TU WIEN Informatics

Debugging Interface für VADL

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Simon Josef Kreuzpointner
Matrikelnummer 12021358

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall
Mitwirkung: Projektass. Dipl.-Ing. Tobias Schwarzinger, BSc

Wien, 11. September 2024

Simon Josef Kreuzpointner	Andreas Krall

TU WIEN Informatics

Debugging Interface for VADL

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Simon Josef Kreuzpointner
Registration Number 12021358

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall
Assistance: Projektass. Dipl.-Ing. Tobias Schwarzinger, BSc

Vienna, September 11, 2024

Simon Josef Kreuzpointner	Andreas Krall

Erklärung zur Verfassung der Arbeit

Simon Josef Kreuzpointner

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11. September 2024

Simon Josef Kreuzpointner

Danksagung

Ich möchte mich bei meiner Familie und meinen Freunden für ihre ständige Unterstützung bedanken, nicht nur während dieser Arbeit, sondern während meines gesamten Studiums. Mein besonderer Dank gilt meiner Freundin für ihre unermüdliche Ermutigung.

Meinem Betreuer Dr. Andreas Krall danke ich für seinen Rat und seine wertvolle Unterstützung und Dipl.-Ing. Tobias Schwarzinger für seine kontinuierliche Hilfe während der Arbeit an dieser Bachelorarbeit.

Schließlich möchte ich mich bei meiner Lerngruppe für die großartige Unterstützung und Freundschaft bedanken.

Acknowledgements

I would like to thank my family and friends for their constant support not only while working on this thesis, but throughout my studies. Special thanks also to my girlfriend, for her unwavering encouragement.

I would also like to thank my thesis supervisor Dr. Andreas Krall for his advice and valuable guidance as well as Dipl.-Ing. Tobias Schwarzinger for his continual help during the work on this thesis.

Finally, I would like to thank my study group for the great support and friendship.

Kurzfassung

Die Vienna Architecture Description Language (VADL) ist ein leistungsstarkes System, das die Erzeugung einer Vielzahl von Artefakten und Anwendungen, einschließlich Simulatoren beliebiger Prozessorarchitekturen ermöglicht. Ziel dieses Projekts ist die Implementierung einer Schnittstelle, die das Debuggen von Skripten, welche auf solch einem Simulator ausgeführt werden, mit herkömmlichen Mitteln ermöglicht. Diese Arbeit stellt einen Ansatz vor, wie dies erreicht werden könnte. Sie umreißt die Aspekte, die notwendig sind, um die Fähigkeiten zwei populärer Tools, GNU Debugger (GDB) und Visual Studio Code (VS Code), insbesondere für das Debuggen von ausführbaren Dateien auf einer beliebigen Architektur, zu nutzen. Darüber hinaus wird ein umfassender Ausblick auf mögliche zukünftige Ergänzungen und Erweiterungen der vorgestellten Lösung vorgeschlagen.

Abstract

The Vienna Architecture Description Language (VADL) is a powerful system that can be used to generate a multitude of artefacts and applications, including simulators for an arbitrary processor architecture. This project aims for the implementation of an interface that facilitates the debugging of scripts executed on such a simulator using conventional means. This thesis presents one approach on how this could be accomplished. It outlines the necessary aspects to leverage the power of two popular tools, GNU Debugger (GDB) and Visual Studio Code (VS Code), especially for debugging executables run on an arbitrary architecture. Furthermore, it provides an extensive outlook of potential future additions and enhancements to the presented solution.

Contents

CHAPTER 1

Introduction

Debugging is a fundamental aspect of software development. As programs become more intricate, the significance of debugging increases. One area where the ability to debug is particularly beneficial is in analyzing a program that is run on a simulated processor. This project aims to implement such a debugging interface for the Vienna Architecture Description Language (VADL) ecosystem.

Chapter 2 lays out fundamental knowledge of different tools involved in this project. Subsequently, Chapter 3 describes the overall approach taken, in order to achieve the aforementioned goal. Next, the actual implementation is discussed and explained in Chapter 4. Following that, instructions on how to use the debugging interface are presented in Chapter 5. Afterwards, Chapter 6 focuses on evaluating the implementation, followed by the conclusion to this project in Chapter 7.

CHAPTER 2

Background

This chapter introduces the reader to the GNU Debugger (GDB), the Debug Adapter Protocol (DAP) and the Vienna Architecture Description Language (VADL). These pieces of background information will help to build a comprehensive knowledge base for a better understanding of the subsequent discussion regarding this project.

2.1 GDB

The GNU Debugger is an open-source debugger [dev23] which enjoys great popularity. GDB supports debugging a range of different languages, like C++ or Assembly [SPS24, p. 229]. Furthermore, GDB can be compiled for different architectures, depending on the host or target, which is often used for *cross-debugging* [SPS24, pp. 742, 743].

2.2 GDB Remote Serial Protocol

The *GDB Remote Serial Protocol* is a protocol that allows the use of the GNU Debugger [dev24] on a remote target [SPS24, p. 765]. For simplicity, the full protocol name will be abbreviated to *serial protocol* in the following chapters.

To debug a program, GDB is in charge of starting the *debuggee*, which is the program that should be debugged [dev24]. The GDB instance that controls the debuggee, will henceforth be called *host*, while the debuggee will be called *target*. These two parties are generally speaking run on two different machines, although it is also possible to execute them on the same machine as well [SPS24, p. 314].

When using the serial protocol, the host will connect to a target that implements the receiving end of the protocol. This might either be a *remote stub* [SPS24, pp. 328, 329] or the *gdbserver* [SPS24, p. 316], a specific program run on the target machine. While the host acts as a client, both the stub and the GDB server application appear as a

server, implementing their part of the protocol. A remote stub is, in contrast to the gdbserver, not a standalone program, rather it is compiled into the program one wants to debug [SPS24, pp. 328, 329].

2.2.1 Remote Stub

The remote stub serves as an interface and connection point for the serial protocol, and is dependent on the architecture of the target machine. GDB does provide a few working remote stubs for different architectures, like i386 or Scalable Processor Architecture (SPARC) [SPS24, p. 329].

2.2.2 Connecting to a Target

There are two different types of connections to a target: *remote* and *extended remote*.

There are numerous differences between the two types, but a significant one is the contrasting behavior when the debuggee exits. In case of the remote connection mode, the host will disconnect from the target, while in the extended remote connection mode, the host will remain connected to the target, potentially restarting it. Similarly, the standard remote connection does not support attaching to the target, while the extended version does [SPS24, pp. 311–312]. Usually, a target only supports remote connection types, and not extended remote connection ones [SPS24, p. 311].

To actually connect to a target, the user has to issue the `target remote <medium>` or `target extend-remote <medium>` command respectively. The `medium` specifies the instrument that carries the so called *packets*. This might be a serial device, a local Unix domain socket or an Internet Protocol (IP) address [SPS24, pp. 313, 314].

2.2.3 Packet Structure

All data shared between the host and target, meaning commands and their responses, is transmitted using characters from the American Standard Code for Information Interchange (ASCII) in the form of *packets*, with the exception of *acknowledgements*, which do not follow the typical packet structure.

Acknowledgements consist of two types, either positive or negative. This is expressed with the ASCII characters + and − respectively [SPS24, p. 819].

Besides that, there are also special packets, that do not require acknowledgements [SPS24, p. 816]. These packets are called *notifications*, and they differ from the ordinary packet structure [SPS24, p. 765]. Throughout the present paper, “packet” will refer to an ordinary packet and not to a notification.

A packet has the following form:

```
$packet-data#checksum
```

A packet will always be introduced by the `$` character [SPS24, p. 765]. This is also a key difference to the notification packet, which starts with `%` [SPS24, p. 816].

The *packet data* includes the actual data carried by the packet. It must not include the characters `#` and `$` [SPS24, p. 765].

The *checksum* is computed from all the characters in the packet data. Concretely, it is the modulo 256 sum of its characters [SPS24, p. 765].

Prior to GDB version 5.0, the packet also included a sequence identification (ID), which was to be found directly after the packet start symbol `$`, delimited by a colon. Stubs that are build for later version of GDB do not have to handle sequence IDs [SPS24, p. 765].

In most cases binary data in the packet data is "encoded as two hexadecimal digits per byte of binary data" [SPS24, p. 765]. But there is also a second binary data representation that is in use. The second variant uses the ASCII character `}` with the hexadecimal value of `0x7d` as an escape character. The ASCII characters `#`, `$` and `}` must always be escaped. In order to achieve that, first the escape character is send, next the character to be escaped is combined via Exclusive OR (XOR) with `0x20` and then sent. Responses from a stub must also escape the `*` ASCII character because of run-length encoding [SPS24, p. 766].

Generally speaking, the first variant of encoding binary data is used in older packets like `g` or `G`, while the second one is utilized in more recent packets like `qXfer` [SPS24, pp. 765, 769, 770, 801].

Run-Length Encoding

The data of responses from a stub may be run-length encoded. Run-length encoding is most effective if the same character appears multiple times in a row, since it replaces repeated characters with only one instance of that character together with information on how often it shows up in the original data. In case of the serial protocol, any character that occurs more than three times in a row can be run-length encoded.

Between the repeated character and the number of occurrences in the run-length encoded data, the ASCII character `*` is placed. This is the reason for the obligatory escape of this character as mentioned in the previous Section 2.2.3.

The number of occurrences is also encoded in a specific manner. To obtain a printable ASCII character, the number 29 has to be added to the number of occurrences. As to not run into problems with the delimiter of packets, repeats of six or seven must be chopped into multiple smaller runs, since they would otherwise produce the ASCII characters `#` and `$` respectively for the encoded amount. Additionally, runs greater than 126 are not permitted [SPS24, p. 766].

2.2.4 Packet Flow

After receiving a packet, it is verified via the attached checksum. Based on the result of this verification a packet acknowledgement is send back. This can either be a + or − character. Both the host and the target will respond with an acknowledgement upon receiving a packet. Acknowledgements can also be disabled by switching into no-acknowledgement mode. In this mode the checksums are ignored and acknowledgements are neither expected nor sent [SPS24, p. 819].

2.2.5 Packets Overview

This section deals with the different packet types, sometimes also called packet forms, that the serial protocol provides. In this case "packet type" is referring to the structure of a packet's content, especially the first character, since the first character of the packet data determines the packet type. A common notation is *c*-packet, where *c* is the first character of the corresponding packet's data. For this reason, all packet forms starting with an upper- or lower-case letter are reserved by GDB [SPS24, p. 767]. For some packet types, even subsequent letters are reserved, as is the case for `q` packets [SPS24, p. 781].

While some packet types like `?` directly define a single command, in this case to query the halting reason [SPS24, p. 767], others define a whole family of more specific commands. For example, packets starting with `q` or `Q` are *general query-* and *general set packets* respectively [SPS24, p. 781].

2.2.6 Minimal Implementation

While the serial protocol provides more than 45 different types of packets [SPS24, pp. 767–776], the minimum number of required packets a stub has to understand is just seven. If the stub supports multithreading, a total of eight packets are needed. The required packets are:

- the `?` packet, so the host can query the halting reason,
- the `g` and `G` packets to read and write the general registers,
- the `m` and `M` packets for memory access,
- the `c` packet for continuing single-threaded implementations,
- the `s` packet for stepping, if the target supports it and finally
- the `vCont` packet, for multi-threaded continuing [SPS24, p. 766].

It is important to note, that with this subset of packets, debugging is hardly fruitful, due to the lack of breakpoint support.

2.2.7 Replies

Usually, each packet also defines the structure of its reply, with each packet having a different response. However, some of them are grouped together into a family of replies, that are applicable to a multitude of different packets, such as *stop reply packets*, or *standard error responses*.

Empty Response

The empty response, a packet with empty packet data[1], is sent by the target, if it does not support or understand the previous request [SPS24, p. 766].

Stop Reply Packets

Stop reply packets contain information about the current state of the debuggee, such as (i) if the process exited and what the exit code was, (ii) information about the received signal, (iii) if the process terminated or (iv) simply some output of the program.

Usually, stop reply packets are send right after the target halts. The only exception to that rule are the `?` and `vStopped` packets. In the non-stop mode, the target will reply immediately with `OK` and later signal the halt to the host via a notification [SPS24, p. 777].

The packets, that return a stop reply packet are: `C`, `c`, `S`, `s`, `vCont`, `vAttach`, `vRun`, `vStopped` and `?`.

The stop reply packets can take one of many forms, depending on the data they transmit or the state of the target. There are multiple different stop reply packets that report signals the target has received back to the host, like packets `S`, `T` or `X`, depending on whether the target is still running, has terminated or already exited. These are also the most used stop reply packets for this project. The signals are encoded into a two-digit hexadecimal number, defined in `signals.h`[2] of the GDB source [SPS24, p. 777].

Standard Error Responses

Error responses start with a capital `E`. In general, the standard error responses are applicable for each packet. They report an error to the host, along with either an error number or an error message. The error number is encoded as a two-digit hexadecimal number. Usually, the meaning of this error number is not specified [SPS24, pp. 766, 767].

[1] The whole packet will still contain the start and end character, along with the checksum. Thus, the empty packet is in fact `$#00` [SPS24, p. 766].

[2] The actual definitions are placed in the `signals.def` file, which are then imported in the header file. Both of these files can be found here: `https://sourceware.org/git/?p=binutils-gdb.git;a=tree;f=include/gdb;hb=a81f4e591fdb8530d832addc39beb31353b0ef2d` (visited on 07/16/2024)

2.2.8 Target Description

The so called *target description* is used to inform the host about the processor architecture of the target. This is useful, as many processors do have a common processor core as a basis, but differ in some other, smaller aspects [SPS24, p. 851]. For example, a processor might use the Reduced Instruction Set Computers V (RISC-V) architecture, although with a reduced register set.

With the target descriptions, GDB can make some changes at runtime, or tell the user, whether the requested architecture of the target is supported or not [SPS24, p. 851]. More crucially, it can support processor variants it has "never seen before" [SPS24, p. 851].

The target description is an Extensible Markup Language (XML) file, with a predefined format. It's format is defined in the `gdb-target.dtd` file[3], in the GDB source [SPS24, p. 851]

The overall stucture of the target description is as follows:

Listing 2.1: Schematic of the structure for the target description [SPS24, p. 852].

```
<?xml version="1.0"?>
<!DOCTYPE target SYSTEM "gdb-target.dtd">
<target version="1.0">
    [architecture]
    [osabi]
    [compatible]
    [feature...]
</target>
```

In this illustration elements in the square bracket signalize optional elements. The `feature` element is special in the regard, that it may occur multiple times [SPS24, p. 852].

A short overview over the top level elements is given here:

`architecture` This element specifies the architecture of the target. It accepts the same values as the `set architecture` command in the GDB frontend[4] [SPS24, p. 852].

`compatible` Similarly to the previous element, `compatible` specifies another architecture, that the target also understands. This element is only supported since GDB version 7.0 [SPS24, p. 853].

`osabi` This element specifies the Operating System Application Binary Interface (OS ABI) of the target. The acceptable values are the same as for the `set osabi` command in the GDB frontend, and is also supported since version 7.0 [SPS24, p. 853].

[3] The file can be found here: https://sourceware.org/git/?p=binutils-gdb.git;a=blob;f=gdb/features/gdb-target.dtd;h=d07703fca8b6b22bffb8acea99b40ce7f7a590c8;hb=HEAD (visited on 07/17/2024)

[4] The accepted values for that command depend on how GDB was built [SPS24, p. 307].

`feature` The most important element is the `feature` element. A *feature* in the target description is a logical element of the target. For now, features are used to describe the registers of the processor [SPS24, p. 853]. This includes, among other things, their name, type and size.

One other optional but crucial attribute for a register is `regnum`. This represents GDB's internal index of a register, which is used to order the registers in increasing order. It can be supplied by the user, or, if not present, is assumed to be one larger than the previous register number, starting from zero for the first register defined in the target description. This ordering is directly reflected in the serial protocol, for example in the `g` and `G` packets [SPS24, pp. 854, 855].

GDB also provides *standard target features*, so the user does not have to define every register of an existing architecture. The available standard target features can be found under the `features` directory[5] in the source files of GDB. In order to use one of the standard target descriptions, the name of the target description has to be supplied to the `name` attribute of the `feature` element in the target description [SPS24, p. 857].

It is important to note, that, while the target description can inform the host over the architecture of the target, the host still has to support the described architecture. That means, GDB must be compiled for the architecture of the target [SPS24, p. 307].

In the serial protocol, the target description can be requested by the `qXfer` packet, with an annex of `target.xml`, or via the `show tdesc` command in the frontend [SPS24, p. 851]. The `qXfer` packet is described in more detail in Section 4.1.4.

2.3 Executable and Linkable Format

The Executable and Linkable Format (ELF) is a binary representation of a program [95, p. 1]. There are three different types of ELF files: (i) relocatable files, (ii) executable files and (iii) shared object files [95, p. 1].

An ELF file always starts with an *ELF header*, after that, the file contains other elements like a *program header table*, multiple *sections* and a *section header table* in no particular order [95, p. 2].

The ELF header explains the layout of the file. The sections hold, among other things, symbols or actual instructions. The program header table is solely needed for program execution, whereas the section header table, which is comprised of information about the file's sections, is required while linking [95, p. 2].

Some of the sections in the ELF file are predefined. For example, the `.text` section contains the instructions of the program, while the `.debug` section, or any other starting

[5] The directory can be found here: `https://sourceware.org/git/?p=binutils-gdb.git;a=tree;f=gdb/features;h=14359c1c9c7c6f3ba509cda7637873666f2d62ab;hb=HEAD` (visited on 07/17/2024)

with that particular wording, hold debug information, which is not further specified by the ELF standard. Furthermore, the `.line` section accommodates data for line number mappings [95, pp. 15, 16].

2.4 DWARF Debugging Information Format

DWARF is a format for packaging debugging data into an executable file. The core concept of DWARF is to create a format independent to any compiler or debugger, while also maintaining a high level of abstraction in order to support any programming language on any architecture and Operating System [17, pp. 1, 3, 4].

Internally, DWARF uses so called Debugging Information Entries (DIEs) to represent the source program. A DIE consists of a tag, identifying the entry, along with multiple attributes [17, p. 15].

Debugging Information Entries might also *own* other DIEs. Ownership is represented by referencing the children DIEs. Generally speaking, all DIEs together creates a tree structure where nodes of that tree are DIEs referencing its children [17, p. 25].

These DIEs are usually placed in the `.debug_info` section of the object file [17, p. 15].

DWARF also supports line information, which is placed in the `.debug_line` section, which is referenced by the aforementioned `.debug_info` section. Since this section generally contains a lot of data, it is encoded to save some space. The encoding is done by creating a state machine, that, upon execution creates the necessary line information data. Only the *input* to that state machine is then saved in the object file [17, pp. 148, 149].

2.5 Debug Adapter Protocol

The DAP is an abstract protocol that connects development tools, such as Integrated Development Environments (IDEs), and concrete implementations of debuggers in a standardized way. The idea is, that different development tools can implement a generic debugger, that consumes data from an actual concrete debugger. The tool then only has to support the DAP, and any debugger that is able to communicate via that protocol can be used.

Since it is not assumed, that already available debuggers were to be rewritten to communicate via the DAP, another component is needed. The so called Debug Adapter (DA). A debug adapter is an intermediary between the DAP interface and a concrete implementation of a specific debugger. Each of the concrete debuggers simply need a corresponding adapter to be able to work with the DAP.

This architecture has the beneficial implication, that a debugger with a corresponding adapter only has to be written once, but can then be used in any tool that supports the Debug Adapter Protocol [24h].

2.6 Vienna Architecture Description Language

The Vienna Architecture Description Language is a language for characterizing processor architectures and Instruction Set Architectures (ISAs). Additionally, with VADL a whole family of tools and artefacts concerning the described processor can be generated. Among these tools is a complementary Instruction Set Simulator (ISS), capable of simulating any processor described by VADL [Him+24, p. 1]. Furthermore, LLVM artefacts are also generated, which can then be used to employ LLVM to create suitable compilers, assemblers and linkers [Him+24, p. 3]. Considering these features, VADL is an impressive tool for the development and analysis of processors with arbitrary architectures.

CHAPTER 3

Approach

This chapter addresses the aim of this project along with illustrations for basic requirements. Furthermore, the necessity of this work is also discussed.

The ultimate goal for this project is the ability to debug a program running on a simulator, defined and generated through the VADL ecosystem, on the Central Processing Unit (CPU) cycle level. This would include a Visual Studio Code extension handling the frontend of the debugger and a stub for the connection to the simulator. Ultimately, the simulator together with the stub would run on a server, which could be accessed over the internet.

Unfortunately, due to time constraints, only part of these ambitions could be completed. Any remaining work or ideas will be discussed in Section 7.1.

While debugging programs which are executed on the Instruction Set Simulator is already possible [Sch20, pp. 44–46], the fundamental idea for this project was the aspiration to debug programs running on an ISS with conventional debuggers and tools. More specific, the tech-stack chosen included the widely known debugger GDB and lightweight IDE Visual Studio Code (VS Code) as it can be used both as a text editor as well as a frontend for debugging. As a starting point, this project was implemented using an ISS, which is not capable of a cycle accurate representation.

GDB was not only chosen because of its popularity, but also because of its well documented serial protocol. This protocol and its corresponding stub serve as a sound base for debugging a simulated processor. The reason being, that we do not want to debug the actual simulator, but rather the program being run by the simulator, also known as the *guest* [Him+24, p. 5]. Thus, simply debugging the simulator with an ordinary debugger is not an option.

Furthermore, a valuable side-effect of using the GDB serial protocol, is the ability to also use the conventional GDB distribution to connect to the stub. Thus being able to debug

the program without the need for the VS Code extension, albeit having a few limitations explained in depth in Section 5.2.1.

In order to achieve this, the ISS has to be augmented by a GDB stub, taking over the role of the target, with the counterpart of this stub being a VS Code extension implementing the host part.

3.1 Requirements

The requirements for this project can be separated into two categories: technical requirements and essential features. The technical requirements describe how this project should be implemented, while the latter is a set of necessary features in order to achieve the goal.

Since the stub will be linked into the ISS, and the generated simulator might simulate a processor with an arbitrary architecture, the stub must be written in an architecture agnostic way. Additionally, as the eventual objective is to let the simulator run in the cloud, the stub must be accessible via the internet. The protocol, that handles the data communication between IDE and ISS must suit the needs for debugging on a CPU-cycle level, especially for arbitrary architectures.

The latter requirements for the features reflect the desire for effective debugging. The most important demand is the ability to interrupt the flow of execution via breakpoints. To then reason about the program, reading the contents of the registers of the processor is of utmost value. Reading the contents of the registers is beneficial in and of itself; however, setting the values of the registers is also a valuable tool for easily experimenting with the written program. For this project, the ability to view the disassembly was also regarded as important. This allows for more detailed debugging of source instructions that are expanded into multiple machine instructions. Such instructions are commonly known as *pseudo instructions*. Displaying the disassembly together with the addresses of the instructions also allows for breakpoints at exact addresses, not only source file lines.

CHAPTER 4

Implementation

This chapter presents a detailed description of the implementation of this project is discussed. First, the remote stub is described in detail, and then the Visual Studio Code extension is illustrated.

4.1 Stub

Conceptionally, a program should only have a single remote stub. To facilitate that, the stub is realized using the *singleton*[1] design pattern. This restriction also makes sure, that only a single connection to the stub at a time is possible. Additionally, to make linking easier, the whole stub is contained in a single header file.

The stub has two main tasks: (i) servicing the host via the serial protocol, and (ii) interrogating and controlling the ISS it is linked into.

The overarching procedure for serving the remote protocol is simple. Assuming a connection to the host has already been established, the process of handling requests consist of (i) waiting for a packet to arrive, (ii) performing a task, specific to that packet and (iii) responding to the host appropriately. These three operations are executed in a loop indefinitely, as shown in Figure 4.1.

4.1.1 Connection Establishment

First, the implementation of the connection creation is described. This connection to the host is the central pillar for the stub, since it is essentially inoperable without it.

Connection-based communication is used, which can be reached via an Internet Protocol Version 4 (IPv4) address. If not further specified, the target will assume the port number

[1] See Gam95, page 127 for more information about this pattern.

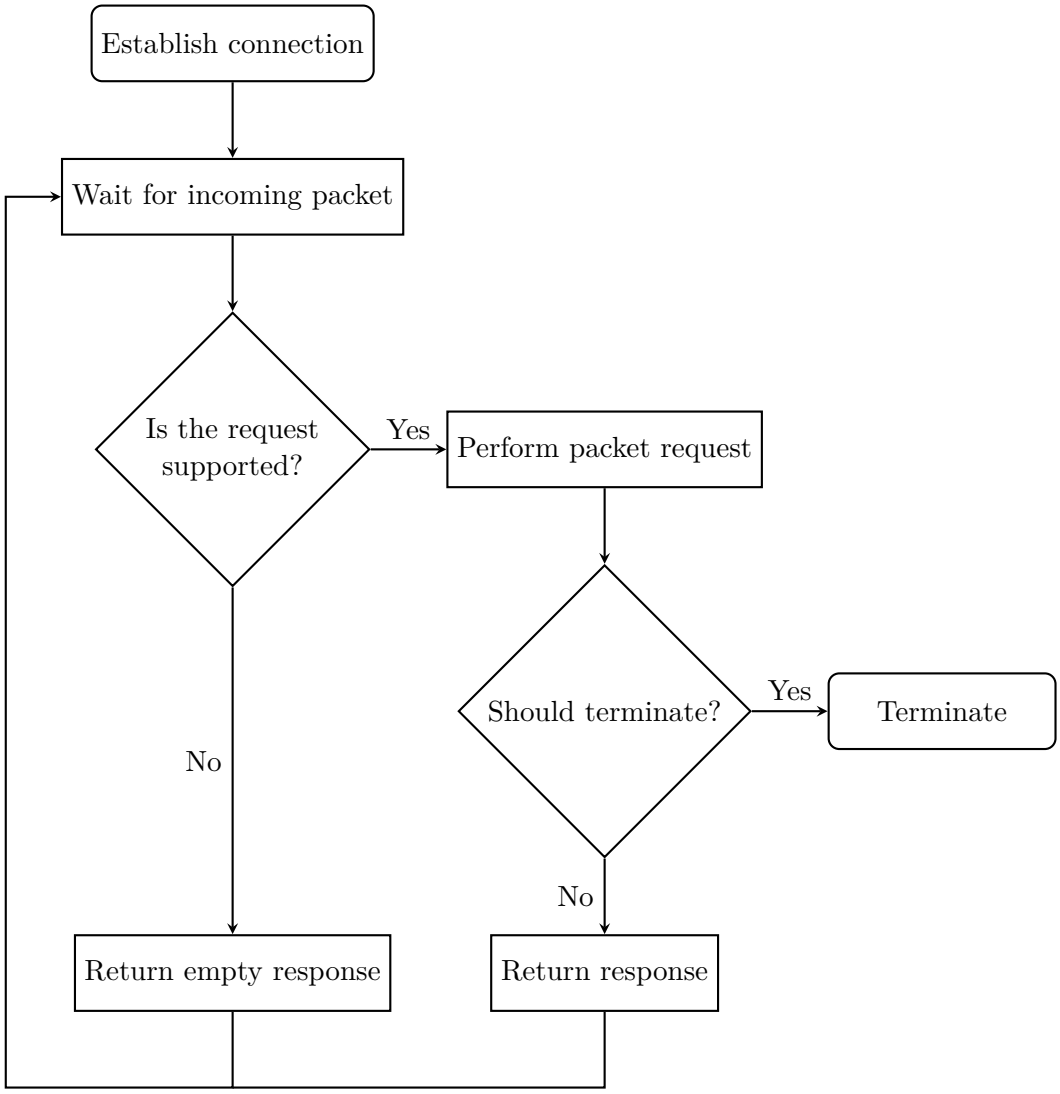

Figure 4.1: This flowchart illustrates the core loop of the stub.

to be 2159. This number was chosen as the default, as it is the assigned port number for the "gdbremote" service by the Internet Assigned Numbers Authority (IANA)[2].

4.1.2 Packet Management

In order to service the host, the stub needs functionality for sending and receiving packets for the serial protocol. In this section, packet sending and receiving will be outlined.

[2]Details on the assigned port number can be found here: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?search=2159 (visited on 07/17/2024).

As already mentioned in Section 2.2.1, the GDB distribution comes with a few implemented remote stubs. But these stubs are written for specific architectures, not architecture agnostic [SPS24, p. 329]. This is a problem, since the goal for this project is a single stub that can be linked into the generated ISS, regardless of its architecture. That means, a generic implementation of a remote stub is needed.

The GDB manual does state, that for implementing a custom stub, one could use an existing stub for guidance. Especially the stub written for the SPARC architecture `sparc-stub.c`[3] is particularly well "organized" [SPS24, p. 329].

Said stub was used as a template for the stub implementation. In particular, the functions that handle the packet transmission process: `getpacket` and `putpacket`. Besides that, the decoupling of sending and receiving characters from transmitting whole packets was also replicated.

Handling the Sending of a Packet

The aforementioned function `putpacket` will send a given packet one character at a time, until it receives a positive acknowledgement from the host.

The implementation of this function was followed closely from the SPARC stub, and can be summarized with the pseudo-code displayed in Algorithm 4.1.

In that example, the parameter P represents the packet to be sent, and can be thought of as a string, containing only the packet-data. The variable s keeps track of the checksum. P_{start} and P_{end} stand for the start and end packet characters respectively — as mentioned in Section 2.2.3 — analogously, A_{ok} and A_{err} stand for the acknowledgement characters, as seen in Section 2.2.4.

Handling the Receipt of a Packet

The function `getpacket` from the SPARC stub implementation will wait until a whole packet was received. Internally, it consumes all incoming characters, until a complete packet could be formed. If a received packet does not match the appended checksum, it will communicate that error to the host, by responding with an error acknowledgement, and waiting for the new transmission. Thus, this function will always return a complete packet.

These characteristics can also be found in the implementation of the corresponding functions in the project. The former can be outlined with the following pseudo-code 4.2. In this illustration, s stands for the checksum of the received packet, as calculated by the receiver. The variable s_t is the actually transmitted checksum of the packet. The latest character read is stored in the variable c.

[3] The source file for the SPARC architecture can be found here: https://sourceware.org/git/?p=binutils-gdb.git;a=blob;f=gdb/stubs/sparc-stub.c;h=24631cebc43e775c9338a2a31c2b2c920e0593b1;hb=dda83cd783075941aabe9b0292b004b11f00c831. (visited on 05/27/2024)

Algorithm 4.1: The `sendPacket` function

```
1  function sendPacket(P) is
2      repeat
3          s ← 0  /* checksum                                   */
4          sendChar(P_start)
5          foreach c ∈ P do
6              sendChar(c)
7              s ← s + c
8          end
9          sendChar(P_end)
10         send checksum s
11     until getChar() = A_ok
12 return
```

Returning the buffer b is thought of as returning the contents of it until and including the last character put into it, which is essentially the packet data of the incoming packet.

Note, that this implementation, and thus the whole stub, does not support notifications, as it ignores any characters before receiving a packet start character. This is in line with the specification [SPS24, p. 817].

4.1.3 Controlling the Simulator

The stub does also need to be able to interrogate and control the simulator. Interrogation is achieved via internal Application Programming Interfaces (APIs) from the VADL system. The manipulation of the execution process is described in the following section.

Stepping through the Simulator

Usually, the simulator would execute one CPU-cycle after the other, but for debugging, the stub needs to take control over this part of the simulator. In stark contrast to the ISS, the stub is not generated by the VADL framework, and cannot respond directly to different architectures. In order for the stub to execute one CPU-cycle, this action has to be implemented in an abstract architecture agnostic way, decoupling it from the stub's implementation.

This is realized by encapsulating this logic into a separate function, that can then be referenced in the stub. Since the target is a single threaded application, meaning the ISS and the stub run on the same thread, executing one cycle is as simple as calling the function once. The signature of the function, containing logic for a single cycle can be seen in Listing 4.1.

Algorithm 4.2: The `getPacket` function

```
1  function getPacket() is
2      s ← 0 /* checksum                                              */
3      s_t ← 0 /* transmitted checksum                                */
4      c ← 0 /* current character                                     */
5      while true do
           /* Consume incoming chars until a packet start char has been
           read.                                                      */
6          while (c ← getChar()) ≠ P_start do
               /* ignore character                                   */
7          end
8          while buffer b is not full do
9              c ← getChar()
10             if c = P_start then
11                 s, s_t ← 0
12                 clear buffer b
13                 continue
14             end
15             if c = P_end then
16                 break
17             end
18             s ← s + c
19             append c to buffer b
20         end
21         s ← s mod 256
22         if c = P_end then
23             if s = s_t then
24                 sendAcknowledge(A_ok)
25                 return b
26             else
27                 sendAcknowledge(A_err)
28             end
29         end
30     end
31 return
```

Listing 4.1: Signature for the single CPU-cycle function

```
void(TestBench<TCPU> &tb,
    BoundCheckingModule<TCPU> &boundCheckingModule,
    int32_t &result,
    bool &should_break)
```

The first two arguments are part of the generated API for controlling the simulator. The third argument `result` will store any error value, while the final argument `shouldBreak` will indicate to the stub, whether it should break, thus terminating simulation.

4.1.4 Supported Packets

Here, all packets that the stub supports are listed.

`?` This packet is used by the host to query the initial halting reason [SPS24, pp. 767–768]. Its response is a stop reply packet, as mentioned in Section 2.2.7, and will always be the `SIGTRAP` signal number[4].

`c` Continues the execution until the next breakpoint [SPS24, p. 768]. The optional argument of this packet for continuing at a specific address is not supported and will therefore always be ignored.

`g` Reads the general registers [SPS24, p. 769]. The order of the registers that are read is defined in the target description, as described in Section 2.2.8. Additionally, the target description also specifies what registers are belonging to the general section.

`G` Writes to the general registers [SPS24, p. 769]. This packet works analogously to the previous one. If not enough data is given for this packet, a standard error response with the error code 0 will be returned, as specified in Section 2.2.7.

`k` Kills the debugging process [SPS24, p. 770]. Upon receiving this packet, the core stub loop mentioned in Section 4.1 is terminated. No response will be sent.

`m` Reads memory at the given address by the specified amount of addressable memory units [SPS24, p. 770].

`M` Writes memory at the given address [SPS24, p. 771]. If not enough data is provided, the stub responds with a standard error response with error code 0, as specified in Section 2.2.7.

`p` Reads a specific register [SPS24, p. 772]. The register is selected via the *register number* as defined in the target description.

`P` Writes to a specific register [SPS24, p. 772]. Similar to the previous packet, the register is selected via the register number.

`qAttached` Returns whether the host attached itself to the debuggee, or launched it [SPS24, p. 805]. Since the stub only supports single threaded applications, the optional Process Identification (PID) will always be ignored. Moreover, since the stub cannot be attached to a running process, the stub will always return 0, indicating, that the stub created the debuggee process.

[4] The signal number for `SIGTRAP` is defined as 5 in the `signals.def` file.

qSupported Returns all features that the stub supports [SPS24, pp. 791–799]. The stub will always return the `PacketSize` feature, as well as all supported objects, the host may read from via the `qXfer` packet. The packet size defines the maximum number of characters in the packet data the stub supports [SPS24, pp. 795, 796]. Currently, the stub has a packet size defined as 2048. If a larger packet is received, the stub will ignore it and wait for the next packet.

qXfer:object:read:annex Requests the data of a specific *annex*, which is part of an *object* [SPS24, p. 801]. If the given object is not supported or available, the stub responds with an empty response. However, if the object is supported and available, but the annex is not, the stub replies with a standard error response, with the error code 0. How the `qXfer` packet is implemented is illustrated in the following Section 4.1.4.

s Performs a single step [SPS24, p. 771]. This translates to a single CPU-cycle in the ISS. The optional argument of this packet for resuming at a specific address is not supported.

z Removes the breakpoint at a specific address [SPS24, pp. 775–776]. Only software breakpoints (type 0) are supported by the stub. Additionally, the breakpoint-kind will always be ignored. The breakpoint-kind is architecture specific, and usually indicates the size in bytes of the breakpoint [SPS24, p. 775]. If the breakpoint could not be removed, a standard error response with error code 0 will be returned.

Z Adds a breakpoint to the given address [SPS24, pp. 775–776]. Similarly to the previous packet for removing breakpoints, only software breakpoints are supported. Again, the breakpoint-kind will always be ignored. Furthermore, conditional breakpoints are not supported, thus ignoring the optional conditional list.

General Query Transfer Packet

Packets that start with a `q` are part of the *general query* packets. Together with the *general set* packets, starting with a `Q`, they provide a family of packets for transferring data between the host and the target [SPS24, p. 781].

The specific packet out of the general query packets, that is most important to this implementation is the *general query transfer packet*, starting with `qXfer`. For the following sections, this packet is referred to as *transfer packet*.

The `qXfer` packet has the following form:

```
qXfer:object:read:annex:offset,length
```

Here, the transfer packet with the *read* operation has been chosen. This packet here asks the target to read from a given *annex*, which is located in the specified *object*. To be more precise, exactly *length* bytes should be read, starting from the given *offset*. The target will then respond with the data, also indicating whether there are more bytes left to read [SPS24, p. 801].

This transfer packet is used to transfer line information, disassembly information and target descriptions from the target to the host. For the stub, objects are treated as directories, while annexes are interpreted as files.

The host needs to know what objects and annexes are available for querying. To inform the host about the possible transfers, the response of the `qSupported` packet is used [SPS24, pp. 791–799].

4.1.5 Target Description Details

Section 2.2.8 already describes how target descriptions work in theory. While the target description already carries a lot of information, they do not provide enough data for a complete architecture agnostic host. In order to solve that problem, the target description is augmented by new elements. The new elements are united under a common new element, the `details` element.

Listing 4.2: Example for target description details.

```
<?xml version="1.0"?>
<!DOCTYPE details>
<details version="1.0">
    <pc regref="pc" />
    <instructions size="4" />
    <memory minAddressableUnitSize="1" />
</details>
```

In order not to disturb the original document type, the details are placed in a separate file, which can then be included into the original target description [SPS24, p. 852]. This makes the new target description details compatible with the generic GDB implementation, since they ignore unknown tags by default [SPS24, p. 853]. Notably, the reference to the target details Document Type Definition (DTD) is omitted, otherwise, the target description parsing will not work for GDB. Locally, the file can still be validated against its corresponding DTD `target-details.dtd`.

Similar to the target description, the details also have a version attribute. This allows quick detection, if the DTD changes.

The new elements can be seen in Listing 4.2, their purpose is outlined in the following paragraphs:

`pc` This contains information for the Program Counter (PC) register. Currently, only one attribute, `regref` is available. It references the PC out of all registers in the target description by its name. The name instead of the register number has been chosen for reference, since it is mandatory to provide one when describing the register, and it has to be unique for the whole target description [SPS24, p. 855]. While the register numbers are also unique [SPS24, p. 855], they may change if the number of registers is modified.

But more importantly, register numbers do not have to be mentioned explicitly [SPS24, p. 855], thus reducing readability.

`instructions` Similarly to the previous element, the `instructions` element provides additional information for the instructions. At this point, only one attribute can be provided. It signals the size of one instruction in bytes. Due to this implementation, only instructions of equal size are supported, although it is conceivable to expand upon that further in the future, allowing different sized instructions.

`memory` The third element, the `memory` element contains additional information about the targets memory. The sole attribute allowed describes the size of the smallest addressable memory unit size in bytes. This is needed for example for the packets `m` and `M` [SPS24, pp. 770, 771].

Strictly speaking, an explicit value for the minimal addressable memory unit size is not required, as one could just use the `m` packet with a length of 1 to read a single addressable memory unit. The size could then be determined by the size of the response. However, the response may carry less data, if part of the requested memory region could not be read, resulting in a faulty assumption over the minimal addressable memory unit size [SPS24, pp. 770, 771].

4.1.6 Line Information

On the target side, a mapping between source code line numbers and the respective PC value is generated. To understand why this is necessary, we have to anticipate the inner workings of the DAP. Under the DAP, many request refer to the source code line number[5], since it is the location the developer sees, when stepping through the code. The GDB serial protocol, however, does not work via source code line numbers. For example, the packet for breakpoints requires the address for the breakpoint location [SPS24, p. 775].

The mapping between source line numbers and addresses are extracted directly from the executable. The compiled executables from the VADL ecosystem are in the Executable and Linkable Format. When compiling with debug symbols, the wanted mappings are stored using the DWARF format[6].

As already mentioned, DWARF debug information is stored in the object files. Since they follow the ELF format, the ELF parsing library (libelf)[7] was used to navigate the object file.

[5] For example, when setting a breakpoint, the line number (and optionally the column number as well) is used to indicate the location. [24a]

[6] The assembly is done via `llvm-mc`. To get debug symbols the `-g` flag has to be appended. This generates DWARF debug symbols [24f].

[7] The manual pages for `ELF(3)` can be found here: `https://man.freebsd.org/cgi/man.cgi?query=elf&sektion=3&manpath=FreeBSD+14.0-RELEASE+and+Ports` (visited on 08/04/2024).

The debug information could then be read using the DWARF access library (libdwarf)[8]. Since line information is stored per Compilation Unit (CU) [17, p. 148], the needed data is accumulated by iterating over all CUs in the object file[9] and querying for the source lines in the current CU, as shown in Algorithm 4.3. The execution of the state machine in charge of generating the line information is handled by the libdwarf library.

Algorithm 4.3: Illustration how the line information is extracted.

```
1 foreach CU ∈ object file do
2     foreach top level DIE ∈ the current CU do
3         L = all source lines stored in the current DIE
4         foreach l ∈ L do
5             get and save detailed information from l
6         end
7     end
8 end
```

Stucture

These mappings are stored in a separate file in Comma-Separated Values (CSV) format. To be more precise, it follows the definition presented in Requests for Comments (RFC) 4180[10].

The Figure 4.2 describes a single line for the CSV file. As specified, the different values are separated by a comma[11], and each line is terminated by a Carriage Return Line Feed (CRLF)[12] combination [Sha05]. The generated CSV file does not contain a header.

`source filename` The full path of the source file.

`address` The address corresponding to the generated machine instruction [17, p. 150].

[8] The manual pages for `DWARF(3)` can be found here: `https://man.freebsd.org/cgi/man.cgi?query=dwarf&sektion=3&manpath=FreeBSD+14.0-RELEASE+and+Ports` (visited on 08/04/2024).

[9] The implementation follows the examples of the manual pages for `DWARF_CHILD(3)` and `DWARF_SRCLINES(3)` available here: `https://man.netbsd.org/NetBSD-7.0/dwarf_siblingof.3#EXAMPLES` and here: `https://man.freebsd.org/cgi/man.cgi?query=dwarf_srclines&sektion=3&manpath=FreeBSD+14.1-RELEASE+and+Ports#EXAMPLES` (visited on 08/04/2024) respectively.

[10] This RFC can be found here: `https://datatracker.ietf.org/doc/html/rfc4180` (visited on 25/07/2024).

[11] The specification in the RFC defines a comma to be the character with an ASCII value of `0x2c` [Sha05].

[12] Again, the RFC precisely defines these characters. The Carriage Return is the character with ASCII value `0x0d`, and the Line Feed is the character corresponding to an ASCII value of `0x0a` [Sha05].

source filename	address	line number	column number	is begin statement
`string`	`unsigned integer`	`unsigned integer`	`unsigned integer`	`bool`

is begin block	is end sequence	is prologue end	is epilogue begin	ISA	discriminator
`bool`	`bool`	`bool`	`bool`	`unsigned integer`	`unsigned integer`

Figure 4.2: The structure for the line mappings CSV file.

`line number` The line number in the source file. Line numbers start at 1, if instructions cannot be mapped to a line number, the compiler might output a 0 for the line number [17, p. 151].

`column number` Similar to the line number, the column number represents the column within a line from the source file. Again, they start with 1, the value 0 indicates, that the statement starts before any characters in the line [17, p. 151].

`is begin statement` A Boolean indicating if this address represents the start of a statement. This is a suitable location for breakpoints [17, p. 151].

`is begin block` A Boolean signaling that this address is the beginning of a *basic block*. A basic block is a "sequence of instructions where only the first instruction may be a branch target and only the last instruction may transfer control" [17, p. 150].

`is end sequence` A Boolean marking the end of a *sequence*. A sequence is a "series of contiguous machine instructions" [17, p. 150].

`is prologue end` A Boolean indicating that the current address is the end of the prologue. Therefore, a function breakpoint should be set here [17, p. 151].

`is epilogue begin` Similar to the previous Boolean, this marks the start of the epilogue and thus is a suitable spot for the last breakpoint in a function [17, p. 151].

`isa` The encoded ISA for the current instruction [17, p. 152].

`discriminator` This identifies the block to which the current instruction belongs [17, p. 152].

The most important values for this project are the source filename, the address along with the line number and the flag indicating whether the current instruction is the beginning of a statement or not.

Requesting the Line Information

The host is then able fetch these mappings via the `qXfer` packet. The benefit of this approach is, that the serial protocol does not have to be augmented with new custom packets in order to support these line number mappings. The host is informed about the mappings file via the `qSupported` response, which indicates the appropriate object and annex for the transfer packet.

Known Issues

For larger source files, this line information file will get huge quickly. The main contributor for the size is the filename, which will be repeated for each line in the CSV file.

The current implementation was chosen because of its simplicity, but there is definitely room for improvement here, which will probably result in a different format than CSV.

4.1.7 Disassembly

The frontend of the debugger is also capable of displaying disassembly. The required data to display the disassembly is generated once for the whole source code on the target side. This is achieved by employing the object file dumper from LLVM called `llvm-objdump`[13]. It reads and outputs the contents of a supplied object file. Since this object dumper is generated through the VADL system, it is best suited for understanding and therefore disassembling the instructions [Him+24, p. 31]. Only executable sections are disassembled, which is achieved with the `-d` flag.

Security Concerns

The target uses a subprocess to evaluate the `llvm-objdump` command. Since the filename, which is supplied as a positional argument to the command, is not sanitized, it poses a critical threat to security, potentially allowing arbitrary code execution. This is especially important to note, as the user has control over the filename.

Due to time constraints, this matter has not been further revised.

Parsing the Object File Dumper's Output

The output of `llvm-objdump` starts with a header displaying the file format of the passed in file. Next, the disassembly for each section is output. The output for one section is comprised of a section header, followed by the disassembled instructions. Each disassembled line starts with an address, followed by an optional label. After that, a semicolon prior to the actual data encoded in hexadecimal digits can be found. Last, the instruction is printed. If an address has a label, it does not contain any data or instructions. The same address may appear multiple times in the output.

[13] More information about `llvm-objdump` can be found here: `https://llvm.org/docs/CommandGuide/llvm-objdump.html` (visited on 04/08/2024).

In order to extract the required information, regular expressions are used. Since multiple parts of one line are of interest, groups are used to match the different components.

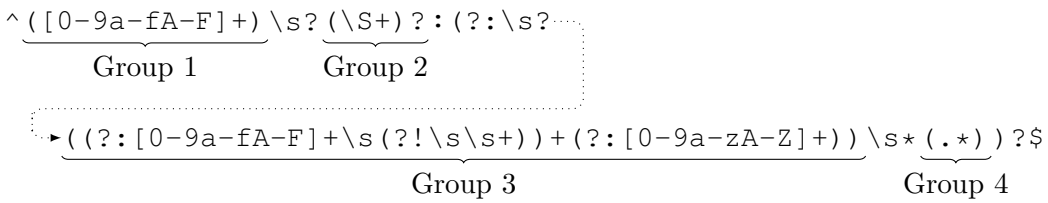

Figure 4.3: An illustration of the used regular expression pattern using the ECMAScript flavor.

The first group matches the hexadecimal address. Secondly, any optional label is matched. After that, the raw hexadecimal data is captured, and finally, the instruction is matched with the last group.

As can be seen in Figure 4.3, the third group is quite complex. This stems from the aspiration to not have to clean up any matches from the regular expression pattern afterwards. The data to be captured with the third group is a series of two hexadecimal digits followed by a single space between these bundles. But simply matching two hexadecimal digits with a following space would then also match the last whitespace, which is not part of the wanted data.

If the object file dumper cannot disassemble a specific instruction it will output the string `<unknown>` instead of the disassembled instruction [24g]. Internally, any instructions exactly matching that string will be replaced with an empty string. The frontend may then decide how to handle empty instructions, regardless of what disassembler was used on the target side.

Structure

Likewise to the line information discussed in the previous section, the disassembled data fetched via the `llvm-objdump` utility is persisted in a CSV file. Again, the definition of RFC 4180 is followed.

address	label	bytes	instruction
number	string	string	string

Figure 4.4: The structure of the disassembly CSV file.

The structure is quite simple, as can be seen in Figure 4.4. Since the instruction might contain commas it is important to wrap it in double quotes, as stated by the specification [Sha05]. The labels must not contain any double quotes or commas.

`address` The address in hexadecimal format

`label` The label, or an empty string if no label is present.

`bytes` The actual data in the same format as produced by the LLVM object file tool, or an empty string if not present.

`instruction` The instruction as output by the object file dumper. This element will always be encased in double quotes if present. When not present, or not able to be disassembled, an empty string is placed instead.

Requesting the Disassembly

Similar to the transmission of the aforementioned line information, the disassembly is also relayed to the host via a `qXfer` packet. Again the `qSupported` packet is used to inform the host about the possibility to request this information.

Known Issues

The disassembly file will grow in proportion to the number of instructions in the source file. This will have an effect, if a more abstract language is used, generating many more machine instructions than statements in the source code.

4.2 Visual Studio Code Extension

The complementary part to the stub is a debugger extension for VS Code. It serves as a host for the serial protocol while simultaneously providing a generic User Interface (UI) to the user.

The core task of this extension is to manage the interface between two different protocols. Between Visual Studio Code and debugger extension, the Debug Adapter Protocol is used, while the extension communicates with the stub using the already familiar GDB remote serial protocol.

The extension development process was influenced heavily by the debugger extension guide[14] from VS Code. The underlying folder and file structure for the extension was generated as suggested using *yeoman*[15] together with a code generator especially designed for VS Code extensions[16] [24k].

[14] The guide can be found here: `https://code.visualstudio.com/api/extension-guides/debugger-extension` (visited on 08/10/2024).

[15] The website can be accessed here: `https://yeoman.io/` (visited on 08/10/2024).

[16] The package can be found here: `https://www.npmjs.com/package/generator-code` (visited on 08/10/2024).

While the implemented Debug Adapter (DA) is independent of VS Code and could be used to bring debugging functionality to a variety of other tools that support the DAP[17], this implementation and the following sections are written with regards to VS Code.

4.2.1 Contributions

The outline of the extension is defined in the `package.json` file. This includes metadata like the name or version of the extension, but also so-called contributions. Contributions declare the overarching category for the features the extension provides. In this case, the extension defines two contributions: (i) breakpoints, and (ii) a debugger.

Breakpoints Contribution

The contribution of breakpoints is necessary to enable the user to set breakpoints. Additionally, a language ID is required, which VS Code utilizes in order to enable specific capabilities based on the language in use [24i]. In this case that would be the ability to work with breakpoints on source file with the given language ID.

VS Code does not support language IDs for assembly by default [24e]. In order to avoid conflicts, which could occur when defining custom language IDs specifically for this project, an already existing extension[18] for working with assembly in VS Code was chosen as a dependency. This extension defines language IDs for assembly alongside additional quality of life features like syntax highlighting. Another extension, that is declared a dependency is the hex editor from Microsoft[19] which is necessary to view the targets memory. The extension's dependencies are expressed in the `package.json` file under the `extensionDependencies` key. Moreover, as defined in the package manifest, the breakpoints are enabled for the `asm-collection` language ID.

Debugger Contribution

The debugger contribution specifies a debugger of type `vadl` with corresponding configuration attributes. These attributes can be understood as arguments to the debugger and can be changed by the user. How this is done exactly will be elaborated on in Section 5.1.

4.2.2 Serial Protocol Host

As already mentioned, the VS Code extension should serve as the host for the serial protocol. There are a few different ways to achieve that goal. The initial idea was to use GDB itself as the host and connect to the stub as described in Section 2.2.2. The

[17] A list of supported tools is available here: `https://microsoft.github.io/debug-adapter-protocol/implementors/tools/` (visited on 08/25/2024).

[18] The extension's ID is `maziac.asm-code-lens` and its marketplace page can be found here: `https://marketplace.visualstudio.com/items?itemName=maziac.asm-code-lens` (visited on 09/04/2024).

[19] The extension's marketplace page can be found here: `https://marketplace.visualstudio.com/items?itemName=ms-vscode.hexeditor` (visited on 09/04/2024).

extension then only has to manage running a GDB instance. This is actually anticipated by the GDB implementation, as it provides a machine-readable mode specifically tailored to this use case. The so called GDB/MI interface is a "line based machine oriented text interface to GDB" [SPS24, p. 583]. It allows using GDB as a small part constituting a larger project.

This approach is definitely viable and in use[20], but it requires the user to have not only a working, but also the correct GDB installation on their system. This problem also occurs when using GDB with the built-in DAP interface, which would even make the DA obsolete. In the circumstances of this project, having the correct installation is of utmost importance. Recalling, that GDB has to be compiled for a specific target architecture, the task of building their GDB distribution from the source code appropriately to a given architecture would fall onto the end user. This is not only demanding for the user but also susceptible to errors.

To avoid this premise, the next idea was to ship the extension with the GDB source code and to compile it for the correct architecture upon starting the extension. This concept also comes with its own set of problems. For example, the source files of GDB version 15.1 is around 212 MB[21]. Compiling the source without any additional components takes around 15 min[22]. Furthermore, there would be the large overhead of compiling GDB for an arbitrary architecture. While `gdb-multiarch`[23], a precompiled GDB instance supporting multiple architectures, would work with the most popular architectures, it would not solve the issue of using GDB with a custom architecture.

Considering these obstacles this approach was aborted. Finally, the whole idea of using a GDB installation locally was scrapped, in favor of a custom GDB serial protocol host side implementation.

Packets

For the host, a more abstract approach for the packets was chosen. This is reflected by having dedicated implementations of the different packets with custom constructors, hiding the actual packet data from the developer. Besides the constructor, a packet also contains functionality to parse its corresponding response. Why this is important is exemplified in the following section.

When implementing a new packet three key aspects have to be considered: (i) the response type (ii) the actual implementation of the packet and (iii) the implementation of the response parser.

[20] The guide on C and C++ debugging by VS Code suggests a GDB installation [21a]. Furthermore, there are existing GDB extensions on the marketplace, like: `https://marketplace.visualstudio.com/items?itemName=coolchyni.beyond-debug` (visited on 14/08/2024) using this method.

[21] The source code was fetched from here `https://ftp.gnu.org/gnu/gdb/` (visited on 14/08/2024) and extracted using the 7-Zip file manager.

[22] This was timed on a Windows Subsystem for Linux (WSL) machine using the `time` command.

[23] The package can be found here: `https://packages.debian.org/en/sid/gdb-multiarch` (visited on 14/08/2024).

The response parser is an object, that provides functionality in order to dissect and extract data from the responses raw string form. This data is then captured by a requested response type. To facilitate that, the implementation of this parser is different for each response type.

While responses are technically also packets, inwardly, they are not represented by the common `GdbPacket` interface, rather, each response has its own interface. This is viable, as there is no need for a low level packet representation of a response.

Sending and Receiving Packets

The implementation for sending and receiving packets is akin to that of the stub. But contrary to the stub, the host has to wait for the corresponding response. This is usually the next packet, that is received. In order to retain this sequence of events, a lock is used, making it impossible to send two consecutive requests without handling the response[24]. There are also additional layers in place to make this send-receive mechanism usable in a more abstract way. More precisely, sending and receiving a packet is combined in a single function, which also handles standard responses. The implementation is outlined in Algorithm 4.4.

Moreover, the returned packet is parsed to make the extraction of the information easier. The internal API then presents the developer with a corresponding response object for each request.

Algorithm 4.4: Outline of the send-receive process.

```
1  function getChar(p) is
2      sendPacket(p)
3      r ← await readPacket()
4      if r = unsupported response then
5          throw
6      else if r = standard error response then
7          throw
8      else if p requires success response ∧ r ≠ standard success response then
9          throw
10     end
11     r_parsed ← parseResponse(r)
12     return r_parsed
13 return
```

[24] For packets that do not have a response — like the `k` packet — a special variation of this mechanism is in place, where waiting for a response is omitted.

Connection and Initialization

Conceptionally, the host is split into two parts, the *GDB client* and the *GDB session*. The former handles the connection to the requested address and port and providing raw functionality for sending and receiving packets, while the later serves as a more abstract view on the host. It provides functionality for specific actions on the target, like stepping or reading registers. The session is also incorporating packet sending like mentioned in the preceding Section 4.2.2. Additionally, the session emits events, that are subsequently used by the Debug Adapter Protocol. Further details may be found in Section 4.2.3.

Moreover, it is also responsible for the storage of additional data the host may require. The data comprises the following elements: (i) the target description, (ii) the features of the target, as queried by the `qSupported` packet, (iii) line information and (iv) disassembly.

Upon startup, first the supported features of the target are queried. Next, the target description is requested, followed by the line information. Subsequently, the disassembly is retrieved, before querying the stop reason. This marks the end of the initialization process.

When querying for the supported target features, the host also sends its features along. The only feature from the host is currently `xmlRegisters`. It tells the target, that the host is capable of understanding target descriptions. The host also includes the understood architectures via a target specific string. If the target can provide the target description for an understood architecture it will respond appropriately to this packet [SPS24, p. 792]. Since this host is architecture agnostic, it understands any target description. In order to convey this, instead of specifying an architecture, the keyword `any` is used. This is a deviation from the serial protocol.

Although the DAP specification defines both attaching and launching without allowing for the disabling of either, the implementation only foresees launching the program as a connection method. This limitation can be attributed to the fact that the stub is unable to attach to a running process, as outlined in Section 4.1.4.

Setting Breakpoints

The action of setting a breakpoint is provided and handled by the VS Code UI. Using the source line mappings, gathered from the stub, the breakpoint locations are translated into addresses. With the current implementation, there are a few limitations on where breakpoints can be set. Breakpoints cannot be set on labels, only on instructions. Any breakpoint, that is set on an illegal location will be indicated as unverified.

Using Target Data on the Host

The supported features, target description, line information and disassembly are parsed into native TypeScript (TS) objects for ease of use.

Supported Features The response for the `qSupported` packet is a list of features. Each feature then declares whether it is supported or not. This can be done in four ways:

`feature-` Signals, that the feature is not supported.

`feature+` Simply indicates, that the feature is supported.

`feature=value` Likewise to the prior method, this indicates that the feature is supported, while also providing an associated value with the feature.

`feature?` Means, that this feature may be supported. The host then has to find other ways to check for support [SPS24, p. 792].

The implementation for this project actually requires the `PacketSize` feature to be sent, as it does not have the capabilities to infer the maximum packet size of the target by other means. The default packet size of the host is — likewise to the implementation of the stub — 2048.

Due to simplicity, this host's implementation will treat any feature responses with the question mark as not supported.

Target Description After receiving the target description, all includes are resolved, by issuing another `qXfer` request with the new annex. Next, the whole target description is mapped onto a native TypeScript (TS) object. Since the target description might get very large, another object is employed to operate on the actual data. This approach, which involves the utilization of a more streamlined abstract operator to shield a subsystem, is referred to as a *facade* [Gam95, p. 185]. It provides an easy way for developers to access relevant data. The same mechanism is used for the line information and disassembly as well.

Currently, the target description implementation supports all but the `type` element. Notably, the retrieved target descriptions are not validated against their corresponding DTDs.

Line Information Similar to the target description, the line information is also first mapped onto an object, but then accessed via a user friendly facade.

Disassembly The disassembly is treated analogously to the line information.

4.2.3 Debug Adapter

The DA takes the role of the server for the DAP. The debugging tool, in this case VS Code, then sends requests to that adapter, in order to provide debugging functionality.

The DAP specifies not only requests, but also events the DA might emit. For example, for indicating the conclusion of the initialization phase, or to signal that the debuggee has stopped or that it has exited [24a].

These events should not be confused with the events, the GDB session internally emits, even though it emits events for the same reasons, e.g. when the debuggee stopped by various causes, or upon termination. Usually, the debuggee stops when a breakpoint is hit, but it may also be suspended upon reaching the main entry point[25]. While these events are strictly seen not identical, they are wired in such a way, that the DA may often just relay the received events from the underlying session to VS Code.

Any data the developer might be interested in is usually only accessible, when the execution stopped and the debuggee is in a suspended state. This is achieved, for example, when a breakpoint is reached. While in a suspended state, all objects which are accessible via the DAP, receive a so called *object reference*. If the client then requests an object, this is done via this object reference. These references only remain as long as the debuggee is interrupted, afterwards, they are newly assigned [24h].

For instance, when hitting a breakpoint, the DA emits a stopped event. The client is then eligible to examine variables or stackframes. In order to read variables, it is first necessary for the client to request all scopes for the current state. This allows the client to then examine the available variables via their references, as described in more detail in Section 4.2.3 and Section 4.2.3.

This DA is implemented as an *inline* adapter. That means, it is run directly by VS Code, not in a separate thread or server [24b].

Capabilities

After the initialization of the DAP connection, the adapter will communicate its so called capabilities. A capability can be thought of as a supported feature of the DA [24h]. The implemented extension has the following capabilities:

`supportsConfigurationDoneRequest` This request declares, that the client finished its initialization. If supported by the DA, the client will send this request [24a].

`supportsSetVariable` The DA is capable of changing the values of variables [24a].

`supportsTerminateRequest` The DA is able to terminate the debuggee gracefully [24a].

`supportsReadMemoryRequest` As the name suggests, the DA can read arbitrary memory [24a].

[25] How this is achievable is described in Section 5.1.2.

`supportsDisassembleRequest` The DA can access disassembly information about the debuggee, which further enables the disassembly view in VS Code[26] [24a].

`supportsInstructionBreakpoints` The DA can set breakpoints based on addresses [24a].

Threads

Via the `ThreadsRequest`, the client inquires about any threads of the debuggee. This request is not part of any capability, rather it is a core part of the DAP and has to be supported.

For this project, only a single thread is supported. This results in an identical response for any threads request.

Stackframe

Each thread contains a list of stackframes, the so called stacktrace [24h]. This implementation does not support multiple stackframes, as it is usually not needed when executing assembly code. That means, semantically, the whole user code is run in a single stackframe, that is constantly changing as the execution progresses.

If possible, the topmost stackframe also contains an instruction pointer reference. That is the current address, the PC points at. This is important, as the built-in disassembly view from VS Code would not work without it.

Scopes

Variables in the DAP are grouped in so called scopes. When a user requests the value of a variable, after the threads and stackframe requests, the client first issues a scope request, which then holds references to the actual variables [24h].

Registers are treated as variables in this implementation. They are grouped into a register scope. Since there are currently no more variables, only this scope exists. When variables are requested, the client uses the variable references, stored in the previously requested scope to inquire about their value.

Variables

Via the variables request, the client retrieves information about any variables, using the variables reference [24h]. Registers are treated as variables in this implementation. That means, in order to see the registers value, the corresponding variables have to be inspected.

[26] Only enabling the capabilitiy is not enough to use the disassembly view, as mentioned in Section 4.2.3 and 4.2.3.

Visual Studio Code introduced the memory view with version 1.64 [22]. The memory view displays raw data in a hex-editor-like style. If memory writing would be implemented, then this memory view would be used to modify binary data [22]. The debugging tool communicates the capability to display binary data to the DA in the initialization phase [24a]. VS Code has this capability [22].

In order to make a variables content available for the memory view, the `memoryReference` for the variable has to be set. The current implementation automatically sets this memory reference for all registers of the predefined type `data_ptr`[27] in the target description.

Disassembly

As of version 1.59, VS Code includes a native disassembly view, which is used to present the disassembly to the user, alongside with functionality to work with instruction level breakpoints [21b]. As aforementioned, an instruction pointer reference has to be provided in the topmost frame of the stacktrace, in order for the disassembly view to work expectedly.

When setting an instruction level breakpoint, the disassembly view has to be used. Only breakpoints on source line level are possible when setting them on the source file.

As implemented by VS Code, the disassembly view is only accessible while the debuggee is in a suspended state, such as on a breakpoint.

Considering the unnecessity when using assembly code, the inlay option, to show source code alongside the disassembled instructions, is currently not supported.

4.3 The Big Picture

To facilitate a more comprehensive grasp on the project's overall structure, an illustration is provided in Figure 4.5. It is easy to see, how the debugger is split into two parts, one residing in the VS Code extension and the other being the stub. Furthermore, the usage of the different protocols, the DAP and the serial protocol, can be seen clearly.

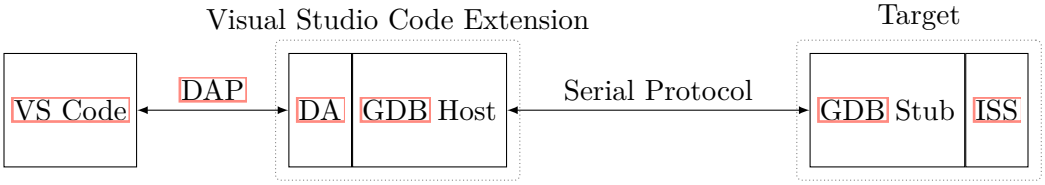

Figure 4.5: The overarching structure of this project.

[27] Usually, the Stack Pointer (SP) as well as dedicated address registers are marked as `data_ptr` [SPS24, p. 856].

CHAPTER 5

Usage

This chapter sets forth the intended way to utilize the stub in conjunction with the extension. As already mentioned, the VS Code extension is not strictly necessary in order to establish a connection to the stub. Furthermore, this chapter details the manner in which the stub may be used with a generic GDB installation, along with the constraints inherent to this approach.

5.1 Intended Usage

The intended usage for this project is to utilize the stub residing in the simulator together with the VS Code extension.

It is assumed that the reader is already familiar with the compilation of the source code for the ISS, as well as the setup of the VADL ecosystem, the generation of artefacts and tools together with their compilation.

5.1.1 Setting Up the Target

Following the compilation of the source code for the ISS, the additional artefacts required are the target description, along with its details. Both the target description and the accompanying details should then be placed under the `transfer/features` directory of the simulator.

Next, the ISS can be started. To enable debugging, the `-g` flag must be passed as an argument. Additionally, the source file that is to be executed on the simulator must also be provided.

Now that the ISS is running, it waits for an incoming connection from the extension.

5.1.2 Setting Up Visual Studio Code

Posterior to installing the extension with its dependencies, the language ID of the source file has to be set accordingly. Otherwise, breakpoints cannot be set. Next, in order to start debugging, a so called launch configuration is needed. In VS Code, launch configurations are used to configure and save details for the debugging process, regarding a specific debugger [24c]. The available attributes for this configuration is set in the extension's manifest. They are:

`type` The type of the debugger [24c]. This is `vadl` in this case.

`request` Whether the debugger should launch the debggee or attach to it [24c]. As already stated, only launching is supported.

`name` The name of the launch configuration, which will be displayed to the user [24c].

`program` The file to run, when starting the debugger. The default value for this attribute is `${file}`, which represents the current opened file [24j].

`stopOnEntry` A Boolean flag signaling whether the execution of the program should be interrupted, right after it has been launched. By default, this is set to be true.

`host` The IP-address of the remote target. The default value for this is `localhost`.

`port` The port of the remote target. The assigned port for GDB debugging, as stated in Section 4.1.1, is used as a default here.

`trace` A Boolean flag indicating if the events of the DAP should be logged[1].

The first three attributes are required regardless of the debugger [24c]. Moreover, for this debugger to work, the attributes `program`, `host` and `port` are also compulsory.

5.2 Without the Visual Studio Code Extension

Although not ideal, a local GDB installation can also be used instead of the extension. In order to facilitate a flawless execution, a few configurations have to be done beforehand.

The setup of the target is identical to that presented in the previous section. However, GDB demands special treatment. It is assumed, that the local GDB installation is able to understand the target's architecture.

Once the simulator has been launched, GDB can be started. Subsequently, a few commands have to be issued in GDB, before the debugging can begin. The initial step is to select the target's architecture. This may be achieved via the `set architecture` command. As the stub utilizes the big-endian notation exclusively, it is imperative that

[1] The logging does not seem to work, if the extension is run in the inline mode [jus23].

the host is also configured to interpret data the same way. This may be accomplished by the `set endian big` command. Ultimately, the host may connect to the target via the `target remote` command as previously outlined in Section 2.2.2.

When debugging the connection, `set debug remote 1`, `set debug xml 1` and `set debug arch 1` are valuable commands to gain insight in the data transmission and architecture information. These commands should be issued prior to connecting and setting the architecture respectively.

5.2.1 Limitations

As the utilization of the stub without the extension deviates from the intended usage, it is only natural, that there exist some limitations.

For example, disassembly and line information are imperceptible to the standard GDB implementation. The corresponding request is inline with the serial protocol's specification, but it lacks the understanding of what to do with that information. Therefore, GDB cannot provide disassembly or symbol information without passing it the file that is running on the simulator. To gain the maximum insight, it is therefore advised to pass the executable to GDB and the ISS simultaneously.

The limitations of not having insight into the executable is becomes apparent when using breakpoints. With no information about source file lines, the only way to set breakpoints is by referring to an address. Additionally, GDB has to infer the instruction size when stepping resulting in a large time duration to perform a single step. Explicit timings can be found in Table 6.1.

CHAPTER 6

Evaluation

The implementation of this project is evaluated on both data volume as well as in the temporal domain. This chapter presents this evaluation and its key findings.

6.1 Environment

All participated tools, the ISS with the stub, GDB and the extension were run on a Windows Subsystem for Linux (WSL) machine. The utilized GDB distribution was `gdb-multiarch` version 12.1.

GDB was set up like described in Section 5.2 with the additional configuration to turn of pagination. This is achieved via the `set pagination off` command [SPS24, pp. 368, 369]. Pagination might otherwise occur when GDB outputs a large quantity of text, requesting the user for further input in order to see the next chunk of data. To measure the time a specific action on GDB took, its Python support was utilized. For instance, in order to get the exact time, issuing a step command takes, the time before and after the command was sent could be captured with the `datetime` API in Python. Since this would also include any time the user takes to input commands into GDB, an internal function was written. An example of such a function can be seen in Listing 6.1. When executing this function, the time before and after a command is noted in order to compute the time span. Similarly, the time until the entry break was calculated. The results could then be printed using the Python integration as well.

Listing 6.1: Example of a function measuring the time the `step` command takes.

```
define fns
    Python start_time = datetime.datetime.now()
    step
    Python end_time = datetime.datetime.now()
    Python delta = end_time - start_time
    Python deltas.append(delta)
end
```

It is important to note, that this will count the wall clock time, meaning the time the machine takes to execute a specific command. This is highly subjective and can vary largely based on what computer this is executed on. Nevertheless, the gathered information can still be used to reason about relative differences in timing.

For measuring timings on the extension, the `performance` API was used. This provides high resolution timestamps to evaluate durations in milliseconds [con24]. The time stopped for the extension was the time spent *inside* of the extension. This excludes the process of sending the corresponding DAP requests to the extension and back. However, the time spend for transmitting these requests is negligible, since the extension is running inline, not on another server or thread.

For data tracking, a tool called `iptraf-ng` was used. This tool is capable of tracking Transmission Control Protocol (TCP) packets per port [ipt], which is ideal for this particular use case. In order to capture traffic to the localhost, the loopback interface[1], commonly named `lo` under Linux, was used. Notably, the data in the evaluation is captured on the TCP level. This results in additional data and does not reflect the actual amount send via the serial protocol.

All tests were conducted using the same script. Regarding the stepping, all instructions in the main section of the script where executed, resulting in a total of 21 steps. Two different methods of stepping where used for the evaluation: (i) ordinary stepping and (ii) instruction level stepping. The former continues the program, until the next source code line is reached [SPS24, p. 84], while the latter only executes the next machine instruction before halting again [SPS24, p. 86]. The stub currently does not differentiate between these two types of stepping. It will always just execute the next machine instruction. The final result, regardless of timing or data evaluation, was then computed using the average. Likewise, the values for the break on entry time and reading memory from the Stack Pointer (SP) register where conducted five times each, and summarized using the average. For the first test, the time until the debugger halts after a connection was established was timed. Importantly, no breakpoints where set prior to connecting in order not to influence the startup time. Next, to measure memory reading times, an

[1] More information about this interface can be found here: `https://man.freebsd.org/cgi/man.cgi?query=lo&sektion=4&manpath=FreeBSD+14.1-RELEASE+and+Ports` (visited on 09/03/2024).

arbitrary register with the already explained `data_ptr` type has been chosen, in this case it was the SP. When opening the memory view in VS Code, 4096 B are requested. The memory at that section is filled with zeros. This process was timed, until all of the requested data was available. Finally, the disassembly setting measures the time until the disassembly of the whole source file is accessible. Notably, GDB does not provide a disassemble command for the whole source file, rather the disassemble command has to be issued for each function. Using memory ranges does not work in this case, since the evaluated source file has inaccessible memory areas between functions. There is a difference for the used timings regarding the disassembly situation. As indicated in the tables, the first execution time was used for the evaluation without an extension, as GDB caches the result of the disassembly command, drastically reducing its execution duration on subsequent calls.

6.2 Local Execution

In this setup, both the host and the target were executed on the same machine. Communication was established through the localhost.

Under this setup, three distinct cases where inspected: (i) using both the simulator with the stub and the extension as intended, (ii) using a GDB installation, which had access to the compiled object file and (iii) using a GDB installation without reference to the object file. The results of the local timing evaluation are presented in Table 6.1.

	Step through `main`		Break on Entry	Reading SP Memory	Disassembly
	Step (ms)	Step Instruction (ms)	(ms)	(ms)	(ms)
With the Extension					
	-	53.20	4 506.49	5 562.65	298.21
Without the Extension					
Exe.	1 029.41	1 030.69	918.59	208 611.93	714.71[a]
No Exe.	-	6 294.89	863.97	209 055.27	-

[a] First execution time.

Table 6.1: Timing evaluation for the local execution.

The first overall observation is that using the extension is faster in almost all tested aspects. This is especially notable for the read memory case. The large discrepancy in speed can be explained by the fact, that GDB issued a single memory read command for each byte, while the extension requests a memory read with the largest number of bytes possible, only being limited by the packet size.

Additionally, the difference in step times can also be easily explained. The extension

sends a step packet for each step, but GDB unexpectedly does not do that. Rather it sends a few memory read packets, followed by a breakpoint command.

Furthermore, the large increase in time until the debugger breaks on entry can be attributed to the extension requesting and processing relevant data about line mappings and disassembly information.

GDB requires more time to complete the disassembly request. When using the extension, the process of disassembling the object file is done at the stub, reducing the overall time it takes to display that information.

When taking all five repetitions into account, the average timing for providing the disassembly of the whole source file would result in a time of 146.36 ms.

Origin	Step (B)	Step Instruction (B)	Break on Entry (B)	Reading SP Memory (B)
With the Extension				
Ext.	-	≈1 335[a]	16 392	9 480
Stub	-	≈1 349[a]	28 558	25 784
Without the Extension				
With the Executable				
GDB	≈9 125[a]	≈9 125[a]	10 938	1 793 504
Stub	≈7 254[a]	≈7 254[a]	15 476	1 335 400
Without the Executable				
GDB	-	54 506	10 260	1 810 432
Stub	-	≈41 983[a]	15 040	1 335 400

[a] Rounded to the nearest byte.

Table 6.2: Data evaluation for the local execution.

Regarding the evaluation of the transmitted data, as seen in Table 6.2, the timing differences when stepping comparing the use of the stub with and without the extension is also reflected by the transferred data. Similarly, the increased transmission volume for the memory read request without the executable can be contributed to the same aforementioned fact. The one case where using the extension results in an increase of traffic, is the break on entry situation. Again, this is due to the fact, that the extension requests information about source line mappings and disassembly, both of these request are not issued by GDB.

6.3 Remote Execution

The temporal dimension was also evaluated over a remote usage of this debugging interface. The setup involved the simulator to run on one machine, while another device

in the same network executed the extension. In order to reach the stub running on a WSL, the corresponding port had to be forwarded [Con24].

	Step through `main`		Break on Entry	Reading SP Memory	Disassembly
	Step (ms)	Step Instruction (ms)	(ms)	(ms)	(ms)
With the Extension					
	-	114.03	842.06	453.59	23.68
Without the Extension					
Exe.	2 305.28	2 125.19	1 835.66	439 905.38	401.20[a]
No Exe.	-	13 431.44	1 872.84	456 492.04	-

[a] First execution time.

Table 6.3: Timing evaluation for the remote execution.

Overall, the timings when used without an extension increased as expected in comparison to the local execution. The speedup utilizing the extension in the remote setting can be attributed to the increased performance of the device. This performance difference will elaborated on in the succeeding section.

The discrepancy in duration for the disassembly case might also be connected to the performance differences. Moreover, as already stated previously, the recorded time from the first execution was used for evaluation. When regarding all five tests, the average results in a time of 82.25 ms until the whole source file is disassembled.

6.4 File Transmission

The file transmission using the `qXfer` packet was also evaluated. Subject for the transfer was a file with the size of precisely 1 MB. Its content consisted entirely of the ASCII `NUL` character. The environment is identical to that of the previous setups.

Local (ms)	Remote (ms)
423 182.75	66 364.06

Table 6.4: Timing evaluation for file transmission.

Notably, the transmission is faster by a significant factor when requesting the file remotely. The large contrast between the times can be attributed to the power of the different devices in use. More concretely, the machine running the extension has 14 cores, while

the other device, running GDB in this case, and both host and target in the previous context only has 8 cores.

A more performant implementation of the extension, especially the packet reading might alleviate this problem.

CHAPTER 7

Conclusion

The presented debugging interface provides an accessible way to debug scripts run on a VADL generated simulator. It provides a seamless integration into VS Code, enabling the user to use its powerful UI. Furthermore, it is possible to set breakpoints, examine and manipulate the values in registers and it supports reading arbitrary memory of the simulator. Additionally, disassembly information is also provided to the user. The extension is fully architecture agnostic and is able to cater to an arbitrary architecture. Moreover, the debugging interface is developed in a distributed manner, which enables remote debugging.

7.1 Future Work

While the current state of the project is operational, there are still some missing features, as well as a few minor alterations that would significantly enhance the debugging interface. First and foremost, the previously stated support for displaying the contents of the instruction pipeline must be incorporated. To achieve this, it is likely that the serial protocol will require extensions beyond the use of additional file transfers via the `qXfer` packet. The specification of the pipeline can easily be described in the target description details. Moreover, supporting a representation of the instruction pipeline, would most likely also necessitate the development of a custom view in VS Code in order to present the pipeline in a meaningful manner to the user. The necessary simulator, that operates on a cycle accurate level can already be generated through the VADL system [Sch20, pp. 47–50].

In order to support architectures with variable instruction sizes, the target description details, and its associated elements in the extension, would also need an augmentation. One potential solution would be to explicitly list the instruction size for each possible mnemonic. Furthermore, the ability to specify the instruction size and minimal address-

able memory size in bits rather than of bytes would facilitate the support of more unique architectures.

There are also features which are supported by the tools in use, just not implemented yet. This includes support for manipulating memory, setting conditional breakpoints or restarting the debuggee. Additionally, there are some elements of the serial protocol that are not fully completed yet. For example, the target description does not support the `type` element, and received XML documents are not validated against their DTD. Another example would be the use of run-length encoding in order to minimize transferred data, potentially increasing overall performance. Moreover, currently the maximum packet size is fixed to an arbitrary number. There is definitely room for experimentation to find the optimal size. Notably, there is also a small deviation from the serial protocol, as both the stub and the extension do not allow an unlimited packet size prior to the `qSupported` packet, as advised by GDB [SPS24, p. 793].

Other small changes, which would greatly impact the perceived usability of the extension, is the option to set breakpoints on lines with labels, or to have the extension wait until it is able to connect to the stub. Regarding the former, it would be plausible for the extension to try to set the breakpoint on the next line, if possible, and informing the debugging tool of the changed location. For the latter, currently, the stub has to be started prior to the extension in order to achieve a successful connection.

Another great improvement focuses on the involved artefacts for the extension. Currently, the target description along with its details have to be typed manually. In the future, it would be conceivable to generate these files together with the ISS right from the VADL source files. In conjunction with the generation of the target description files, it is imaginable for the VADL system to generate additional files describing the ISA run on the simulator, enabling the possibility of a language extension for VS Code. This would include correct syntax highlighting, language snippets and language configurations like indentations, folding and autosurrounding [24d]. The extension written for this debugging interface has the potential to evolve into an extensive set of tools, making development for and on a specific architecture straightforward and accessible. In the long run, this would also void the need for the extension dependency for the language ID, required for VS Code.

All these ideas would contribute to this project with more features. However, the stub would also benefit from a more structured architecture, and the extension from a more performance centric implementation. Furthermore, there are still a few places, where the stub is not entirely architecture agnostic.

The presented aspects are all useful, when the simulator is run on the same machine as the extension. However, ultimately, the ISS would not be something the user has to worry about. Ideally, it would run on a server, executing the source on the cloud, while communicating with the extension. Implementing this whole infrastructure might also imply some changes to the extension, in particular on how sources are handled.

As can be seen by the multitude of possibilities and enhancements, this debugging interface presents significant potential.

List of Figures

List of Tables

List of Algorithms

Glossary

e.g. for example. 34

Exe. Executable. 43, 45

Ext. Extension. 44

ID identification. 5, 29, 38, 48

Acronyms

API Application Programming Interface. 18, 20, 31, 41, 42

ASCII American Standard Code for Information Interchange. 4, 5, 24, 45

CPU Central Processing Unit. 13, 14, 18, 19, 21

CR Carriage Return. 24

CRLF Carriage Return Line Feed. 24

CSV Comma-Separated Values. 24–27, 51

CU Compilation Unit. 24

DA Debug Adapter. 10, 29, 30, 33–36

DAP Debug Adapter Protocol. 3, 10, 23, 28–30, 32–36, 38, 42

DIE Debugging Information Entry. 10, 24

DTD Document Type Definition. 22, 33, 48

ELF Executable and Linkable Format. 9, 10, 23

GDB GNU Debugger. xi, xiii, 3–9, 13, 14, 17, 22, 23, 28–30, 32, 34, 36–39, 41, 43, 44, 46, 48

IANA Internet Assigned Numbers Authority. 16

IDE Integrated Development Environment. 10, 13, 14

IP Internet Protocol. 4, 38

IPv4 Internet Protocol Version 4. 15

ISA Instruction Set Architecture. 11, 25, 48

ISS Instruction Set Simulator. 11, 13–15, 17, 18, 21, 36, 37, 39, 41, 48

LF Line Feed. 24

OS Operating System. 10

OS ABI Operating System Application Binary Interface. 8

PC Program Counter. 22, 23, 35

PID Process Identification. 20

RFC Requests for Comments. 24, 27

RISC-V Reduced Instruction Set Computers V. 8

SP Stack Pointer. 36, 42–45

SPARC Scalable Processor Architecture. 4, 17

TCP Transmission Control Protocol. 42

TS TypeScript. 32, 33

UI User Interface. 28, 32, 47

VADL Vienna Architecture Description Language. 1, 3, 11, 13, 18, 23, 26, 37, 47, 48

VS Code Visual Studio Code. xi, xiii, 13–15, 28–30, 32–38, 43, 47, 48

WSL Windows Subsystem for Linux. 30, 41, 45

XML Extensible Markup Language. 8, 48

XOR Exclusive OR. 5

Bibliography

[17] *DWARF Debugging Information Format Version 5*. 10th ed. DWARF Debugging Information Format Committee, 2017. URL: https://dwarfstd.org/doc/DWARF5.pdf (visited on 07/25/2024).

[21a] *Configure C/C++ debugging*. 2021. URL: https://code.visualstudio.com/docs/cpp/launch-json-reference (visited on 08/14/2024).

[21b] *July 2021 (version 1.59)*. 2021. URL: https://code.visualstudio.com/updates/v1_59 (visited on 08/25/2024).

[22] *January 2022 (version 1.64)*. 2022. URL: https://code.visualstudio.com/updates/v1_64 (visited on 08/23/2024).

[24a] *Debug Adapter Protocol*. 2024. URL: https://microsoft.github.io/debug-adapter-protocol/specification (visited on 07/24/2024).

[24b] *Debugger Extension*. 2024. URL: https://code.visualstudio.com/api/extension-guides/debugger-extension (visited on 08/10/2024).

[24c] *Debugging*. 2024. URL: https://code.visualstudio.com/docs/editor/debugging (visited on 08/25/2024).

[24d] *Language Extensions Overview*. 2024. URL: https://code.visualstudio.com/api/language-extensions/overview (visited on 09/04/2024).

[24e] *Language Identifiers*. 2024. URL: https://code.visualstudio.com/docs/languages/identifiers (visited on 08/14/2024).

[24f] *llvm-mc - LLVM Machine Code Playground*. 2024. URL: https://llvm.org/docs/CommandGuide/llvm-mc.html (visited on 08/08/2024).

[24g] *llvm-objdump - LLVM's object file dumper*. 2024. URL: https://llvm.org/docs/CommandGuide/llvm-objdump.html (visited on 08/05/2024).

[24h] *Overview*. 2024. URL: https://microsoft.github.io/debug-adapter-protocol/overview (visited on 05/20/2024).

[24i] *Programming Languages*. 2024. URL: https://code.visualstudio.com/docs/languages/overview (visited on 08/14/2024).

[24j] *Variables Reference*. 2024. URL: https://code.visualstudio.com/docs/editor/variables-reference (visited on 08/25/2024).

[24k] *Your First Extension*. 2024. URL: https://code.visualstudio.com/api/get-started/your-first-extension (visited on 08/10/2024).

[95] *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. 1.2. TIS Committee, 1995. URL: https://refspecs.linuxbase.org/elf/elf.pdf (visited on 08/04/2024).

[Con24] Contributors. *Accessing network applications with WSL*. 2024. URL: https://learn.microsoft.com/en-us/windows/wsl/networking (visited on 09/10/2024).

[con24] MDN contributors. *Performance: now() method*. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now (visited on 09/02/2024).

[dev23] GDB developers. *GDB: The GNU Project Debugger*. 2023. URL: https://www.sourceware.org/gdb/contribute/ (visited on 05/15/2024).

[dev24] GDB developers. *GDB: The GNU Project Debugger*. 2024. URL: https://www.sourceware.org/gdb/ (visited on 05/14/2024).

[Gam95] Erich Gamma. *Design patterns : elements of reusable object-oriented software*. eng. 37th printing. Addison-Wesley professional computing series. Reading, Mass.: Addison-Wesley, 1995. ISBN: 0321700694.

[Him+24] Simon Himmelbauer et al. *The Vienna Architecture Description Language*. 2024. arXiv: 2402.09087 [cs.PL].

[ipt] iptraf-ng. *iptraf-ng*. URL: https://github.com/iptraf-ng/iptraf-ng (visited on 09/03/2024).

[jus23] justarandomgeek. *LoggingDebugSession: does not create log file when running inline in vscode #290*. 2023. URL: https://github.com/microsoft/vscode-debugadapter-node/issues/290 (visited on 08/28/2024).

[Sch20] Hermann Schützenhöfer. "Cycle-Accurate simulator generator for the VADL processor description language". PhD thesis. Technische Universität Wien, 2020.

[Sha05] Yakov Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. RFC 4180. Oct. 2005. DOI: 10.17487/RFC4180. URL: https://www.rfc-editor.org/info/rfc4180.

[SPS24] Richard Stallman, Roland Pesch, and et al. Stan Shebs. *Debugging with GDB*. 10th ed. Free Software Foundation, 2024. URL: https://sourceware.org/gdb/current/onlinedocs/gdb.pdf (visited on 05/16/2024).