to developers of optimization passes specifiying how to handle certain situations [32]. For example, an accurate stack trace is desired even though inlining or tail call optimization was applied [31].

4. The LDI is optimized with the rest of the program (i.e., unused information is removed).

So, if one compiles the code with optimizations and with debugging information, the same optimizations must be applied that would be applied if no debugging information was generated (running `clang -O3 -g` and `clang -O3` must result in the same optimizations applied). The code must also be debuggable with optimizations enabled and provide correct information. For example, suppose a variable is completely optimized out. In that case, the debugger may present the message `optimized out`, which reduces the amount of debug information available to the developer. However, the information is still correct and not misleading. Displaying an invalid value must not happen, even with optimizations [31].

In figure 2.22, we provide an example of LLVM IR with debugging information. It is encoded in the form of metadata, starting with an exclamation mark. LLVM IR also provides intrinsic functions to register debugging information about variables. The function from the example `llvm.dbg.declare` receives the address of the variable, the description of the variable, and its location in the form of a complex expression. This is similar to the Dwarf expressions used for locations described in section 2.3.2.3. There is also the `llvm.dbg.address`, which has the same role but can be used multiple times for one variable. This is primarily used after optimizations to describe how a variable's address changes throughout the program.

The debugging information in the figure is otherwise mostly self-explanatory. The variables and locations also have their scope included because LLVM IR does not have any notion of scoping. LLVM provides *consumers*, which are able to transform the LLVM IR debugging information into some other format, like DWARF.

There are some more nuances that need to be dealt with, especially considering optimizations like instruction scheduling and register selection. This is beyond the scope of this thesis and is described in [31].

```
define dso_local i32 @main() #0 !dbg !10 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  call void @llvm.dbg.declare(metadata ptr %2,
        metadata !15, metadata !DIExpression()), !dbg !16
  store i32 2, ptr %2, align 4, !dbg !16
  call void @llvm.dbg.declare(metadata ptr %3,
        metadata !17, metadata !DIExpression()), !dbg !19
  store i32 1, ptr %3, align 4, !dbg !19
  %4 = load i32, ptr %2, align 4, !dbg !20
  %5 = load i32, ptr %3, align 4, !dbg !21
  %6 = add nsw i32 %4, %5, !dbg !22
  ret i32 %6, !dbg !23
}

declare void @llvm.dbg.declare(metadata, metadata, metadata) #1

!0 = distinct !DICompileUnit(language: DW_LANG_C99,
        file: !1, producer: "clang version 15.0.7",
        isOptimized: false, runtimeVersion: 0,
        emissionKind: FullDebug, splitDebugInlining: false,
        nameTableKind: None)
!1 = !DIFile(filename: "main.c", directory: "/home/gregofi1/dev",
        checksumkind: CSK_MD5,
        checksum: "408b866a5672c0f12dbc2c9bf3baaa58")
!2 = !{i32 7, !"Dwarf Version", i32 5}
!3 = !{i32 2, !"Debug Info Version", i32 3}
!10 = distinct !DISubprogram(name: "main", scope: !1,
        file: !1, line: 1, type: !11,
        scopeLine: 1, spFlags: DISPFlagDefinition,
        unit: !0, retainedNodes: !14)
!11 = !DISubroutineType(types: !12)
!13 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
!15 = !DILocalVariable(name: "x", scope: !10, file: !1, line: 2, type: !13)
!16 = !DILocation(line: 2, column: 9, scope: !10)
!17 = !DILocalVariable(name: "y", scope: !18, file: !1, line: 4, type: !13)
!18 = distinct !DILexicalBlock(scope: !10, file: !1, line: 3, column: 5)
!19 = !DILocation(line: 4, column: 13, scope: !18)
!20 = !DILocation(line: 5, column: 16, scope: !18)
!21 = !DILocation(line: 5, column: 20, scope: !18)
!22 = !DILocation(line: 5, column: 18, scope: !18)
!23 = !DILocation(line: 5, column: 9, scope: !18)
```

Figure 2.22 An example of LLVM IR with debugging information, abbreviated.

2.5 Existing debuggers and disassemblers

In this section, we will focus on existing debugger implementations and what each brings to the table. We will not be discussing every feature of said debuggers since even some of the manuals are longer than this thesis[10], but point out the ones that were interesting to this thesis. Each debugger discussed here implements, at the very least, breakpoints, single-stepping, reading and writing into program memory and registers.

2.5.1 GDB

The GNU Project debugger (GDB) [33] is a source and instruction-level debugger. It supports languages like C, C++, D, Fortran, Go, Objective-C, Pascal, Rust, and some others. GDB is written in C/C++. It offers a CLI application through which one might interact with the debugger. It also has an API for Python.

An interesting feature GDB offers is *reverse debugging*. This allows us to go back in the execution flow. For example, the command *reverse-next* returns to the previous line. A command to record executed instructions must be explicitly invoked. GDB then sets the state of the program depending on the record. It still is not fully supported, and from our limited testing, cannot handle even some basic statements, like the usage of `printf` in C.

2.5.2 IDA Free

The primary role of IDA is to be an interactive disassembler, but it also offers some instruction-level debugging capabilities. It is a tool that is mainly used for reverse engineering. An example of interactivity is that it is possible to rename a location like `[RBP + 4]` to `counter` to make the assembly more readable. It can also follow references (like jumps or calls), create a control flow graph from the assembly, and much more. We show an example of an iterative factorial function disassembled via IDA in figure 2.23. In the figure, the variable locations were renamed to represent their semantic meaning. This has to be done by hand. IDA itself can sometimes know that it is a variable but not its exact purpose. IDA is an invaluable asset in reverse engineer's toolkit for static analysis.

2.5.3 Microsoft Visual Studio Debugger

Microsoft Visual Studio is an Integrated Development Environment. It packs together many tools that help develop programs for the Windows Operating System. It mainly supports the C# and C++ languages. It also has a built-in debugger. It is both a source and assembly-level debugger.

In figure 2.24, we show an example of how the GUI looks. On the left side, we can see the program's source code. The red points are the breakpoints. The arrow signalizes the current point in the execution. On the right side, we can see the corresponding assembly. The assembly contains the lines from the source code to make the mapping between them more apparent. The value and type of every variable in the scope are displayed on the bottom left. A call stack is shown in the middle, currently only having main and factorial

[10] Most notably GDB manual [33], which has whooping 930 pages!

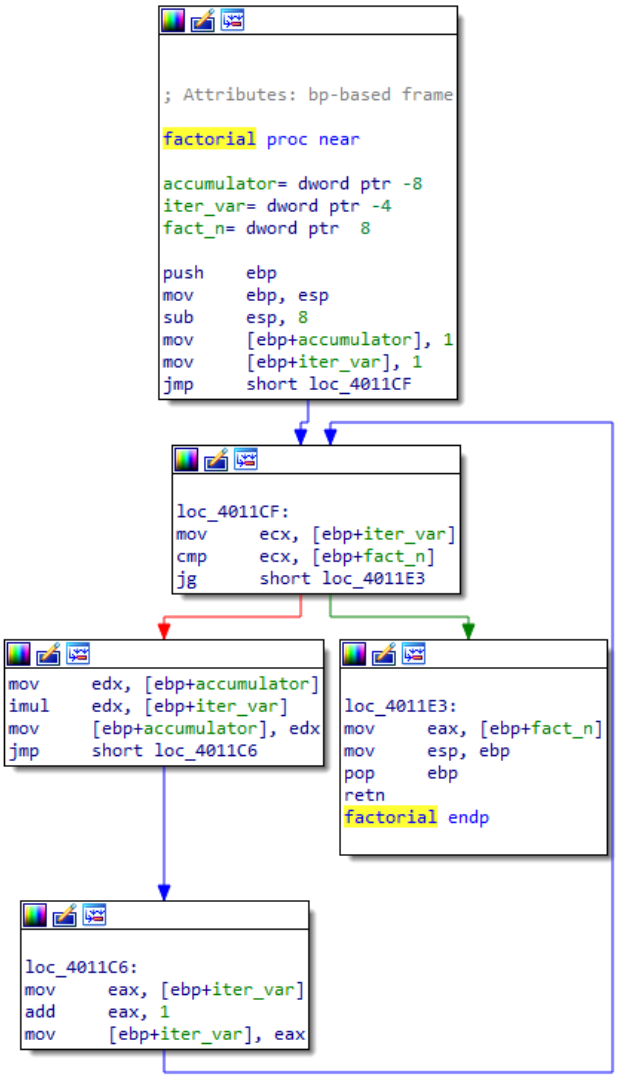

Figure 2.23 An example of a factorial program disassembled with IDA.

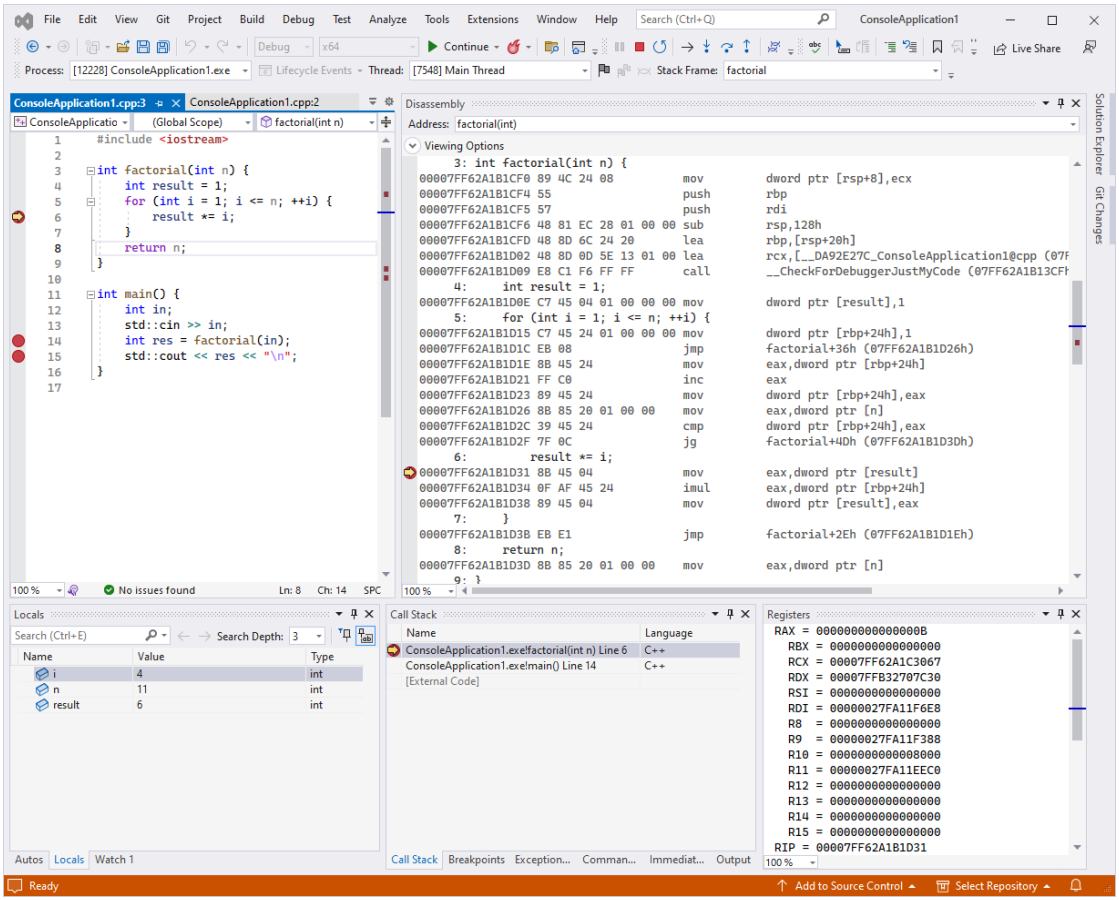

Figure 2.24 An example of debugging factorial program with MSVC debugger.

calls and locations from where those calls happened. On the right side, values in registers can be seen. This is only a basic overview of the features, as there are many more.

2.5.4 Case study - LLDB

In this section, we go into more significant details about the LLDB [6] debugger. It is a part of the LLVM tools collection [30]. Supported languages for debugging are the C, Objective-C, and C++ languages (LLVM tools support mainly those three languages) on desktop and iOS devices. It works closely with the Clang compiler, which is also part of LLVM, and the LLVM disassembler. Thanks to this, it stays very up-to-date with current C++ standards. The source code is kept modular with a plug-in architecture. LLDB is both a source and instruction-level debugger.

LLDB offers an API through which one can interact with the debugger. For example, the command line application through which one can use LLDB uses this API. LLDB is written in modern C++, but a bridging interface to the API is provided for Python. Thanks to this, one can use LLDB for debugging and, for example, as a library to disassemble machine code [6].

LLDB provides a command line interface (CLI) through which one can use the debug-

ger directly. The CLI provides a variety of commands that one can use. Some examples include

- `process launch` - Starts the debuggee.
- `process attach --pid 123` - Attaches to the process with PID 123.
- `thread step-in` - Source level single step in currently selected thread.
- `thread step-inst` - Instruction level single step in currently selected thread.
- `thread until 12` - Run until line 12 is hit or the current function returns in currently selected thread.
- `breakpoint set --name foo` - Set breakpoint on all functions named `foo`.
- `breakpoint enable 1` - Enable breakpoint 1.
- `watchpoint set variable x` - Watch a variable `x` for any writes.

All commands are of the following structure:

```
<noun> <verb> [-options [option-value]] [argument [argument...]]
```

Some of the commands have shorter forms, like `step` for `thread step-in`. Using an unambiguous prefix works as well, like `br s -M foo` for `breakpoint set --name foo`.

The LLDB debugger has an expression parser. While debugging, one can write an expression, and the debugger interprets that expression and prints the result back. An example of usage can be to look at some value in an `std::map` structure, which can be achieved by using the `frame variable map_var["Value"]` command. The debugger uses Clang, the C++ compiler, for parsing expressions. This can be used in conjunction with backticks in commands. For example, the `register write rax` `` `loc` `` command will write the value of variable `loc` into the register `rax`.

Chapter 3
Tiny x86

At the FIT CTU, in the NI-GEN course, students have to write a compiler. The Tiny x86 (T86) architecture was created to make code generation easier. This allows the students to focus on more interesting parts of compiler design, such as optimizations. To be able to execute programs written using this architecture, a virtual machine was created. This was all done as a part of a master's thesis made by Ivo Strejc [34]. This chapter explores said architecture and the virtual machine. It also delves into the existing debugging capabilities of said virtual machine.

3.1 The T86 Instruction Set Architecture

The T86 name is an abbreviation for Tiny x86. It was designed with a goal in mind, that goal being the fact that it is an educational ISA. The Virtual Machine made for the ISA allows configuring the number of registers, the RAM size, or the length of each instruction. This allows the student to develop their compilers incrementally.

The T86 uses a Harvard architecture, meaning the data and instructions are physically separated. The author doesn't specify the reason for this. Presumably, it is to ease the implementation of the virtual machine since the memory and instructions can be represented as two separate array-like members of the virtual machine. Memory is addressable by 64-bit blocks, not by 8-bit blocks as is the custom in modern computers.

It shares some of the registers we saw on x86-64: the program counter (PC), the stack pointer (SP), the base pointer (BP), and the flags register. The intended roles for these registers are the same as on x86-64 architecture. It also has other general-purpose registers. As previously said, the amount of these is configurable. The registers store a 64-bit value. It also has float registers, which can store 64-bit float values. These are separated from the standard registers, similarly to x86-64.

The addressing modes, or what kind of operands instructions can have, include immediate values, registers, and memory accesses. They can also be combined in various ways, like `[R0 + R1 * 2]`, or `[R0 + 10 + R1 * 2]`. The addressing modes are, however, not arbitrary, `[R0 + R1 + R2]` is not a correct addressing mode for T86. For a full list, refer to [34]. The instructions that are taken over from x86-64 have more restrictive addressing modes than in x86-64. For example, the `add` instruction can take a memory offset as a destination operand in x64-86, but in T86, it can only take registers.

```
SUB SP, 1
MOV R0, 5
MOV [BP - 2], R0
LEA R1, [BP - 2]
MOV [BP - 1], R1
MOV R2, 4
```

Figure 3.1 A small piece of a T86 program.

```
ProgramBuilder pb;
pb.add(MOV{Reg(0), 50);
pb.add(PUTCHAR{Reg(0)});
pb.add(HALT{});
auto program = pb.program();
Cpu cpu;
cpu.start(std::move(program));
while (!cpu.halted())
cpu.tick();
```

Figure 3.2 Simple example of how to create and run a simple program in the T86 VM.

Other than that, the ISA is a subset of the x86-64 most used instructions, many of which we have already seen in various examples throughout the thesis. Interesting exceptions are the IO instructions - `PUTCHAR` and `GETCHAR`, which allow for very primitive input and output handling. Also, an `DBG` and `BREAK` are defined. These are used for debugging, but in a very different way than we have seen in previous sections. We will touch upon them when discussing the virtual machine implementation since they are very much tied to it. A small sample of a T86 program can be seen in figure 3.1.

3.2 T86 Virtual Machine

The primary objective of the virtual machine is to replicate the CPU as accurately as possible, without prioritizing execution speed. For instance, the virtual machine simulates the out-of-order technique briefly described in section 2.1.2. The purpose of the virtual machine is to allow the students to gain a deeper understanding of the effects of pipeline stalls and similar events on program speed. The virtual machine is able to generate statistics that provide information about these factors and how much they influenced the speed of the generated program.

The virtual machine (VM) is implemented in C++, using the newer standards up to C++17. The VM offers only a single interface, and that is the `ProgramBuilder`. This is a class through which one may construct a program for the T86 VM. An example of how to use this class is in figure 3.2. Currently, there is no other way for users to run programs in the VM. This means that the students are tied to the C++ language, or use some bindings if they want to use other language.

```
pb.add(DBG{[](Cpu& cpu) {
    if (cpu.getRegister(Reg{0}) == 0) {
        std::cerr << "Register 0 is set to zero!\n";
    }
});
```

Figure 3.3 Example of an debug instruction in T86 VM.

```
cpu.connectBreakHandler([](Cpu& cpu) {
    char command;
    std::cin >> command;
    if (command == 'c') return; // continue
    else if (command == 'r') { // Read register
        int num;
        std::cin >> num;
        int regval = cpu.getRegister(num);
        std::cerr << std::format("Register {} = {}\n",
                                 num, regval);
    } else if (command == 'w') { // Write register
        int num;
        int val;
        std::cin >> num >> val;
        cpu.setRegister(Reg{num}, Reg{val});
    }
    ... // Other commands
});
```

Figure 3.4 Small debugger implementation using T86 BRK instruction, abbreviated.

3.2.0.1 Debug Instructions

The VM offers some limited debug capabilities. It has the `DBG` and `BRK` instructions. The `DBG` instruction takes as an operand a function. The function has the following signature: `void fun(Cpu&)`. This function then gets executed when the instruction is hit. This can prove helpful in inspecting the internal state of the CPU. In figure 3.3, we show a possible usage of this instruction. The `BRK` instruction works similarly. It, however, has no operand. Instead, a function must be provided before execution to the CPU itself. `BRK` then always runs this function when hit.

Such debugging capabilities can be helpful but quickly prove insufficient. For example, when a step-by-step inspection is sought, a debug instruction must be placed at every second line of the program. Also, interactivity is not present. However, this function can accept input, so one could create a robust enough to handle register and memory writing. An idea of how this could be done is illustrated in 3.4 via the `BRK` instruction.

This still leaves much to be desired. Not to mention placing the debug instruction can prove very bothersome.

Chapter 4
Implementation

In this chapter, we describe how we went about the implementation of the debugger and reason about the design choices we made. Also, we describe which parts of the virtual machine were modified or added for the benefit of the debugger or, in general, for the greater good.

4.1 T86 ISA Extensions

In chapter 3.2, we showed how to build a program for the T86 VM. It was necessary to use the builder the T86 VM provides. We remedied this with a custom ELF-like format, which we provide a parser for. This format semantically follows the ELF format and allows the creation of a T86 executable. Contrary to the ELF, it is a text format to allow ease of use. In chapter 3.2, we showed how to build a program for the T86 VM with the builder interface. To allow the usage of other programming languages, we have created an ELF-like format for the T86 executables. The format is a text one, making it easy to use. An example of a program in said format is shown in figure 4.1. It is very similar to the assembly we have shown in previous sections. Thanks to this, students can implement their compiler in any programming language they want and emit the T86 program in this format as a text file. An unfortunate side effect is that we can no longer use the `DBG` instruction. However, the debugger we will later present will be much more powerful than the `DBG` instruction.

As can be apparent from the example, we also use sections. The `.text` section is the only mandatory one. It contains the instructions that will be executed. Another one is the `.data` section. Here, either raw numbers or strings can be written. The contents of this section are then loaded by the VM and stored into the memory, beginning at memory cell 0 and upwards. There are also debug sections, which we will present when discussing debugging information.

We will also add two new instructions. First is the `PUTNUM` instruction, which prints the numerical value in the register and a newline. This is intended as a very primitive debug instruction and to ease the automated testing of the compiler. The only other way of output was to print a char which was represented by the ASCII value. With this instruction, students can bootstrap the basic implementation of their compiler more easily.

```
.data
"Hello, World!\n"

.text
0   MOV [BP - 1], 0
1   JMP 8
2   MOV R0, [BP - 1]
3   MOV R1, [R0]
4   PUTCHAR R1
5   MOV R0, [BP - 1]
6   ADD R0, 1
7   MOV [BP - 1], R0
8   MOV R0, [BP - 1]
9   CMP R0, 13
10  JLE 2
11  HALT
```

Figure 4.1 Example of an T86 program which prints "Hello, World!".

Another one is the `BKPT` instruction. This instruction is similar to the `INT3` instruction from x86-64 or the `BKPT` instruction from ARM. It is a software breakpoint. Control will be passed to the connected debugger when the instruction is hit. This is thoroughly described in the next section.

4.2 T86 Debugging Support

We could bake the debugger into the virtual machine itself, which would likely be the simplest way to implement it. However, the goal of the debugger is not only to ease the code generation part but to be a learning point so that students might grasp how a real debugger works[1]. Because of this, we aim to simulate the real-world debuggers as closely as possible. The compilers may also have more targets in the future, not just the T86 VM. If we made the debugger part of the T86 VM, we could not use it for a possible new virtual machine. In conclusion, the virtual machine and the debugger will be two entirely different programs and as such, two completely different processes.

In the debugger implementation for Linux, the subject of section 2.2.1, we described how an operating system's kernel allows the debugger's implementation via a specific API. There is no operating system between the virtual machine and the program. Still, we will strive to make the API similar to the ptrace API. The debugger and the VM will have to communicate together somehow. For interprocess communication, there are several possibilities.

Both the VM and the debugger use an abstract class representing an interface that provides two methods, `Send` and `Receive`. The implementation of this interface then handles the concrete way of communication. The debugger and VM do not care about it; they merely use these two methods. There are currently two implementations of this interface. One is using network communication through sockets. This way, the debugger may attach to an existing process, even on an entirely different computer. It, however,

[1] The VM followed the same philosophy.

has a disadvantage. The messages sent are often short and we need to send a lot of them. This proved too slow, even with few messages being sent. The second implementation is via threads. The debugger runs the VM in another thread, and they communicate via shared queues. This is far faster and allows the debugger to run the process by himself, making it easier to use and behave like real-world debuggers.

The format of the communication is a text one, merely because of the ease of use as opposed to binary format, it is also clearer to see what is happening. The commands that the virtual machine API offers are

- `PEEKREG x` - Return values of all normal registers.
- `POKEREG x y` - Sets the value in register `x` to y.
- `PEEKFLOATREG` - Return values of all float registers.
- `POKEFLOATREG x y` - Sets the value in float register `x` to y.
- `PEEKDEBUGREG` - Return value in all debug registers.
- `POKEDEBUGREG x y` - Sets the value in debug register `x` to y.
- `PEEKDATA x cnt` - Return value in memory at addresses x to $x + cnt - 1$.
- `POKEDATA x y` - Writes a value y into a memory at address `x`.
- `PEEKTEXT x cnt` - Returns instruction from x to $x + cnt - 1$.
- `POKETEXT x INS` - Rewrite the instruction at address `x` with the newly supplied instruction.
- `CONTINUE` - Continue the execution.
- `TERMINATE` - End the execution.
- `REASON` - Get the reason why the program stopped (breakpoint, singlestep, halt).
- `SINGLESTEP` - Does native level single step.
- `TEXTSIZE` - Returns the size of the program.

An example of how those commands can be used for communication between the virtual machine and the debugger is shown in figure 4.2. The interface is similar to basic ptrace commands. We separate the memory and instruction writing because T86 uses Harvard architecture, whereas Linux doesn't separate text and data address spaces[13], so the two requests were equivalent there. The API is made to be simple on purpose. Anything more complex should be handled in the debugger itself.

The `Cpu` class is the backbone of the whole virtual machine. It is responsible for executing the program. The `Cpu` has some light debugging capabilities; they were mentioned in section 3.2.0.1. Unrolling must be done to display proper values in registers and memory. This was described in section 2.1.2. This is already implemented in the VM by its author, and we can use the same mechanisms for the software breakpoints and single-stepping.

The `Cpu` class offers the `halted` function. This function returns true if the `Cpu` executed the `HALT` instruction, which marks the end of a program. It also has a `tick` method, which

does one tick of the CPU. This does not mean that one tick executes one instruction; there are several pipeline stages that the instruction needs to go through. To run the VM, one would have to write a loop that checks if the VM has halted. If it did not, then one tick would be performed. As of now, the Cpu cannot signal abnormal conditions (excluding halt), like breakpoint hits.

We added another manager-like class called *OS*. This class will take care of running the program via the CPU class. We also added an *interrupt* capability to the CPU. For example, when the `BKPT` instruction is executed, CPU signals interrupt number 3. To check if and which interrupt happened, the CPU now provides a function for it, similar to the `halted` function. The OS calls the `tick` method periodically, and after every tick, it checks if a halt or interrupt occurred. If it did, then it passes it to some handler.

When an interrupt that is caused by some debugging features happens (like the `BKPT` instruction), the OS calls a method in the `Debug` class. This class is also a new addition and is responsible for communication with the debugger. It uses the text protocol we mentioned previously.

There are several features we added to the CPU to enhance debugging. One of them is the `BKPT` instruction, which we have already talked about. Executing this instruction causes an interrupt `3` to occur. It is also possible to set a special flag that causes the CPU to send the interrupt `1` after every executed instruction.

We also added debug registers. They are a special type of registers designed for triggering breaks on memory access. There are a total of five debug registers, with the first four containing the memory cell addresses. The fifth register, called the control register, contains four bits that indicate the status of each of the first four registers. If a register is active and the program writes to a memory cell with the same address as is stored in the register, an interrupt `2` is generated. Furthermore, the control register's bits from 8 to 11 reveal which register caused the interrupt. For instance, if bit 10 is set to 1, the third register is responsible for the interrupt and the address stored in that register is the one that was written into.

4.3 Native Debugger

The implementation is done in the C++ language. It uses newer standards up to the C++20 standard. The debugger is implemented as a library. We will call this the backend of the debugger. A command line interface was also developed, through which the users might interact with the debugger. This will be called the frontend of the debugger.

The implemented debugger consists of two main parts. The first one aims to support native (instruction) level debugging. This part work without **any** debugging information whatsoever. The second part focuses on source-level debugging, described in the section 4.4.

The native debugger is split into two additional main layers to make it more modular. The first layer is called a `Process`. It is an interface representing the debuggee process. The implementation of this interface is responsible for dealing with the concrete architecture, the API of that architecture, and the communication with the debuggee. One implementation is provided for the T86 VM. For instance, it has a method called `ReadText` and `WriteText`. The internals of these methods use the `PEEKTEXT` and `POKETEXT` API we described. Outside of this class, the communication API is never used. If, in the future, another virtual machine is made, for whichever architecture, it is only needed to

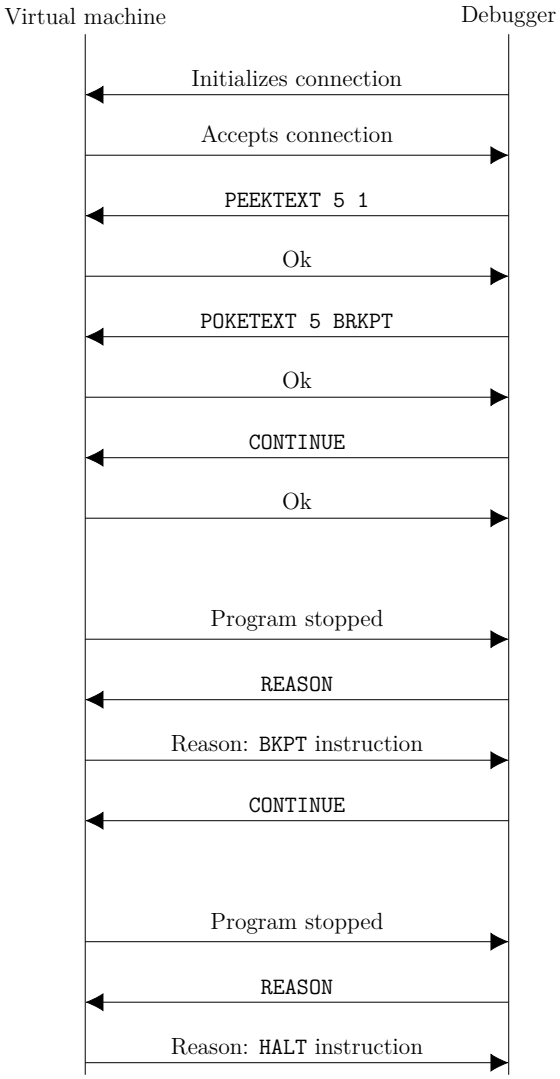

Figure 4.2 A sequence diagram for the communication between the virtual machine and the debugger.

```
SoftwareBreakpoint CreateSoftwareBreakpoint(uint64_t address) {
    auto opcode = GetSoftwareBreakpointOpcode();
    // Read the text at the breakpoint address
    auto backup = process->ReadText(address, 1).at(0);
    // Rewrite it with the breakpoint opcode
    std::vector<std::string> data = {std::string(opcode)};
    process->WriteText(address, data);
    // Check that it was truly written
    auto new_opcode = process->ReadText(address, 1).at(0);
    if (new_opcode != opcode) {
        Error(...);
    }
    // Create a breakpoint object which keeps the text backup
    return SoftwareBreakpoint{backup, true};
}
```

Figure 4.3 Debugger code in `Native` class to enable a breakpoint.

implement this interface. The rest of the debugger can be used as-is.

Another layer is the `Native` class, which implements the complicated logic behind a debugger, like setting a breakpoint, handling single-step, and so forth. It is the primary bread and butter of the native part of the debugger. Most algorithms are similar to the Linux debugger implementation presented in section 2.2.1. For illustration, in figure 4.3 we show a snippet of code used to create a breakpoint. It first reads the text at the address where we want to set the breakpoint. The breakpoint opcode then rewrites this text, and the backup of the text is stored.

When we arrive at the breakpoint and want to continue further, we need to unset the breakpoint, i.e., replace the breakpoint opcode in the T86 program with the backup we saved, do a native-level single step, and write the breakpoint back.

Since breakpoints change the underlying code of the debuggee, we need to be careful when presenting information to the user. If we printed the text that we get from the debuggee, it might contain the `BKPT` instructions we set earlier. We need to mix it with the backup code stored in breakpoints to show the assembly of the program correctly.

The Native class uses a `DebugEvent` structure which indicates what caused the VM to stop. It is implemented as a `variant` of multiple structures, for instance, the `BreakpointHit` or the `WatchpointTrigger` structure. It is a variant because the watchpoint also needs to convey information about an address that caused the break, as do breakpoints. It could also signal if the break was caused by reading or writing to the memory cell, although for now, the T86 VM only interrupts on writing.

The native debugger has the following features:

- Breakpoints - Can set, unset, enable and disable software breakpoints.
- Watchpoints - Can set and unset watchpoints on memory writes.
- Single stepping - Can do native level step into, which executes current instruction, out, which runs the program until it leaves current function and over, which treats function calls as a single instruction.

- Text manipulation - Can read and write into the debuggee text area, effectively allows to overwrite the running code.
- Data manipulation - Can read and write into the program memory area.
- Register manipulation - Can manipulate with normal, float and debug registers.

4.4 Debugging Source Code

With the solid foundation represented by the native part of the debugger, we can extend it by providing some form of source-level debugging. For this part, we need to remember that the debugger will only be used by students. As such, we ought to have gentler debugging information than DWARF, but we certainly can take inspiration from it.

As we previously mentioned, the executable with T86 code is separated into sections. The `.text` and `.data` sections are for the VM. We will introduce new sections where debugging information will be stored. All those sections will have `.debug_` prefix. The simplest new section is `.debug_source`, which should contain the original source code which was compiled into this executable. This later allows us, with the combination of other information, to display the source code lines.

The main philosophy of the source-level debugger is to allow an arbitrary amount of debug information. For instance, the user can generate information about one function only, and for that function, source debugging capabilities will work, but not for any other. This means that users can generate debugging information incrementally.

The debugger is, of course, not only limited to TinyC language. Any imperative language that can be encoded with the following debugging information is suitable to be debugged at the source level. We show an example of this in a provided test case for the debugger. It can be found as an attachment to the thesis on path .

doplnit cestu

4.4.1 Line Information

Line information is one of the simplest to generate. Since it will probably be the most generated information by the students, we choose the following simple format for it: `<line>:<address>`. The section `.debug_line` is made up of these lines. The format is self-explanatory. It says that line `x` corresponds to address `y`. This is very easy to generate.

With this information, we are able to do source-level breakpoints. If the source code is also provided, we can show the user on which line is the debugged program currently paused. It is not necessary to specify every line in the program. The debugger will refuse to put a source-level breakpoint on some line if it does not have the necessary information.

We did this the other way than the DWARF, which map address to instructio and records addition information, like if it is a basic block. Rea son about why we choose this

4.4.2 Debugging Information Format

In the line information, we provided a straightforward format. However, we will need a more sophisticated structure to describe some advanced constructs of the source code. Here we will draw inspiration from the Dwarf Debugging Information (DIE). Take a look at figure 4.4, which shows an example of such debugging information. It has a tree-like structure which, in some ways, mimics the original program. The nodes of this tree are

```
DIE_compilation_unit: {
DIE_function: {
    ATTR_name: main,
    ATTR_begin_addr: 0,
    ATTR_end_addr: 10,
    DIE_scope: {
        ATTR_begin_addr: 0
        ATTR_begin_addr: 10
        DIE_variable: {
            ATTR_name: d,
        },
    }
}
}
```

Figure 4.4 Debugging function information for the T86 debugger.

also called debugging information entries (DIEs). Those entries can have other entries as their children, and each entry has a tag that is part of its name (for example, the `compilation_unit` tag). They can also have attributes that describe their properties. As can be seen, this is very similar to the DWARF debugging information format. Unlike DWARF, it will be a text format. This allows us to generate the format easily and to spot mistakes quickly. We don't want the students to debug their generated debugging information.

For instance, the tag `DIE_function` represents a function. As attributes, it has a name, beginning address, and end address. With this additional information, we can set a breakpoint on a function name. We can also display in which function we are located when a break happens.

It also has one direct child, a `DIE_scope`. The scope entry is mainly used for keeping track of which variables are currently active because the T86 (or any other assembly language in general) has no notion of scopes. In the scope entry in the example, only one variable called `d` exists. Thanks to those entries, we can list currently active variables. We, however, often need to examine the value of a variable. To achieve this, information about the location and the variable type is needed.

The type information is encoded as a standalone DIE. Currently, three type entries are present, one for primitive types (int, double, or char), one for pointers, and one for structured types (`struct` or `class` in C++). Other types can be easily added in the future. The types are saved as separate entries, and as such, we need some way to link them together with the variables. We will use the `ATTR_id` attribute to achieve this. This attribute should be unique for every entry. This role is similar to the `id` attribute of HTML elements [35]. The variables themselves have the `ATTR_type` attribute, which will have an id of the type as its value. An example of a pointer type that points to an int type is in figure 4.5. If we had a variable that is a pointer to int, it would need to have the `ATTR_type: 1` attribute because the id of a pointer type to integer is one.

The primitive types need to have their size. For T86, this is the number of memory cells it occupies, which will almost always be one since one memory cell is 64 bits. It also has a name for its primitive type. Currently, three are supported:

```
DIE_primitive_type: {
    ATTR_name: int,
    ATTR_id: 0,
    ATTR_size: 1,
},
DIE_pointer_type: {
    ATTR_type: 1,
    ATTR_id: 1,
    ATTR_size: 1
},
```

Figure 4.5 Debugging type information for the T86 debugger.

```
- [PUSH R0]
- [PUSH BP; PUSH -2; ADD]
- [BASE_REG_OFFSET -2]
```

Figure 4.6 Debugging type information for the T86 debugger.

- `int` - A signed integer.
- `float` - A floating point number.
- `char` - A number representing an ASCII character.

Structured types are more complicated. They need to have a list of members which are stored in the structure. For each member, an offset from the beginning of the structure must also be specified. It also must provide a size because the compiler might align it, and it may be larger than the sum of the size of its members.

With this information, we can show the type of the variable. Nevertheless, the most valuable thing is its value. Variables are either stored in memory, registered, or optimized out completely. We will follow DWARF's footsteps and provide a language for a virtual machine that one can interpret to gain information about the variable location.

Several examples can be seen in figure 4.6. Same as DWARF, it is a stack-based machine. After computation is done, the top value on the stack is considered to be the location of the variable. The instruction `PUSH` pushes either offset or register onto the stack. In the first example, the final location is the register `R0`. In the second example, the location is a memory offset equal to the value in register `BP` minus two. The third example has the same meaning, but a shorthand was provided since the variables will most often be stored at some offset from the register `BP`. There is also a dereference instruction, which can be used to inspect to which location a pointer points.

This information is stored in a variable attribute called `ATTR_location`. If all this information is provided, we know where the variable is stored and may look up its value. Together with the type information, we might also properly interpret the value and report it to the user.

```
Process stopped, reason: Software breakpoint hit at line 11
function main at 7-18; active variables: a, b
      9:    int b = 6;
     10:    swap(&a, &b);
@->  11:    print(a);
     12:    print(b);
     13:}
```

Figure 4.7 Example of the CLI reporting a breakpoint hit.

4.4.3 Location Expressions

We could make a very straightforward implementation of getting variable value by its name. It is only a matter of finding the variable DIE with the correct name and interpreting its location and type. However, we often need to inspect some more complicated expressions. For example, we may want to display some struct member or a value at which pointer points.

The debugger has a built-in interpreter for such expressions. It builds an AST from the expression and interprets it using AST-walk [29]. It leverages the `Native` class to fetch variable values. It supports almost all C operators, excluding the assignment operator. An example of such an expression is `foo[2]->bar + 3`.

4.4.4 Frontend

Finally, we provide a command line interface that leverages the debugger library to make the debugger usable. It provides many commands, and its manual can be found as an appendix to this thesis.

The main priority of the CLI is to make the debugger easy to use. It consists of several commands, one of them is `breakpoint set 5`, which will set a source-level breakpoint on the fifth line of the program. It is, however, not necessary to write the whole command. Any prefix will do, like `b s 5`. The CLI leverages the *linenoise* [36] library to make the REPL satisfying to use.

The CLI also displays various information on program stop, like why the program stopped, on which address or line, and prints the surrounding lines of assembly or source. The CLI can also list breakpoints and display their locations in the disassembly or the source code. Figure 4.7 provides an example of a breakpoint hit report. The debugger had all debugging information available here. It can show the line in the source code where the breakpoint happened, name the offending function, and variables in scope.

Chapter 5
Evaluation

This chapter aims to assess the effectiveness of the debugger. Speed is measured to evaluate whether the debugger has any performance impact when debugging standard and computative demanding programs. We also evaluate its feature richness and ease of use, comparing it to modern state-of-the-art debuggers like the GDB.

5.1 The development process

The development of the thesis was done in a GitHub repository. The power of Github was leveraged not only for keeping history but also for recording issues, planning the development, or ensuring that the repository is in a consistent and working state via GitHub actions, which runs tests before a pull request is accepted. The actions run on two operating systems, Ubuntu and MacOS, ensuring the project works on both.

The project itself contains many tests. The code is first tested via many unit tests using the `GoogleTest` framework. The unit tests cover almost all parts of the code. It also has integration tests for T86-CLI and the debugger CLI. Another student created a TinyC to T86 compiler as part of his thesis. He was kind enough to send the implementation to us so that we could generate various tests more easily. As a result, most of the integration tests were generated by the said compiler.

citace

5.2 Usage and user testing

We followed the interface of the GDB closely so that the users were familiar with the debugger before even running it. However, we choose to diverge on some of the features. For example, the GDB uses the `stepi` command for single stepping. This form has a common prefix with the `step` command. We, however, expect our users to use assembly-level debugging more often than source-level debugging, so we choose the `istep` command form instead. The two commands have no shared prefix, so typing `is` is enough.

Also, the program is run via `run`, this starts the VM, but the program is paused. In GDB, one has to use the `start` command to get equivalent behavior, `run` runs the program without stopping at the beginning. When we did a brief user testing, it wasn't very clear to the user. However, renaming the command `run` to `start` would cause it to

Test Case	Quicksort - Time	Prime numbers - Time
1.	9.96	11.68
2.	9.32	12.33
3.	10.9	58.32
4.	10.9	72.85
5.	9.73	10.53
6.	9.65	230.81

Table 5.1 Performance comparison when using various features of the debugger. The Quicksort had 674 breakpoints hits, while prime numbers had 101266 hits. Each case was run five times and average was taken.

have the same prefix as `step`. Considering this, we've decided to leave the command as `run`.

There are other minor things; most of them come from the fact that we prioritize assembly-level debugging, whereas GDB focuses more on the source level. We provide a short list of examples showing how to achieve something in GDB and our debugger .

nknout path do urcu nebo to pridat m jako appendix

One student already tested the debugger and said the experience was quite pleasant. There were a few minor things that he didn't like and offered solutions for (for example, the usage message can only be displayed after the debuggee program is executed, meaning the `run` command must be invoked first), most of which we took to heart and corrected. Additionally, the student that provided us with the TinyC compiler used the debugger to debug the code his compiler generated. A talk about how is the repository used in production by students..

mplete this part nen code is public

5.3 Performance

The performance of the debugged program can suffer. In this section, we show which debugging features hurt performance the most and if there is any way around it. The tests were done on two programs: a quicksort [37] algorithm and a naive prime number checker. The difference between those two programs is that the quicksort is recursive, whereas the primes are implemented via a loop. The runtime of those two programs on the virtual machine alone is similar.

clude the source de here? Both the 86 and TinyC

We will measure the speed on the following cases:

1. Run the VM without the debugger.
2. Connect the debugger and immediately invoke the `continue` command.
3. Connect the debugger and set breakpoint at hot spot of the program. For quicksort, this will be the main recursive function. For primes, it will be the body of the loop. When the breakpoint is hit invoke the `continue` command.
4. Same as before, but at every breakpoint, hit read a 100 cells of memory and the `IP` register.
5. Step over a computationally expensive function. For quicksort, it is the `quicksort` function; for primes, it is the `is_prime` function.

6. Set a breakpoint in the most expensive function and run the program. Remove the breakpoint on the first hit and step out of the function. The expensive functions are the same as in the previous test.

The results are in table 5.1. It is apparent that just having the debugger connected introduces almost no slowdown. In the quicksort case, the program is even faster. This is probably due to measurement errors. The breakpoints, however, do cause a slowdown. In the quicksort, the program had 674 breakpoint hits, while in the prime numbers, it had 101266. The quicksort version is 1.1 times slower, which is negligible. However, in the prime numbers test performance takes a severe hit with being at least 5 times slower. This shouldn't come as a surprise, as there is an enormous number of breakpoint hits. The communication between the debugger and the debuggee starts to slow down the program. Given the circumstances, this result is still satisfactory. Reading memory and registers together with the breakpoints introduces minimal slowdown.

The step-over result is not very surprising. It puts a breakpoint after the call and runs the program, so the result should be roughly the same as case number one. Step out is stepping over until a return is encountered. This works well in the recursive function because it skips a large part of the program. But in the prime case, which is implemented via a loop, it severely slows down the program because it essentially single steps through the entire thing.

The debugger is more than suited for usage on regular programs. In the case of very computationally intensive programs, certain debugger features will start to struggle. On the other hand, a hundred thousand breakpoint hits is an unpractically large number. We consider the quicksort example as a peak of what the students will have to debug, and in that example, the debugger had almost no slowdowns. Still, it is something to keep in mind while using the debugger, and there is a potential for improvements in the future.

Chapter 6
Conclusion

6.1 Summary

We explored debugging capabilities of modern CPU architectures. We also described how debugging is supported at various layers, from operating systems to compilers. Then, we discuss the T86 architecture and its lackluster debugging support. We remedy this by adding a debugging API inspired by modern architectures. We also created a native and source-level debugger, which is production ready and already used in the NI-GEN course at FIT CTU. This debugger is extensible enough that if a new architecture, virtual machine, or source language comes into play, it should be fairly easy to add debugging support.

Trosku odvazny ale snad to tak dopadne

The debugger encourages students in the NI-GEN course to investigate how the debugger works, the connection between the generated machine code and the source code, and to emit information about those connections so that the debugger can work at the source level. It can also make their lives easier since the debugger works at the native level without additional work, allowing them to debug the code their compiler generated.

The tool is, along with the enhanced T86 virtual machine, openly available at `https://github.com/Gregofi/monorepo`.

6.2 Future Work

There are many possible improvements to be made. We have created a new executable text format for T86 ISA and, consequently, a parser for that format. Students, however, have to generate this text, which includes not only the instructions themselves but also the debugging information. A builder interface could go a long way, especially for the most commonly used languages in the NI-GEN course.

The native part of the debugger is fairly complete. It can always be extended for new architectures and virtual machines should they emerge. The source-level debugger can, however, be extended. The expression command, which evaluates a TinyC expression, can't handle function calls. There is also no way to format the output (for instance, to print an integer as a hexadecimal number).

New types can always be added to the debugging information. For example, we're missing enums, qualifiers like `const` and `mutable`, or more advanced types that are in the

C++ standard library. Those types are not in the TinyC language that is being taught in the NI-GEN course, but the debugger is not strictly tied to the language.

The debugger currently does not handle frame information in any way. This could be a helpful addition so that it displays call frames and allows one to step out of them. This would require a new section, which would have to contain enough information to simulate an unwind. The exact mechanism was described in section 2.3.2.4.

The T86 VM generates statistics about the program execution. It records things like pipeline stalls. Since the debugger has the capability to connect the source to the assembly, it could display the code hot spots in the source code directly.

Last but not least, a graphical user interface could be created for the debugger. Currently, it only offers a command line interface. A graphical interface could be more pleasant to work in. Alternatively, it could be hooked to an existing editor, like the Visual Studio Code, which allows the usage of other debuggers like LLDB or GDB.

Bibliography

1. TVRDÍK, Pavel; ŠTEPANOVSKÝ, Michal. *Architectures of Computer Systems, Lecture 1* [FIT, CTU]. 2022. lecture slides.
2. AHO, Alfred V; LAM, Monica S; SETHI, Ravi; ULLMAN, Jeffrey D. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.
3. MAREŠ, Martin; VALLA, Tomáš. *Průvodce labyrintem algoritmů*. CZ. NIC, 2017.
4. REASON, James. *Human error*. Cambridge university press, 1990.
5. MYERS, Glenford J; SANDLER, Corey; BADGETT, Tom. *The art of software testing*. John Wiley & Sons, 2011.
6. *LLDB Homepage* [online]. [N.d.]. Available also from: `https://lldb.llvm.org`.
7. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture* [online]. [N.d.]. Available also from: `https://cdrdv2.intel.com/v1/dl/getContent/671436`.
8. BLEM, Emily; MENON, Jaikrishnan; SANKARALINGAM, Karthikeyan. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 2013, pp. 1–12. Available from DOI: `10.1109/HPCA.2013.6522302`.
9. BLEM, Emily; MENON, Jaikrishnan; VIJAYARAGHAVAN, Thiruvengadam; SANKARALINGAM, Karthikeyan. ISA Wars: Understanding the Relevance of ISA Being RISC or CISC to Performance, Power, and Energy on Modern Architectures. *ACM Trans. Comput. Syst.* 2015, vol. 33, no. 1. ISSN 0734-2071. Available from DOI: `10.1145/2699682`.
10. TVRDÍK, Pavel; ŠTEPANOVSKÝ, Michal. *Architectures of Computer Systems, Lecture 5* [FIT, CTU]. 2022. lecture slides.
11. TANENBAUM, Andrew. *Modern operating systems*. Pearson Education, Inc., 2009.
12. PETERSON, James L; SILBERSCHATZ, Abraham. *Operating system concepts*. Addison-Wesley Longman Publishing Co., Inc., 1985.
13. *Linux Programmer's Manual - PTRACE(2)* [online]. [N.d.]. Available also from: `https://man7.org/linux/man-pages/man2/ptrace.2.html`.

14. *Linux Programmer's Manual - signal(7)* [online]. [N.d.]. Available also from: https://man7.org/linux/man-pages/man7/signal.7.html.
15. *Linux Programmer's Manual - execve(7)* [online]. [N.d.]. Available also from: https://man7.org/linux/man-pages/man2/execve.2.html.
16. *Linux Kernel source code - Trap flag usage* [online]. [N.d.]. Available also from: https://elixir.bootlin.com/linux/v6.1.8/source/arch/x86/kernel/step.c%5C#L133.
17. *Writing a Linux Debugger* [online]. [N.d.]. Available also from: https://blog.tartanllama.xyz/writing-a-linux-debugger-setup/.
18. *msdn - the debugging application programming interface* [online]. [N.d.]. Available also from: https://learn.microsoft.com/en-us/previous-versions/ms809754(v=msdn.10).
19. *How Windows Debuggers Work* [online]. [N.d.]. Available also from: https://www.microsoftpressstore.com/articles/article.aspx?p=2201303.
20. *Microsoft Learn - Structured Exception Handling* [online]. [N.d.]. Available also from: https://learn.microsoft.com/en-us/windows/win32/debug/structured-exception-handling.
21. *Microsoft Learn - Debugger Exception Handling* [online]. [N.d.]. Available also from: https://learn.microsoft.com/en-us/windows/win32/debug/debugger-exception-handling.
22. ZAHRADNICKÝ, Tomáš; JIRKAL, Martin; KOKEŠ, Josef. *Reverse Engineering, 6. Debugging and Anti-Debugging* [FIT, CTU]. 2022. lecture slides.
23. *Microsoft Learn - Debug Events* [online]. [N.d.]. Available also from: https://learn.microsoft.com/en-us/windows/win32/debug/debugging-events.
24. *Microsoft Learn - Writing The Debugger's Main Loop* [online]. [N.d.]. Available also from: https://learn.microsoft.com/en-us/windows/win32/debug/writing-the-debugger-s-main-loop?source=recommendations.
25. LU, Hongjiu. *Elf: From the programmers perspective*. May, 1995.
26. *The DWARF Debugging Standard Version 2* [online]. [N.d.]. Available also from: https://dwarfstd.org/.
27. *The DWARF Debugging Standard Version 5* [online]. [N.d.]. Available also from: https://dwarfstd.org/.
28. *Introduction to the DWARF Debugging Format* [online]. [N.d.]. Available also from: https://dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf.
29. NYSTROM, Robert. *Crafting interpreters*. Genever Benning, 2021.
30. LATTNER, C.; ADVE, V. LLVM: a compilation framework for lifelong program analysis & transformation. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86. Available from DOI: 10.1109/CGO.2004.1281665.
31. *Source Level Debugging with LLVM* [online]. [N.d.]. Available also from: https://llvm.org/docs/SourceLevelDebugging.html.
32. *How to Update Debug Info: A Guige for LLVM Pass Authors* [online]. [N.d.]. Available also from: https://llvm.org/docs/HowToUpdateDebugInfo.html.

33. STALLMAN, Richard; PESCH, Roland; SHEBS, Stan, et al. Debugging with GDB. *Free Software Foundation*. 1988, vol. 675.

34. IVO, Strejc. *Tiny x86-Simulator procesorove architektury pro vyukove ucely*. 2021. MA thesis. České vysoké učení technické v Praze. Vypočetní a informační centrum.

35. RAGGETT, Dave; LE HORS, Arnaud; JACOBS, Ian, et al. HTML 4.01 Specification. *W3C recommendation*. 1999, vol. 24.

36. *Linenoise library* [online]. [N.d.]. Available also from: `https://github.com/antirez/linenoise`.

37. HOARE, Charles AR. Quicksort. *The computer journal*. 1962, vol. 5, no. 1, pp. 10–16.

Contents of enclosed media

- `readme.txt` ..... Overview of the medium contents
- `src`
  - `impl` ..... Source code of the implementation
  - `linux-debugger` ..... Source code of the proof of concept Linux debugger
  - `thesis` ..... Source code of the text in LaTeX
  - `text` ..... text
    - `thesis.pdf` ..... text in the PDF format