Danmarks
Tekniske
Universitet

Debugger for Patmos
Project Report

Author

Mariana Santos - s233360

Supervisor

Professor Martin Schoeberl

May 6, 2024

Contents

1 Introduction **1**
1.1 Debuggers . . . 1
1.2 Patmos . . . 1

2 GDB and Remote Serial Protocol (RSP) **2**
2.1 Remote debugging with GDB RSP . . . 3
2.1.1 Communication . . . 3
2.1.2 Setting up of the debugging session . . . 3
2.1.3 GDB debugging commands . . . 4

3 Implementation **5**
3.1 Preparing GDB for a new architecture . . . 5
3.2 Starting a debugging session . . . 6
3.3 Setting up message exchange . . . 7
3.3.1 The `qSupported` packet (1) . . . 9
3.3.2 The `vMustReplyEmpty` packet (2) . . . 9
3.3.3 The `H` packets (3) (9) (12) . . . 9
3.3.4 The `qTStatus` packet (4) . . . 10
3.3.5 The `?` packet (5) . . . 10
3.3.6 The `qfThreadInfo` and `qsThreadInfo` packets (6) (7) (15) (16) . . . 10
3.3.7 The `qAttached` packet (8) . . . 11
3.3.8 The `qC` packet (10) . . . 11
3.3.9 The `qOffsets` packet (11) . . . 11
3.3.10 The `g` packet (13) . . . 11
3.3.11 The `m` packets (14) . . . 12
3.3.12 The `qSymbol` packet (17) . . . 12
3.4 Supporting GDB commands . . . 13
3.5 The `Z0` packet (19) . . . 13
3.5.1 The `vCont?` packet (20) . . . 16
3.5.2 The `c` packet (22) . . . 16
3.5.3 The `z0` packet (24) . . . 17
3.5.4 Additional notes on the `next` command . . . 17
3.5.5 The `p` packet . . . 17
3.5.6 The `D` packet . . . 17

4 Execution and results **18**
4.1 Program to debug . . . 18
4.2 Execution setup . . . 18
4.2.1 Target's side setup . . . 18
4.2.2 Host's side setup . . . 19
4.2.3 Initial setup message exchange . . . 19
4.3 The `next` command . . . 19

4.4 The `break *address` command . . . 20
4.5 The `continue` command . . . 20
4.6 Another `break` command . . . 20
4.7 The `step` command . . . 20
4.8 The `info registers` and `info locals` commands . . . 21
4.9 The `quit` command . . . 21
4.10 Overall results discussion . . . 21

5 Conclusion and next steps 22

List of Figures I

List of Tables I

References II

A Appendix I: Screen captures of results III

1 Introduction

Nowadays, in a world where technology is becoming more and more ubiquitous, the use of tools that aid development are essential and taken for granted. One such tool is called a **debugger**.

On a similar note, safety-critical systems – systems upon which our life can depend – have also become omnipresent. To ensure that no accidents or unexpected events happen, these systems need to be carefully analysed to make sure that their response times is within acceptable and sensible bounds. **Patmos** is a new processor designed with the thought of facilitating the determination of the worst-case execution time (WCET). [1]

Currently Patmos can execute C and C++ code through a compiler adapted from LLVM. However, no debugging solution was created yet for this new processor. The goal of this project is to implement a debugging solution that allows developers to inspect programs executing on the Patmos processor.

The remaining of this section will provide a brief introduction to both debuggers and Patmos. The details on the debugger program used, as well as the protocol will be explored on section 2, while section 3 will focus on the implementation details. The section 4 will delve on the process to debug a program in Patmos and on the results analysis. To finish, section 5 will conclude this report by presenting a final overview and a discussion on the next steps.

1.1 Debuggers

Debuggers are a powerful analysis and inspection tool that allow the developer to obtain important and detailed insights on the program execution. Among other operations, these tools enable the developer to inspect and even edit variable values at run-time, by requesting the program flow to be interrupted. The flow is interrupted with what is known as a **breakpoint**. With the use of a debugger, the user can set breakpoints on specific parts of the code. When the program reaches one of these parts, the execution will be interrupted, and the user can get insight of the program state at that point.

Debuggers are able to provide such fine-grained information because they work closely with the processor. This effectively means that the tool needs to have a very detailed description of the machine where the program is being run. This description needs to include details such as register organisation and their sizes, besides its Instruction Set Architecture (ISA).

One of the most portable and widely used debuggers is called GDB (GNU Project Debugger), and can be used, without any additional configuration, in a number of different systems. [2]

1.2 Patmos

As a small introduction to Patmos, I will describe a few of its characteristics that show special relevance for the project. This information was obtained from the Patmos Reference Handbook [3].

Patmos is a processor that can be implemented on an FPGA board. For this project, it is implemented on the Altera DE2-115. The compiled programs are imported via serial port, which is also used for other types of communication with the development machine.

It contains 32 32-bit general-purpose registers, 8 single-bit predicate registers, and 16 13-bit special-purpose registers. In the special-purpose registers, 2 of them in particular should be highlighted: `$sxb` – exception return base (also represented as `$s9`) – and `$sxo` – exception return offset (also represented as `$s10`). When an exception or trap occurs, the address of the *next* instruction will be set on these two special registers. To obtain the address where the exception/trap occurred, `$sxb` and `$sxo` need to be summed, and the value `4` should be subtracted from it, to obtain the actual instruction instead of the next one.

For this project, it is also relevant to mention the Patmos' **instruction cache**, configured as a **method cache**. The method cache stores contiguous sequences of code which often correspond to entire functions. If a change has been made in main memory to data that is also present in the cache, it is important that the cache is **invalidated**. This is done by writing a value to the `cachectrl` register. If the cache is set as invalid, when the program executes a return instruction (both `ret` or `xret`) the cache will be refilled with updated values.

2 GDB and Remote Serial Protocol (RSP)

GDB is a mature debugger problem with years of development and many functionalities. One interesting feature is that it allows for a program running on a different machine than GDB to be debugged, called **remote debugging**. We will call the machine where the program to be debugged is executing the **target machine**, and the machine where GDB is executing the **host machine**. This is especially useful in the scope of this project, where the target machine is the Patmos processor, and the host is the development machine. The protocol that allows GDB to connect to remote targets called **GDB Remote Serial Protocol (RSP)**. [4]

For this protocol to be successfully implemented, there are two things that are required:

- GDB needs to have a good description of the target's architecture. By default, there are a few common architectures that are configured in GDB. For a recent processor such as Patmos, the debugger needs to be provided with detailed architecture information.
- The target needs to implement a number of functions and behaviour that will allow the communication to happen between the host and the target itself. This includes not only defining how to receive and send data packets, but also how to respond to GDB's requests. Normally, these functions are implemented on a file known as a *stub file*. GDB source code includes a few examples of stub files for some machines on the directory `gdb/stubs/`, while GDB's side of the communication is implemented on the source file `remote.c`.

2.1 Remote debugging with GDB Remote Serial Protocol (RSP)

As soon as a GDB remote debugging session is started, the host will try to communicate with the target machine. To allow for the communication to happen, a few things need to happen on the target side:

- The program needs to be already executing.
- A few stub functions need to execute before the execution of the main function of the program to be debugged. The goal is to do an initial setup for the debugging session to successfully happen. [4, p. 330] This setup can be divided into 2 parts:
  1. define what happens once an exception or trap occurs. In this situation, the "control" should be given to GDB. This effectively means that the program should be waiting and responding to queries from the host machine whenever the program flow is interrupted.
  2. simulate a stop in the program through a trap. This enables the control to be passed over to the host machine before any part of the actual program is executed. GDB will be able to establish connection for the first time, and request setup information.

2.1.1 Communication

The messages exchanged between GDB and the target have the same format:

`$<message>#<checksum>`

in which:

- `<message>` corresponds to a string with ASCII characters.
- `<checksum>` is a hexadecimal value obtained by summing all the character values in `<message>` and computing the modulo of the obtained value with 256.

Both the host and the target counterparts check if the checksum value of the received messages is correct, which allows them to verify if the message was well-transmitted. If this is the case, the receiver will send a single `+`. Otherwise, a `-` will be sent, signalling that the message should be re-transmitted.

2.1.2 Setting up of the debugging session

Normally, GDB will obtain much of the a program's information from the program's compiled ELF file, which is normally compiled specifically to be read by a debugger. These binary files are complemented with debug symbols and other debug information not needed for the normal simple execution of the program. This information includes, for example, details on the program lines and variables names. However, the C/C++ compiler for Patmos, as it is right now, can't produce a binary file with debug data, and therefore GDB won't be able to access this detailed information. This results in a number of commands that GDB will not be able to execute. Nevertheless, the ELF file is useful for information on functions and instruction addresses.

As mentioned, once both the host and target counterparts have been set up, they will start exchanging packets. Figure 1 is a sequence diagram that depicts this initial exchange of information. This exchange documented in the diagram is the result of my own implementation, since I couldn't find reliable documentation that described in depth this workflow.

Most of the exchange focuses on understanding which features are supported by the host and target implementations, as well as exchanging information about threads and registers. Each query and response will be explored in greater detail in section 3.

After the depicted exchange, both the GDB host and the target program will wait for input from the user. The debugger is now set up and the user can select what they want to do next. For example, they may chose to execute the next line of the program, or to insert a breakpoint in a specific line or instruction.

2.1.3 GDB debugging commands

On a typical debugging session, there are a handful of commands that are most likely to be used a number of times. These include:

- `break` – sets up a breakpoint at the current line.
- `break <line number>` – puts a breakpoint at line `<line number>`
- `break *<instruction address>` – puts a breakpoint at the instruction at `<instruction address>`
- `delete <breakpoint number>` – deletes breakpoint with number `<breakpoint number>`
- `info break` – shows information on the existing breakpoints
- `continue` – continues normal program execution until a breakpoint or error occurs
- `step` – runs the next line of the program and will step into a function
- `next` – runs the next line of the program without entering functions
- `where` – shows the current line, address and function name of the program
- `info locals` – shows values on the local variables in the current level
- `info registers` – shows the values in the registers defined in the architecture description
- `quit` – detaches the debugging session from the target device and terminates it

Not all of these commands will trigger immediately an exchange of messages with the target device. For example, there is no need for communication to show the user the existing breakpoints. On the other hand, commands such as `step` or `next` will involve exchange of packets. Figure 3 is another sequence diagram that depicts the communication involved on the execution of a `next` command.

Additionally, as was already discussed, some of these commands would never be able to be executed on the scope of this project since the lack of debugging information on the binary file provided to GDB. From the list provided, these commands include the `break <line number>` and `info locals`.

3 Implementation

Having already presented the theory of how RSP works, and what is necessary to have a functioning debugging session, we can analyse how this can be implemented. This section will focus on the steps that were taken to implement this protocol in Patmos.

The implementation took as a starting point a GitHub repository[1], from which some functions were used as inspiration. Due to the differences between the target devices, only a small part of the code in the repository was useful as an example. My own implementation described in this chapter can be found on my GitHub repository[2].

3.1 Preparing GDB for a new architecture

So that GDB can correctly interpret the data that it is given, such as instructions and registers' structure, it needs to be properly configured with details of the target device's architecture. Without information of the target's Instruction Set Architecture (ISA), GDB cannot correctly identify the meaning of any instruction to be executed, for example. Setting up GDB with an architecture that it doesn't already support includes developing a number of XML and C files that provide a good description of it. Furthermore, it involves compiling GDB.

Unfortunately the documentation doesn't describe this procedure very well and as such I only figured out too late that I need to do this. This issue was only discovered upon trying to find an answer as to why GDB wasn't able to understand the register values that the target informed, which will be described very shortly.

As an alternative to doing a proper setup of the architecture, which wouldn't be feasible for this project's timeline, I analysed a few configuration files from architectures supported by GDB. These can be found on GDB's source code, in `gdb/features/`. Of particular interest are the XML files that describe the registers for each architecture. An inspection of the files showed that the MIPS XML configuration described a similar structure to the Patmos registers: it also has 32-bit 32 general-purpose registers. Table 1 depicts the registers as described in the configuration files. Because of the similarities in this aspect, I set up GDB so that it considers that the target device is a MIPS architecture.

[1]`https://github.com/impedimentToProgress/thumbulator/tree/master`
[2]`https://github.com/immarianaas/patmos-debugger`

0 r0	1 r1	2 r2	3 r3	4 r4	5 r5	6 r6	7 r7
8 r8	9 r9	10 r10	11 r11	12 r12	13 r13	14 r14	15 r15
16 r16	17 r17	18 r18	19 r19	20 r20	21 r21	22 r22	23 r23
24 r24	25 r25	26 r26	27 r27	28 r28	29 r29	30 r30	31 r31
32 sr	33 lo	34 hi	35 bad	36 cause	37 pc		

Table 1: Table representing registers as described in the XML file

3.2 Starting a debugging session

As discussed in subsection 2.1, when the GDB is started in the host machine, it will try to communicate with the target device. The communication between the two is realised by a serial connection over UART. On the target side, the implementation to handle communication over UART is as simple as reading or writing from the UART register.

Furthermore, the two-part setup mentioned in subsection 2.1 needs to be implemented. As a good first step, two functions that realise this behaviour were manually placed in the first lines of the main function of the program to be debugged. The main function of the program looks like Listing 1 In fact, all of the stub code developed was placed directly on the C file of the program that is going to be debugged.

```c
int main()
{
    set_intr_handlers();
    breakpoint();
    // rest of function
}
```

Listing 1: Main function.

These two functions do all the setup necessary for a debugging session on the target side:

- `set_intr_handlers()` registers a function as the exception handler for all the 32 exceptions – ensuring that the handler function will execute if any kind of interrupt occurs. This exception handler function, besides saving the state of the program in the stack cache, will also communicate with the host machine and effectively let the GDB host control the following steps.

- `breakpoint()` will invoke a trap, which will result in the execution of the function handler mentioned above.

The part of the exception handler that implements the communication with the host machine is described in Listing 2, although a bit simplified. It starts by sending a packet that specifies the reason why the program was interrupted. In this situation, the value sent is `S05`, which indicates the interruption happened because of a trap – further explanation can be found in subsubsection 3.3.5. Then there is an infinite loop that will receive messages and handle them according to their content. The only way for the program to continue the normal execution is if GDB sends a packet for the program to continue or to detach, otherwise the program won't be further executed – GDB has control of the target until it explicitly lets the program continue execution.

```c
// part of interrupt handler
void handle_communication()
{
    send_str_packet("S05");

    while (1)
    {
        struct rsp_buf *buf = get_packet();

        // did we receive a 'qSupported' message?
        if (starts_with_substr(buf->data, "qSupported"))
        {
            // handle reply for this message
        }
        else if (starts_with_substr(buf->data, "H"))
        {
            // handle reply for this message
        }
        else if (starts_with_substr(buf->data, "c"))
        {
            // return control to target device
            return;
        }
        // etc
    }
}
```

Listing 2: Pseudo-code of the function that handles communication with the host.

3.3 Setting up message exchange

Before the actual debugging of a program begins, there is an exchange of messages between the host and the target device, as described in the diagram in Figure 1. This subsection will delve into the queries and responses involved in this process. The numbers on the titles of the following segments correspond to the numbers on the diagram annotations.

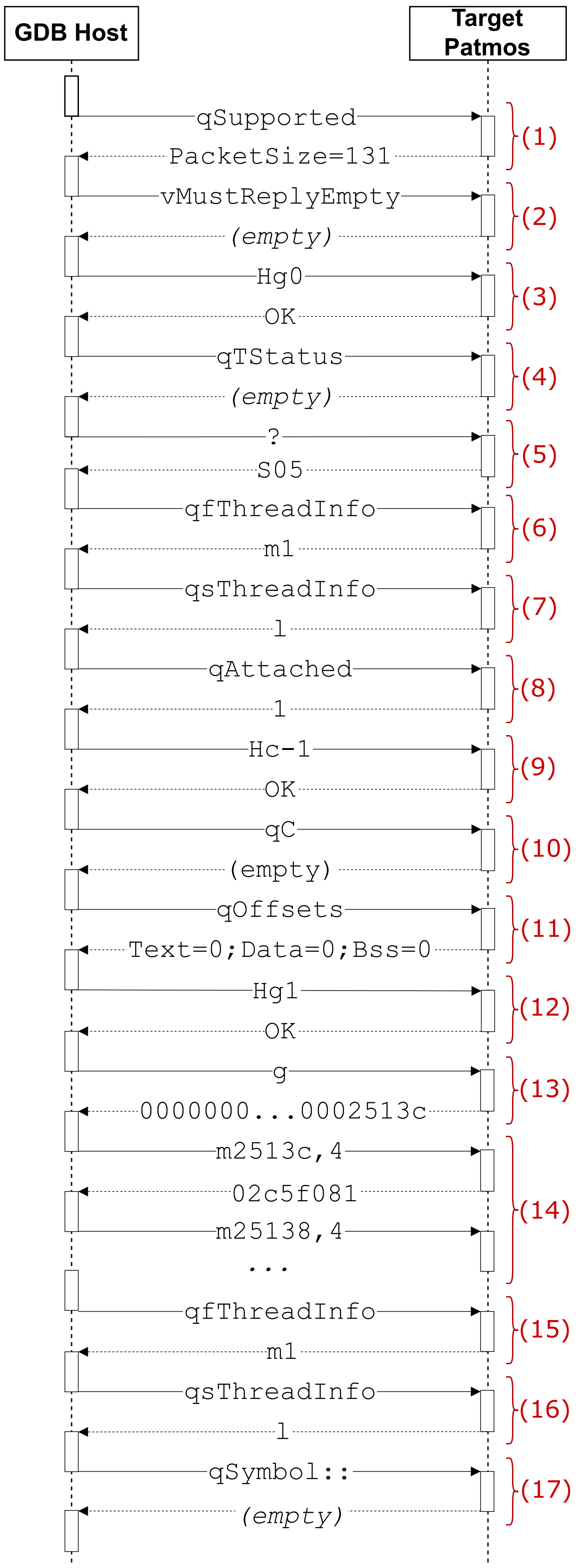


Figure 1: Sequence diagram modelling the initial setup between the target and the host.

After this process is completed, GDB will wait for the user's input for commands, and therefore the target will also be waiting. The following subsection will delve into this next step that happens after the initial setup described.

3.3.1 The `qSupported` packet (1)

The first packet sent by GDB contains information about the features it supports, and is expecting the target to reply with the features it supports. [4, p. 791] The full message received is as follows:
`qSupported:multiprocess+;swbreak+;hwbreak+;qRelocInsn+;fork-events+;`
`vfork-events+;exec-events+;vContSupported+;QThreadEvents+;`
`no-resumed+;xmlRegisters=i386`

The words with a trailing `+` correspond to supported features, while the ones with a trailing `-` correspond to unsupported ones. A full-fledged target debugger implementation could analyse this packet to know what to expect from the host side. However, since we are aiming for a minimum viable product, this information can be safely ignored.

Even though it is not a requirement, it seems to be recommended that the target responds to this packet with information on the maximum length a packet can be – value which should not consider the leading `$` not the trailing `#` and checksum.

For this implementation, the target replies with `PacketSize=131`. The value corresponds to the hexadecimal value of $38 * 8 + 1 = 305$, which is the number of characters needed to send all the register values needed for the `g` packet (further discussed in subsubsection 3.3.10), and an additional trailing `NULL` character. In fact, if this value was not provided to the host at this point, it would eventually infer it from the the answer to the `g` packet.

3.3.2 The `vMustReplyEmpty` packet (2)

This packet is used for GDB to assess if the target correctly supports unknown `v` packets. The expected response from the target is an empty packet. This is used as an indication for the host about potential problems that the target might have when handling other `v` packets. [4, p. 774]

3.3.3 The `H` packets (3) (9) (12)

Both `Hg` and `Hc` are part of what can be called a H-packet. They are used for the host to select a thread to which the following packets are to be applied. [4, p. 770] Such a packet follows the following format:

`H<operation type><thread number>`

in which:

- `<operation type>` can be `c` – for operations such as *step* and *continue* –, and `g` – for all other operations.

- `<thread number>` can be either the thread ID, the value `0` – meaning any thread available –, or `-1` – to indicate that this operation should apply to every thread. [4, p. 767]

Because we are only handling one thread, the target will always acknowledge H-packets with an `OK` response, and no further processing is necessary.

3.3.4 The `qTStatus` packet (4)

The goal of this packet is for GDB to understand if there is a trace experiment executing at the moment. [4, p. 810] Tracepoints are useful in applications with real-time dependencies, where time might alter the behaviour of the program. In these situations, it isn't feasible to stop the program with a normal breakpoint, since this might break the program due to the timing constraints. Tracepoints allow the developer to analyse the system without having to interrupt it. [4, p. 201]

Patmos will return this packet with an empty message, since tracepoints are not implemented, as it's a non-trivial feature.

3.3.5 The `?` packet (5)

The goal of this packet is for the host to know why the target halted. [4, p. 767] The response has the format `S<signal value>` where `<signal value>` is a 2 hexadecimal digit value corresponding to the reason why the target stopped. The values and their meaning are defined in the GDB source code in `/include/gdb/signals.def`. [4, p. 777]

In this initial setup situation, the program paused because of a trap, and therefore the reply packet is `S05`, which can also be used to indicate a breakpoint. In my implementation, every time this packet is received, the target will always reply with this value.

3.3.6 The `qfThreadInfo` and `qsThreadInfo` packets (6) (7) (15) (16)

These packets are for the host to assess the thread IDs of the threads executing. [4, p. 784] Since there might be multiple threads, this process was separated into 2 different exchanges:

1. a `qfThreadInfo` packet will initiate by asking for one of the threads. The expected reply is formatted as `m<thread number>`, where `<thread number>` corresponds to the thread ID. In this implementation, the target will always reply with `m1`.

2. at least one `qsThreadInfo` will be sent, until the target replies with a message containing only the character `l`. If there is more than one thread, the target can reply to each one of these packets in a similar way as described above. In our situation, the target will always reply to this packet with `l`, since there is only one thread.

3.3.7 The `qAttached` packet (8)

The host sends this packet with the intention of knowing if the process executing on the target is attached to the debugging session or not. The goal is to understand if the remote process should be killed once the debugging session is terminated. A response of `1` indicates that the process is not attached to the debugging session, and thus doesn't need to be killed. [4, p. 805] This is the response from the target in this implementation.

3.3.8 The `qC` packet (10)

GDB is looking for information on the current thread ID. It will interpret an empty response as the thread not having changed, which is what happens in this case. [4, p. 781]

3.3.9 The `qOffsets` packet (11)

GDB wants to know ELF section offsets used by the target. [4, p. 787] I didn't find the documentation to be very clear on the topic, so the reply to this message was a trial and error procedure. The reply has to be the following format:

```
Text=<text offset>;Data=<data offset>;Bss=<bss offset>
```

The target will reply with `Text=0;Data=0;Bss=0`.

To reach this solution, a few other options were tested. The first tries involved analysing the ELF file of the program with the **`readelf`** command. A part of the output is depicted on Figure 2, and it shows information on the sections offsets. I tried sending a reply with the offset information displayed, but GDB was not able to correctly identify the function to which the program counter referred.

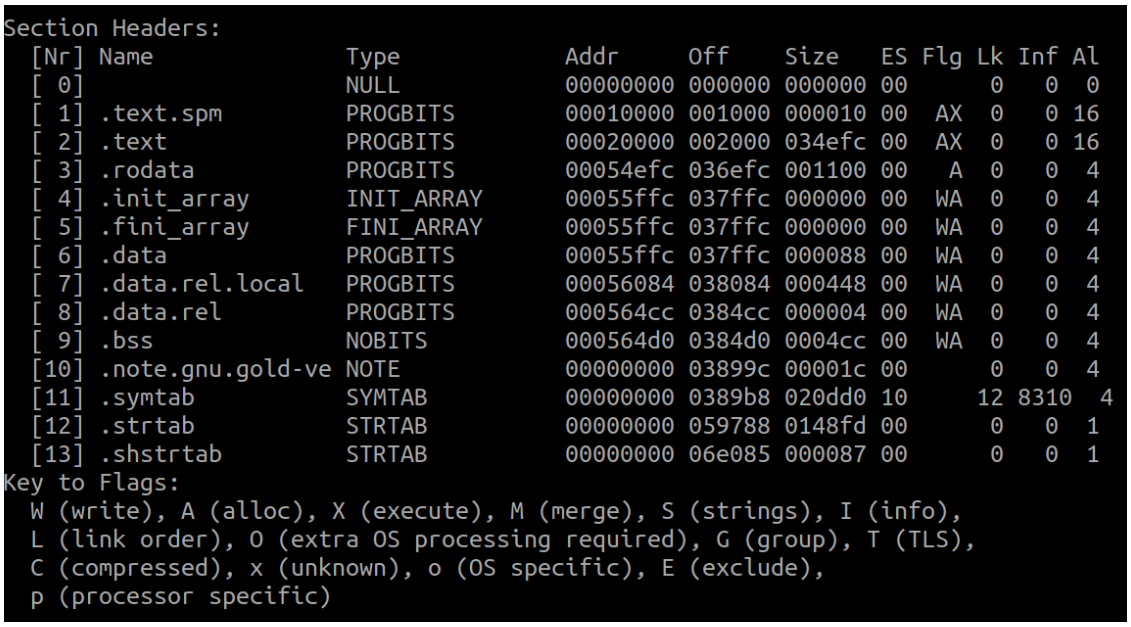
Furthermore, even though the documentation says that the `Bss` value is optional, I realised GDB would not accept it. Similarly, if the `Data` `Bss` have different values, GDB would display an error, even though the documentation describes a different behaviour.

Setting all offset values on the message with `0` seems to make GDB work as expected, so that is the chosen solution.

3.3.10 The g packet (13)

This is an important packet that GDB uses to obtain information of the values that were on the target's registers when the program halted. The response should be a continuous string of values in the order in which the registers are described in the configuration files. In this case, the order is depicted on Table 1. Each value is represented as 8 hexadecimal digits. If a register cannot be read, or if the target doesn't want to inform it, the 8 characters where the value would be specified can be filled with the character `x`. GDB will interpret this value as the register being unavailable. [4, p. 769]

When an interrupt occurs, the data that was present in the registers will be saved on Patmos' stack cache. When a `g` packet is received, the values that GDB is looking for are those that are saved in the stack cache. Therefore, the implementation includes fetching the registers values that are saved on the stack cache.



```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text.spm         PROGBITS        00010000 001000 000010 00  AX  0   0 16
  [ 2] .text             PROGBITS        00020000 002000 034efc 00  AX  0   0 16
  [ 3] .rodata           PROGBITS        00054efc 036efc 001100 00   A  0   0  4
  [ 4] .init_array       INIT_ARRAY      00055ffc 037ffc 000000 00  WA  0   0  4
  [ 5] .fini_array       FINI_ARRAY      00055ffc 037ffc 000000 00  WA  0   0  4
  [ 6] .data             PROGBITS        00055ffc 037ffc 000088 00  WA  0   0  4
  [ 7] .data.rel.local   PROGBITS        00056084 038084 000448 00  WA  0   0  4
  [ 8] .data.rel         PROGBITS        000564cc 0384cc 000004 00  WA  0   0  4
  [ 9] .bss              NOBITS          000564d0 0384d0 0004cc 00  WA  0   0  4
  [10] .note.gnu.gold-ve NOTE            00000000 03899c 00001c 00      0   0  4
  [11] .symtab           SYMTAB          00000000 0389b8 020dd0 10     12 8310  4
  [12] .strtab           STRTAB          00000000 059788 0148fd 00      0   0  1
  [13] .shstrtab         STRTAB          00000000 06e085 000087 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

Figure 2: Part of the output of executing `readelf <ELF file path> -e`, where `<ELF file path>` is the path to the compiled ELF file.

For this situation, we are sending the value of the registers `sr`, `lo`, `hi`, `bad` and `cause` as unavailable, since it shouldn't make a difference for a first version. The program counter (`pc`) should be the address of the exception or trap which was executed. This value can be found with the use of the special registers `$sxb` and `$sxo`. The program counter value that we need is obtained by summing the values on both of these registers and subtracting 4, to refer to the actual trap or exception instruction, as discussed briefly in the introduction.

An example of an actual reply sent by Patmos can be found in Figure 5, on the step annotated with (13).

3.3.11 The `m` packets (14)

`m` packets are how GDB requests to read from a memory address. The message has the format `m<address>,<num of bytes to read>`. [4, p. 770] I verified that the first `m` packet that comes after a `g` packet will always ask for the data in the address of the program counter provided, and normally the following ones will be a one-word increment from it, to read the following instructions. The data is returned as a simple hexadecimal string with the values read, which is the same representation as in the `g` packet.

GDB probably cannot understand these commands because the instructions are different from the MIPS architecture, but it does not display an explicit error.

3.3.12 The `qSymbol` packet (17)

This packet is used to let the target know that it can send symbol lookup requests to the host. [4, p. 800] The target will reply with an empty string since this is not supported.

3.4 Supporting GDB commands

After the setup described is finished, GDB will allow the user to input a command, which may or may not trigger immediate communication with the target device. A good example is the case of the `next` command – which is very similar to the `step` command –, the sequence diagram of which is on Figure 3.

This subsection will not only explore this example, but also discuss another 2 additional packet types that appeared during the project, that were also implemented. The packets described throughout this section should be enough to support a large amount of GDB commands.

The diagram shows some packets that were already discussed in the previous section: the multiple `m` packets – marked with (18) –, the `g` packet – marked with (23) –, and the `H` packet – marked with (21).

The first thing GDB does to tackle the `next` command is sending the multiple `m` packets on values close to the program counter, possibly to understand what kind of instructions they encode, and to decide on an appropriate address to place a trap instruction.

3.5 The `Z0` packet (19)

After inspecting the addresses near the program counter, GDB sends a `Z0` packet for the first time. The message has the following format: `Z0,<instruction address>,<kind>` in which:

1. `<instruction address>` is the address to an instruction in hexadecimal.
2. `<kind>` represents the size of the breakpoint that is to be inserted in bytes. During the tests, the type was always 4.

With this packet, the host is requesting the target device to replace the instruction at the provided address with a trap instruction. [4, p. 775]

Listing 3 shows a pseudo-code implementation for how the target handles this packet, divided into 4 steps.

```c
#define TRAP16 0x05800010

void handle_Z0(char *msg)
{
    // 1) extract the address from the message and make it a pointer
    int addrInt = get_address(msg);
    volatile _UNCACHED int *addr = (volatile _UNCACHED int *)addrInt;

    // 2) if the instruction is 64-bits, then
    // find the nearest following 32-bit instruction
    while (1)
    {
        if ((*addr & 0x80000000) != 0)
```

```
            addr += 2;
        else if ((*(addr - 1) & 0x80000000) != 0)
            addr += 1;
        else
            break;
    }

    // 3) save the original instruction and replace it
    save_current_instruction(addrInt, *addr);
    *addr = TRAP16;

    // 4) invalidate instruction cache
    inval_mcache();
}
```

Listing 3: Function to handle Z0 packets.

1. the address that is contained in the message is extracted and converted into a pointer using the `_UNCACHED` macro, which allows us to access memory with cache bypass. Since the target will need to edit this value, it is important that the value is properly changed in the main memory. Later on this function, the cache will need to be invalidated, so that the change is properly loaded, and eventually executed.

2. because GDB cannot correctly interpret the instruction meaning, I noticed it often sent `Z0` packets with addresses that pointed to 64-bit instructions. Because we do not want to change half of the 64-bit instruction, I defined that the target should look for the next instruction that is 32-bit, and replace that one, even though it is not the one GDB requested.

   Patmos supports instruction bundles of both 32-bit and 64-bit. 64-bit instruction bundles can only be of the format `AluLongImm`, which is depicted on Figure 4. These can be identified by the most-significant bit. If it is `1`, then the bundle is 64-bit.

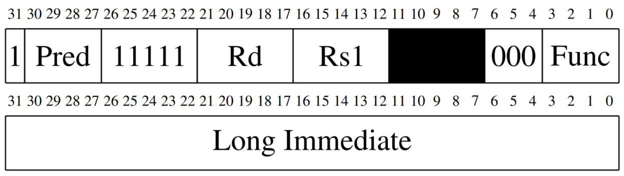


Figure 4: `AluLongImm` instruction format. Image taken from the Patmos Reference Handbook [3, p. 18].

Given an address, if the most significant bit of the instruction is `1`, it can be one of the following situations:

- the address points to the first 32-bits of a 64-bit instruction bundle.

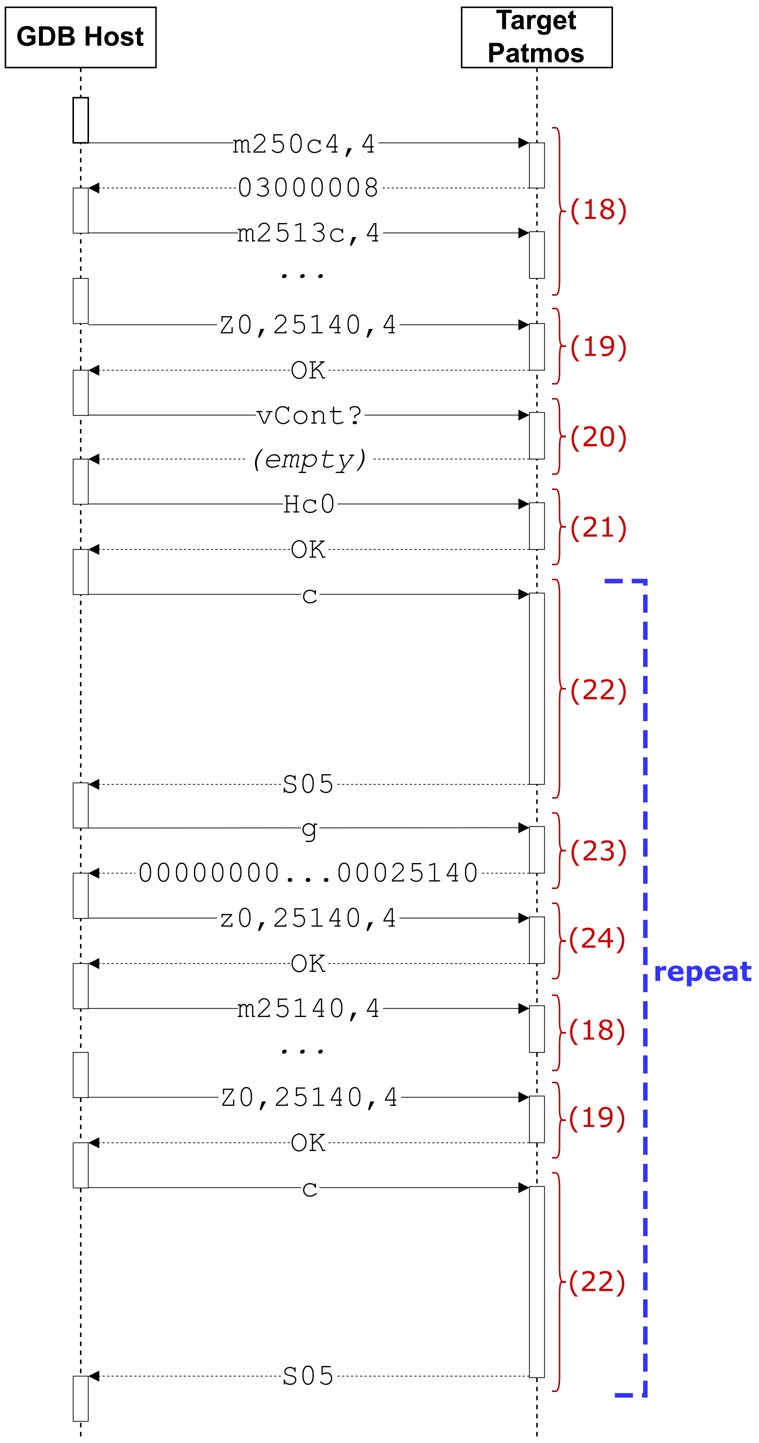


Figure 3: Sequence diagram for the next

- the address points to the second 32-bits of a 64-bit instruction bundle, in which the `Long Immediate` value is negative.

If, on the other hand, the value is , it can be one of the following situations:

- the address points to a 32-bit instruction bundle.
- the address points to the second 32-bits of a 64-bit instruction bundle, in which the `Long Immediate` value is positive.

It is therefore impossible to easily assess if the instruction belongs to a 32-bit or 64-bit bundle.

To simplify the implementation, we are assuming that the address points to a 64-bit instruction bundle if:

- the most significant bit of the value is `1`.
- the most significant bit of the value of the previous address is `1`.

3. after a valid address was found, its current data needs to be saved by the target, so that it can restore it in a future point in time. Then, this data is replaced by the binary code for a `trap 16` instruction. [3, p. 33]

4. the last step is to invalidate the instruction cache. As mentioned, the instruction cache in Patmos – which is a method cache – contains the code for full functions. We must ensure that the code for the function where the breakpoint was set is updated in the cache, so that it is actually executed. Therefore, we must invalidate it by what is essentially setting a flag. Eventually the cache should be correctly refilled with the correct data from main memory. After a `ret` or `xret` instruction, we should be sure that the cache has the correct values. [3, p. 32]

To finish, the target replies with an `OK` packet as acknowledgement.

3.5.1 The `vCont?` packet (20)

GDB is asking for the actions supported by the `vCont` packet. We are returning with empty because this is not supported, since it is not important for a minimum viable product. `vCont` is the non-deprecated version of a few packets to deal with multi-threading, so it is not applicable here. [4, p. 773]

3.5.2 The `c` packet (22)

This packet instructs the target to continue execution from where it left off. From then on, the program will be executed normally, without message exchange, until the trap set previously with the `Z0` is reached, or until another interrupt happens. [4, p. 768]

Even though it is not part of the packet description, it is important that when the program that is normally running reaches a trap sends message to GDB. This message should be the reason why the program halted, which is set to always be `S05`. This will indicate to the host that the target is ready for communication.

3.5.3 The z0 packet (24)

After the program has reached the trap set with the Z0 packet, the original instruction needs to be replaced back to where it was before. This packet instructs the target to do this. The message format is the same as for the Z0. [4, p. 775] The procedure is very similar:

1. the address is extracted and converted into a pointer.
2. if the address points to a 64-bit instruction, then find the following 32-bit instruction, following the rules explained.
3. get the previous instruction data that was saved on an array in the target, and replace the current value to which the address points with it.
4. invalidate the instruction cache.

3.5.4 Additional notes on the next command

The diagram has an annotation on a few of the steps indicating that the procedure will repeat a few times. The diagram doesn't show the complete procedure for this command, but it will be further explored in section 3. This section focused more on the different packets and their implementation.

3.5.5 The p packet

I have only observed the p packet being sent when the user runs `info registers`. This command will not only send the g packet – already discussed in subsubsection 3.3.10 –, but also two p packets. [4, p. 771]

This packet follows the format `p<register number>`, where `<register number>` is a register identifier found on the XML configuration files. It instructs the target to inform the value of the register indicated. In this situation, it seems to be requesting the values of `fsr` and `fir`, even though these don't seem to be part of the expected g packet.

Since this is not necessary for a minimum viable product, the target will always return with `xxxxxxxx`, which GDB will interpret as the registers being unavailable.

3.5.6 The D packet

This packet is used when the user wants to terminate the debugging session, through the `quit` command, for example. [4, p. 769] In this implementation, I defined that the program would continue to run normally after receiving this packet. In a real-world implementation we would need to make sure that no breakpoints are in set in the code. However, for a first testing version, this is not necessary, since we won't use the code for anything else other than debugging.

4 Execution and results

A few GDB commands were selected to show the implementation in action. This chapter will go over the steps done to reproduce this test, and will also include a discussion of the results. The sequence of commands executed is the same as they are presented in this document, however, after the `next` command in subsection 4.3, both the target and the host had to be initiated again. The images referenced in this section were placed on Appendix A, and a few of them contain annotations such as the ones being used throughout the report. Some screen captures were divided into multiple images to facilitate their handling.

4.1 Program to debug

The main function of the program that is going to be used in this section is described in Listing 4. The two first functions were already discussed, while the rest corresponds to a simple program which we want to debug.

```c

int main()
{
    // functions added for debugging
    set_debug_traps();
    breakpoint();

    // original program starts here
    int a = 2;
    int c = 5;
    int result = (a + c) * a;
    char string[50];
    sprintf(string, "Hello world! result=%d", result);
    send_string_uart(string);
    return 0;
}
```

Listing 4: Complete main function of the program.

4.2 Execution setup

This section will delve on the steps needed to successfully setup a debugging session.

4.2.1 Target's side setup

The C file needs to be compiled and downloaded into Patmos in the FPGA. The following command was used:

```
make BOOTAPP=bootable-bootloader APP=<program name> comp config download
```

with `<program name>` being the C file name without the extension.

The make target `download` will keep a serial connection open after downloading the ELF file to Patmos, which has to be stopped since we also want to connect through UART to GDB. Therefore, once the program is loaded, we can kill the `download` program to release the port.

4.2.2 Host's side setup

Because we need to change the architecture to which GDB is compiled in the development machine, we cannot use the `gdb` command. Instead, we will use `gdb-multiarch`, which is virtually GDB configured to support multiple architectures. This command is used just the same way as GDB. This program is initiated with the following command:

```
gdb-multiarch --baud 115200 <ELF file path>
```

with `<ELF file path>` being the path to the ELF file which resulted from the compilation of the program. The `--baud` argument specifies the baud rate for the communication with Patmos.

Before making GDB connect to the target, there are some GDB commands that can be useful to execute:

- `set debug remote 1` – displays information on the messages transmitted between the target and the host. This setting was kept as on for the course of these tests.
- `set debug serial 1` – shows exactly the characters that were transmitted through the serial connection. Is very useful for debugging the debugger, but for these tests, I kept this option off, since it clutters the output.
- `set architecture mips` – makes GDB interpret the target as being a MIPS architecture. Can only be performed using `gdb-multiarch`.
- `target remote <serial port path>` – connects the host to the target. `<serial port path>` should correspond to the port where the target is connected. After executing this command, both counterparts will start to exchange messages.

4.2.3 Initial setup message exchange

Figure 5 and 6 show the output shown on the screen of the GDB program when doing the initial setup as, described in subsection 3.3. Each step on the images are annotated with the same numbers that were used throughout this report.

The most important information GDB will obtain in this exchange is the program counter (`pc`), and also the instructions in the following instruction addresses. With the `pc`, GDB can correctly identify that the control flow is on the main function of the program.

We can verify that the execution happens as expected.

4.3 The `next` command

Figure 7, 8 and 9 show the part of the output obtained from executing a `next` command. In specific, Figure 7 and 8 correspond to the first packets exchange for this command, while

Figure 9 depicts the last messages exchanged. These images are also annotated with the numbers used in the diagram in Figure 3.

Before executing this command, no breakpoint was set. GDB will send multiple sets of the steps (23), (24), (18), (19), followed by a `c` packet. Each iteration includes defining the program counter and other register values, replacing the trap with the original instruction, inspecting the following instructions, setting up a new breakpoint and executing the program until it stops again.

The target will do the operations as expected, until GDB requests that an instruction in an invalid address is replaced. After the `c` packet, the target will not return to communicate with the host device, and the program is left hanging waiting for an answer.

Each packet and the reply is behaving as expected. However, I can't find documentation on how to handle the situation where GDB is referencing invalid addresses. Through analysing the assembly file and comparing with the program counter values, I am lead to believe that this happens once the control reaches the end of the main function.

From this point, the only possible thing to do is to kill the GDB program, and to re-download the program to Patmos.

4.4 The `break *address` command

Since the debugger can't obtain debug information about program lines – which are useful to set breakpoints –, this command comes in handy. It places a breakpoint on the address specified as a hexadecimal value.

Figure 10 shows how this program will make the host send a number of `m` packets. One might expect a `Z0` packet to be sent as a result of this command, but in fact it is only sent along a command to execute the program (such as `continue` or `next`, among others).

4.5 The `continue` command

Now that one breakpoint is set, running the `continue` command should run the program until the address where the breakpoint is set. The messages exchanged are similar to those of the `next` program, however, only one trap instruction will be set in total – in the address that the user set with the `break` command. As shown in Figure 11, the command works as expected, and the program stops in the address specified.

4.6 Another `break` command

To further test the debugger, another breakpoint in a specific address was set. Just like the previous one, this will result on the sending and replying to many `m` packets, as seen on Figure 12.

4.7 The `step` command

Now that we have a breakpoint, commands like `step` and `next` should terminate. In terms of message exchange, the execution is very similar to that of the `next` command, as can be observed in Figure 13 and 14.

Even though the command description states that it should run the next line of the program, what is verified is that the command makes the program execute until the next breakpoint is reached. This is probably because of the lack of debug information on the ELF files, as indicated by the message on the GDB output: "Single stepping until exit from function main, which has no line number information."

4.8 The `info registers` and `info locals` commands

We can display information on the registers with the `info registers`. This will show the values on the registers as they were described in the last `g` packet, and two `p` packets will be sent. The target will respond with `xxxxxxxx`, as explained in subsubsection 3.5.5. The output can be found on Figure 15.

On the other handle, the `info locals` command won't work at all because of the lack of debug symbols on the binary file. GDB displays an error which can be seen in Figure 16.

4.9 The `quit` command

When the debugging session is over, the user can execute the `quit` command to detatch the host from the target. This will make the target send a `D` packet.

4.10 Overall results discussion

In this section we've gone over a few useful GDB commands that are used in debugging sessions. While testing these commands, it was important to verify that the data that the target is sending makes sense. This was done mostly through the analysis of the ELF file, which was converted into assembly with the following command:

```
patmos-llvm-objdump -d <ELF file path>
```

in which `<ELF file path>` is the path to the ELF file that resulted from the compilation of the program.

The target was able to successfully replace the instructions as instructed by the host, which reflects on the GDB commands such as `continue` stopping in the correct program counter address.

However, there is a clear problem in the implementation. `z0` packets are used to restore the original state of an address that was replaced by a trap. By analysing GDB commands, we can see that these packets are usually sent after a `c` packet, and they usually refer to the program counter address.

For example, let's assume the program halted because of a breakpoint on address `0x22614`. When we want to continue the program execution, GDB will send a `z0` packet with this address. This is so that the target can execute the original instruction that had been replaced. However, when the program can continue execution, it will start from the next instruction – in this example, on address `0x22618` –, which means that the original instruction on `0x22614` was not executed at all.

In instructions such as `step` and `next`, where GDB is sending `Z0` and `z0` packets for almost every instruction, this means that almost no instruction actually gets executed.

5 Conclusion and next steps

I consider this project to have been challenging. I can enumerate a few reasons for this:

- the existing GDB documentation doesn't seem to be very oriented for people that have the same goal as me – implementing debugging capabilities in a remote target.
- the examples provided are very complex and have almost inexistent explanation.
- it is difficult to find answers to questions online regarding errors or problems about GDB, which means that many solutions were found based on a trial and error procedure.

Nevertheless, even though the final product is not a usable debugger, I consider a few things as a success:

- a debugging session between the target and the host is created.
- the host and the target can successfully communicate.
- the target can handle all packets that the target sends to it, even though some shortcuts were taken for simplicity.
- the target can inform the host of the register values, and the host can understand where the control flow is from the program counter.
- the host can instruct the target to set breakpoints, and the program is successfully halting in the correct addresses.

However, to have a fully functioning debugger, there are a few tasks that need to be addressed:

- the GDB needs to be compiled with an accurate description of the Patmos architecture, so that it can understand the instructions that are sent to it, among other things.
- the stub functions should be added to the program without the developer having to do so manually. In this project, I manually inserted the 2 setup functions directly on the main function of the program. Ideally, these functions would execute without the need of changing the source file of the program to debug.

  A solution could be to execute this functions on the program bootloader. This wasn't done in the scope of this project, because it would involve a considerable amount of time. This is because the code is heavily dependent on standard library functions, which would need to be implemented directly on the bootloader, without the use of external libraries.

- in the subsection 4.10 we discussed a problem: the original instructions that had been previously replaced by a trap instruction will never be executed. This means that the debugger as is right now will not execute all instructions as defined in the source code. This should be fixed to have a proper debugging session with correct results.

Furthermore, the fact that the compiled ELF doesn't have debug information, means that it will be impossible to get detailed information from a debugger program. If the goal is to have good debugger support, then the compiler needs to be extended to have the functionality of producing a compiled file with debugging symbols.

List of Figures

1 Sequence diagram modelling the initial setup between the target and the host. 8
2 Part of the output of executing `readelf <ELF file path> -e`, where `<ELF file path>` is the path to the compiled ELF file. 12
4 `AluLongImm` instruction format. Image taken from the Patmos Reference Handbook [3, p. 18]. 14
3 Sequence diagram for the next 15
5 Screen capture of the GDB program during the initial setup when the host connects to the target – part 1. III
6 Screen capture of the GDB program during the initial setup when the host connects to the target – part 2. IV
7 Screen capture of the GDB program when executing a `next` command – part 1. IV
8 Screen capture of the GDB program when executing a `next` command (continuation) – part 2. V
9 Screen capture of the GDB program when executing a `next` command (last part) – part 3. V
10 Screen capture of the GDB program when executing the `break *0x22658` command. V
11 Screen capture of the GDB program when executing the `continue` command. VI
12 Screen capture of the GDB program when executing the `break *0x22668` command. VI
13 Screen capture of the GDB program when executing the `step` command (begining) – part 1. VII
14 Screen capture of the GDB program when executing the `step` command (final part) – part 2. VII
15 Screen capture of the GDB program when executing the `info registers` command. VIII
16 Screen capture of the GDB program when executing the `info locals` commandVIII
17 Screen capture of the GDB program when executing the `quit`command. . . . VIII

List of Tables

1 Table representing registers as described in the XML file 6

References

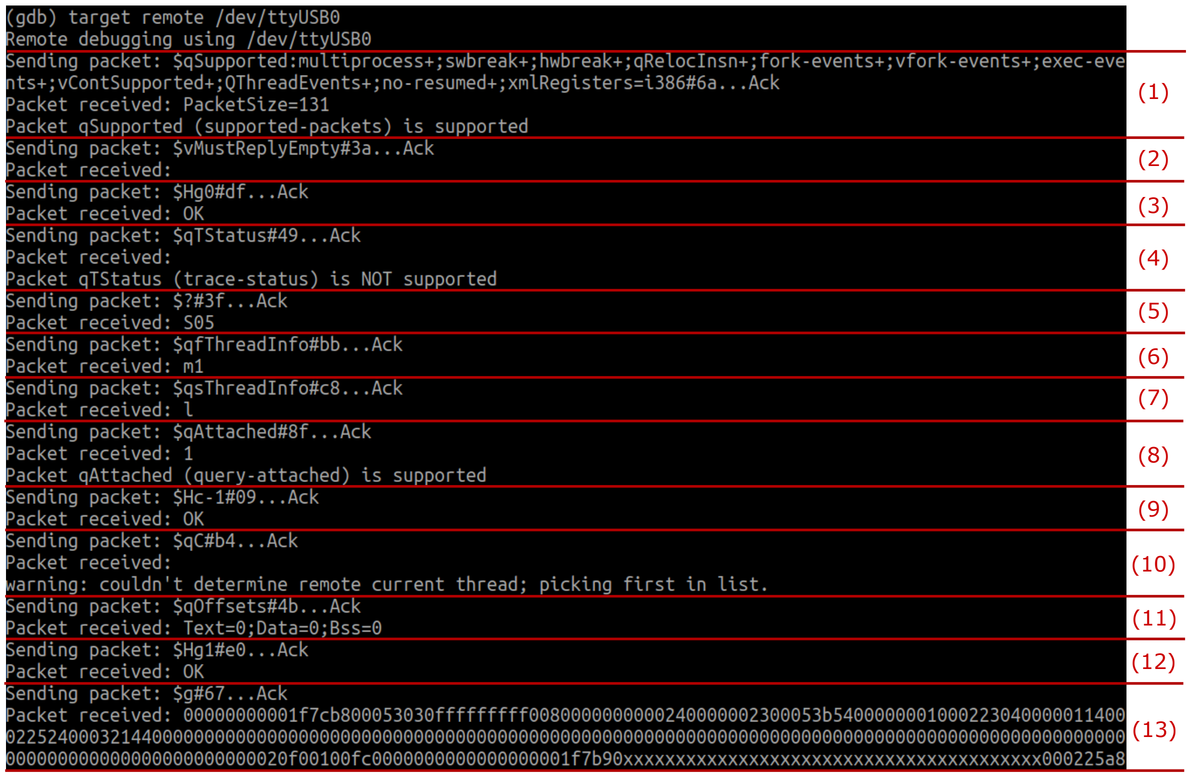
[1] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, and C. W. Probst, “Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach,” in *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, Open Access Series in Informatics (OASIcs), pp. 11–21, 2011.

[2] “Gdb wiki - systems.” `https://sourceware.org/gdb/wiki/Systems`. Accessed: 29-04-2024.

[3] M. Schoeberl, F. Brandner, H. Stefan, W. Puffitsch, and D. Prokesch, *Patmos Reference Handbook*. 2023.

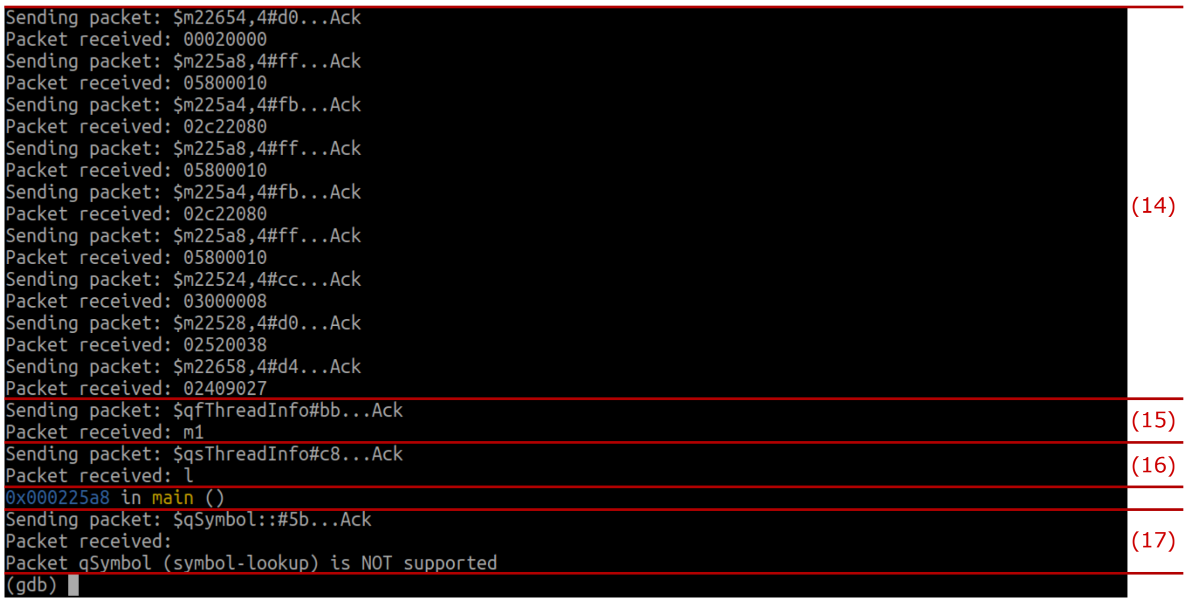
[4] R. Stallman, R. Pesch, and S. Shebs, *Debugging with GDB*.

A Appendix I: Screen captures of results



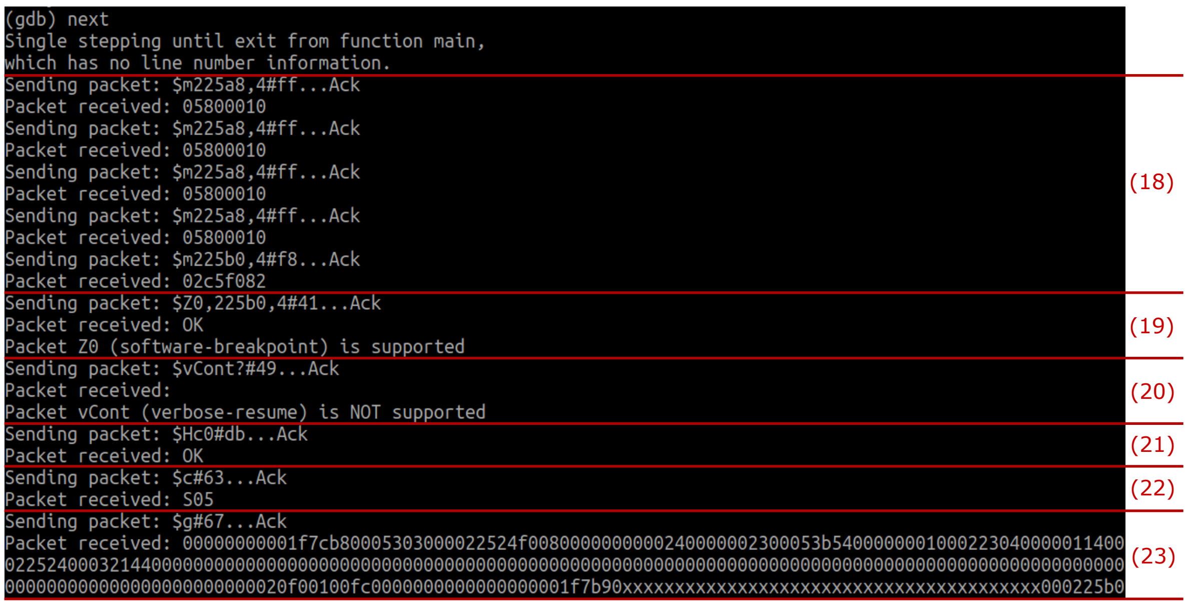
```
(gdb) target remote /dev/ttyUSB0
Remote debugging using /dev/ttyUSB0
Sending packet: $qSupported:multiprocess+;swbreak+;hwbreak+;qRelocInsn+;fork-events+;vfork-events+;exec-eve
nts+;vContSupported+;QThreadEvents+;no-resumed+;xmlRegisters=i386#6a...Ack                                  (1)
Packet received: PacketSize=131
Packet qSupported (supported-packets) is supported
Sending packet: $vMustReplyEmpty#3a...Ack                                                                   (2)
Packet received:
Sending packet: $Hg0#df...Ack                                                                               (3)
Packet received: OK
Sending packet: $qTStatus#49...Ack
Packet received:                                                                                            (4)
Packet qTStatus (trace-status) is NOT supported
Sending packet: $?#3f...Ack                                                                                 (5)
Packet received: S05
Sending packet: $qfThreadInfo#bb...Ack                                                                      (6)
Packet received: m1
Sending packet: $qsThreadInfo#c8...Ack                                                                      (7)
Packet received: l
Sending packet: $qAttached#8f...Ack
Packet received: 1                                                                                          (8)
Packet qAttached (query-attached) is supported
Sending packet: $Hc-1#09...Ack                                                                              (9)
Packet received: OK
Sending packet: $qC#b4...Ack
Packet received:                                                                                            (10)
warning: couldn't determine remote current thread; picking first in list.
Sending packet: $qOffsets#4b...Ack                                                                          (11)
Packet received: Text=0;Data=0;Bss=0
Sending packet: $Hg1#e0...Ack                                                                               (12)
Packet received: OK
Sending packet: $g#67...Ack
Packet received: 00000000001f7cb800053030ffffffff0080000000000240000002300053b54000000001000223040000011400
02252400032144000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000 (13)
0000000000000000000000000020f00100fc00000000000000000001f7b90xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx000225a8
```

Figure 5: Screen capture of the GDB program during the initial setup when the host connects to the target – part 1.



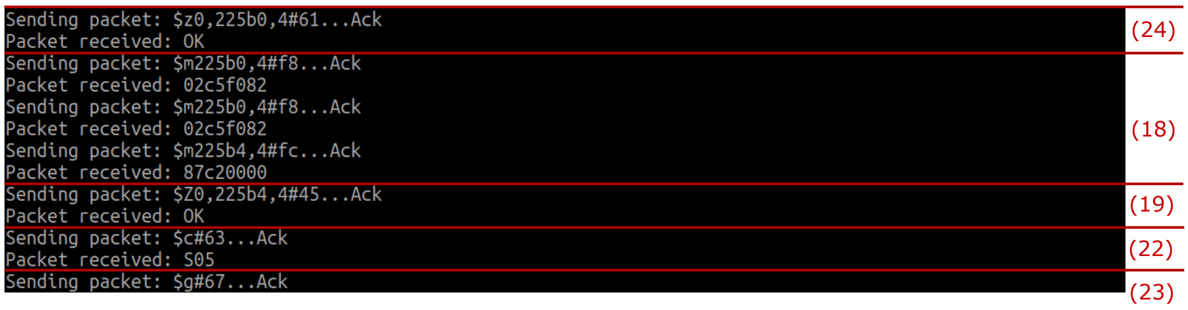
```
Sending packet: $m22654,4#d0...Ack
Packet received: 00020000
Sending packet: $m225a8,4#ff...Ack
Packet received: 05800010
Sending packet: $m225a4,4#fb...Ack
Packet received: 02c22080
Sending packet: $m225a8,4#ff...Ack
Packet received: 05800010
Sending packet: $m225a4,4#fb...Ack
Packet received: 02c22080
Sending packet: $m225a8,4#ff...Ack
Packet received: 05800010
Sending packet: $m22524,4#cc...Ack
Packet received: 03000008
Sending packet: $m22528,4#d0...Ack
Packet received: 02520038
Sending packet: $m22658,4#d4...Ack
Packet received: 02409027
Sending packet: $qfThreadInfo#bb...Ack
Packet received: m1
Sending packet: $qsThreadInfo#c8...Ack
Packet received: l
0x000225a8 in main ()
Sending packet: $qSymbol::#5b...Ack
Packet received:
Packet qSymbol (symbol-lookup) is NOT supported
(gdb) 
```

Figure 6: Screen capture of the GDB program during the initial setup when the host connects to the target – part 2.



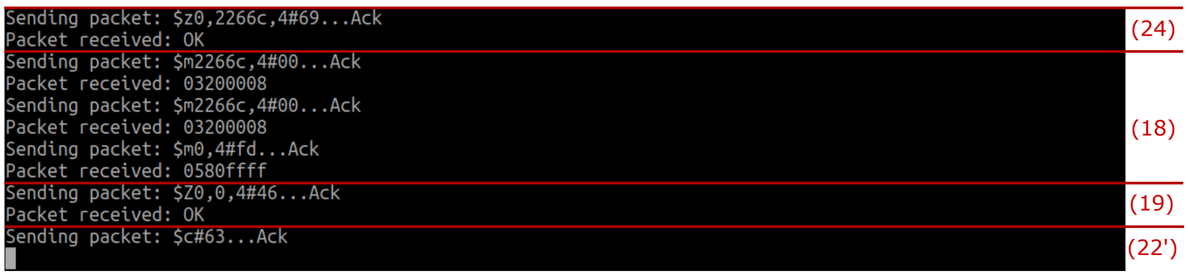
```
(gdb) next
Single stepping until exit from function main,
which has no line number information.
Sending packet: $m225a8,4#ff...Ack
Packet received: 05800010
Sending packet: $m225a8,4#ff...Ack
Packet received: 05800010
Sending packet: $m225a8,4#ff...Ack
Packet received: 05800010
Sending packet: $m225a8,4#ff...Ack
Packet received: 05800010
Sending packet: $m225b0,4#f8...Ack
Packet received: 02c5f082
Sending packet: $Z0,225b0,4#41...Ack
Packet received: OK
Packet Z0 (software-breakpoint) is supported
Sending packet: $vCont?#49...Ack
Packet received:
Packet vCont (verbose-resume) is NOT supported
Sending packet: $Hc0#db...Ack
Packet received: OK
Sending packet: $c#63...Ack
Packet received: S05
Sending packet: $g#67...Ack
Packet received: 00000000001f7cb80005303000022524f00800000000000240000000230053b54000000001000223040000011400
0225240003214400000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000020f00100fc0000000000000000001f7b90xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx000225b0
```

Figure 7: Screen capture of the GDB program when executing a `next` command – part 1.



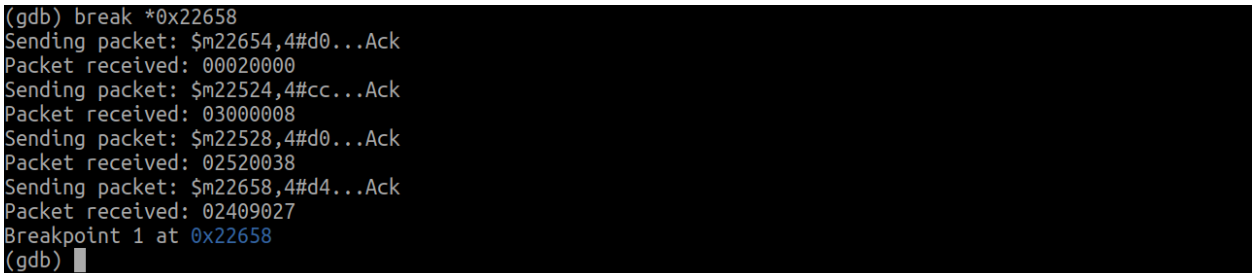
```
Sending packet: $z0,225b0,4#61...Ack          (24)
Packet received: OK
Sending packet: $m225b0,4#f8...Ack
Packet received: 02c5f082
Sending packet: $m225b0,4#f8...Ack            (18)
Packet received: 02c5f082
Sending packet: $m225b4,4#fc...Ack
Packet received: 87c20000
Sending packet: $Z0,225b4,4#45...Ack          (19)
Packet received: OK
Sending packet: $c#63...Ack                   (22)
Packet received: S05
Sending packet: $g#67...Ack                   (23)
```

Figure 8: Screen capture of the GDB program when executing a `next` command (continuation) – part 2.



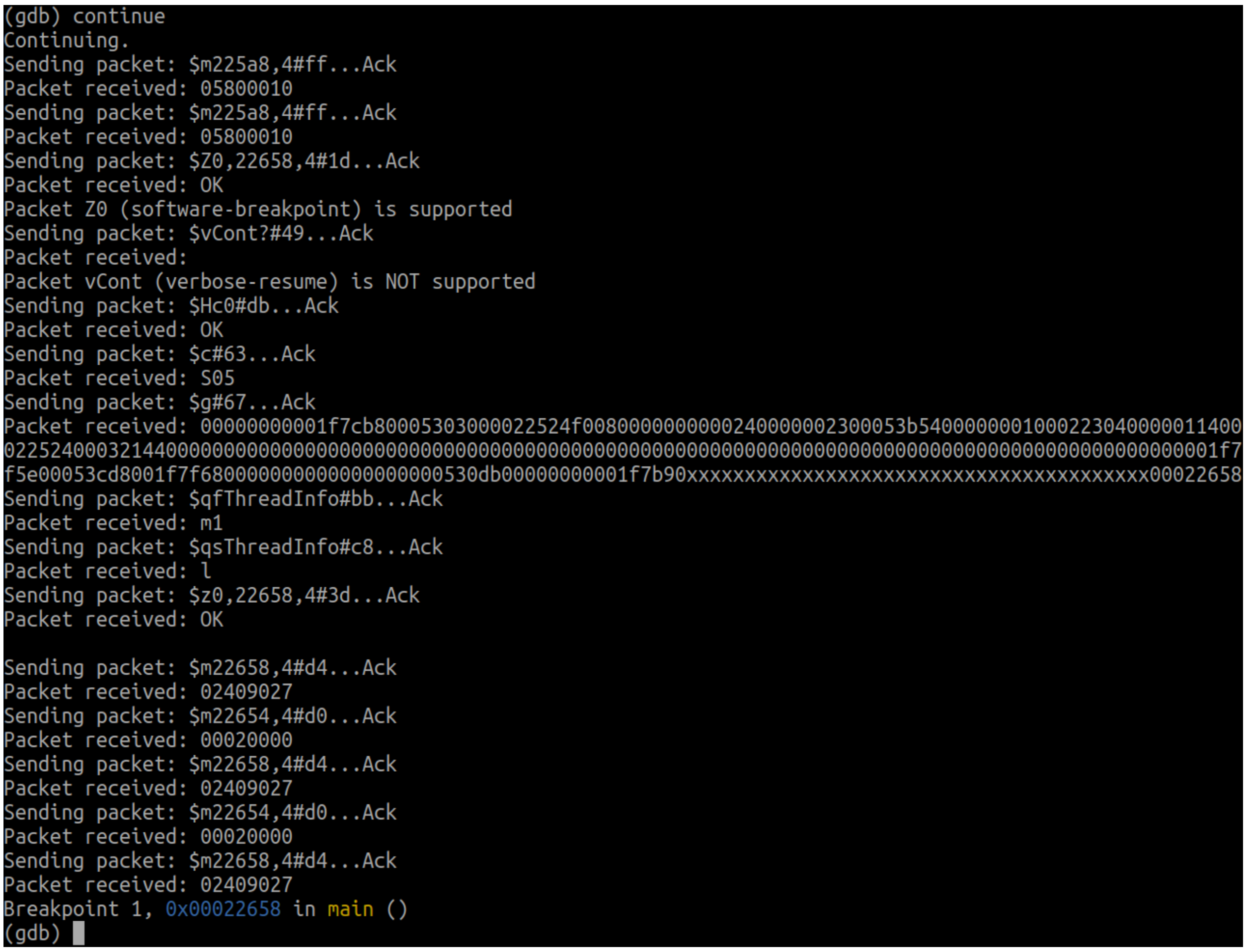
```
Sending packet: $z0,2266c,4#69...Ack          (24)
Packet received: OK
Sending packet: $m2266c,4#00...Ack
Packet received: 03200008
Sending packet: $m2266c,4#00...Ack            (18)
Packet received: 03200008
Sending packet: $m0,4#fd...Ack
Packet received: 0580ffff
Sending packet: $Z0,0,4#46...Ack              (19)
Packet received: OK
Sending packet: $c#63...Ack                   (22')
```

Figure 9: Screen capture of the GDB program when executing a `next` command (last part) – part 3.



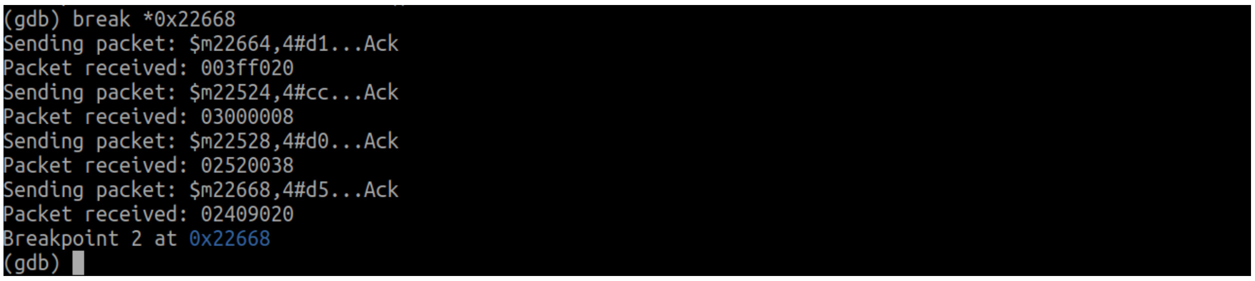
```
(gdb) break *0x22658
Sending packet: $m22654,4#d0...Ack
Packet received: 00020000
Sending packet: $m22524,4#cc...Ack
Packet received: 03000008
Sending packet: $m22528,4#d0...Ack
Packet received: 02520038
Sending packet: $m22658,4#d4...Ack
Packet received: 02409027
Breakpoint 1 at 0x22658
(gdb)
```

Figure 10: Screen capture of the GDB program when executing the `break *0x22658` command.



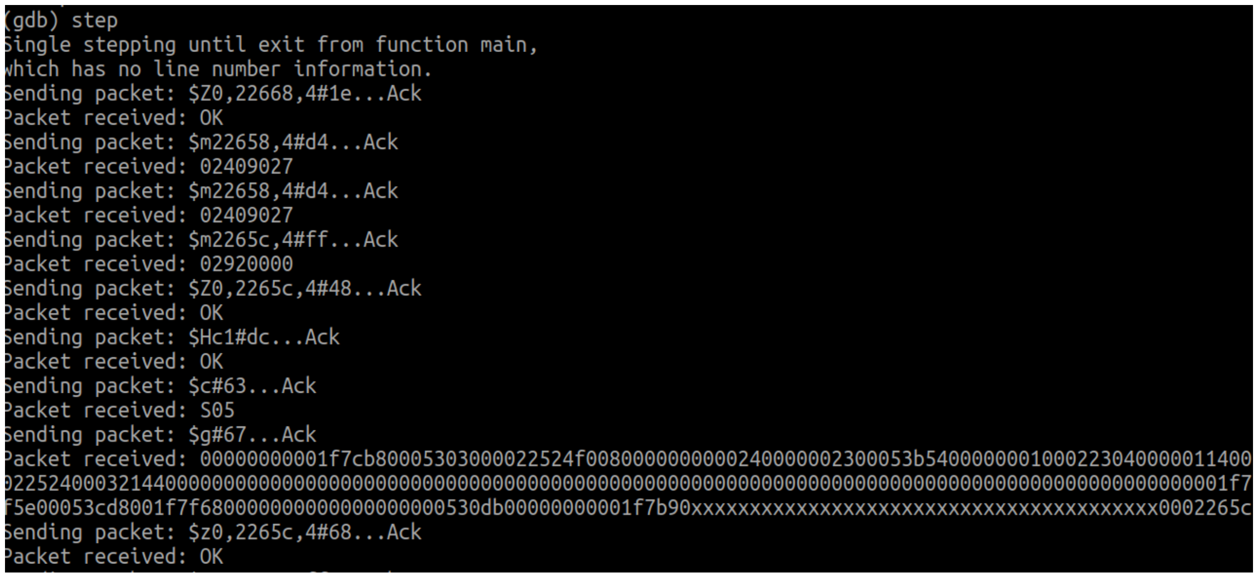
```
(gdb) continue
Continuing.
Sending packet: $m225a8,4#ff...Ack
Packet received: 05800010
Sending packet: $m225a8,4#ff...Ack
Packet received: 05800010
Sending packet: $Z0,22658,4#1d...Ack
Packet received: OK
Packet Z0 (software-breakpoint) is supported
Sending packet: $vCont?#49...Ack
Packet received:
Packet vCont (verbose-resume) is NOT supported
Sending packet: $Hc0#db...Ack
Packet received: OK
Sending packet: $c#63...Ack
Packet received: S05
Sending packet: $g#67...Ack
Packet received: 00000000001f7cb80005303000022524f008000000000002400000002300053b5400000000100022304000001140
0022524000321440000000000000000000000000000000000000000000000000000000000000000000000000000000000001f7
f5e00053cd8001f7f680000000000000000000530db00000000001f7b90xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx00022658
Sending packet: $qfThreadInfo#bb...Ack
Packet received: m1
Sending packet: $qsThreadInfo#c8...Ack
Packet received: l
Sending packet: $z0,22658,4#3d...Ack
Packet received: OK

Sending packet: $m22658,4#d4...Ack
Packet received: 02409027
Sending packet: $m22654,4#d0...Ack
Packet received: 00020000
Sending packet: $m22658,4#d4...Ack
Packet received: 02409027
Sending packet: $m22654,4#d0...Ack
Packet received: 00020000
Sending packet: $m22658,4#d4...Ack
Packet received: 02409027
Breakpoint 1, 0x00022658 in main ()
(gdb) 
```

Figure 11: Screen capture of the GDB program when executing the `continue` command.



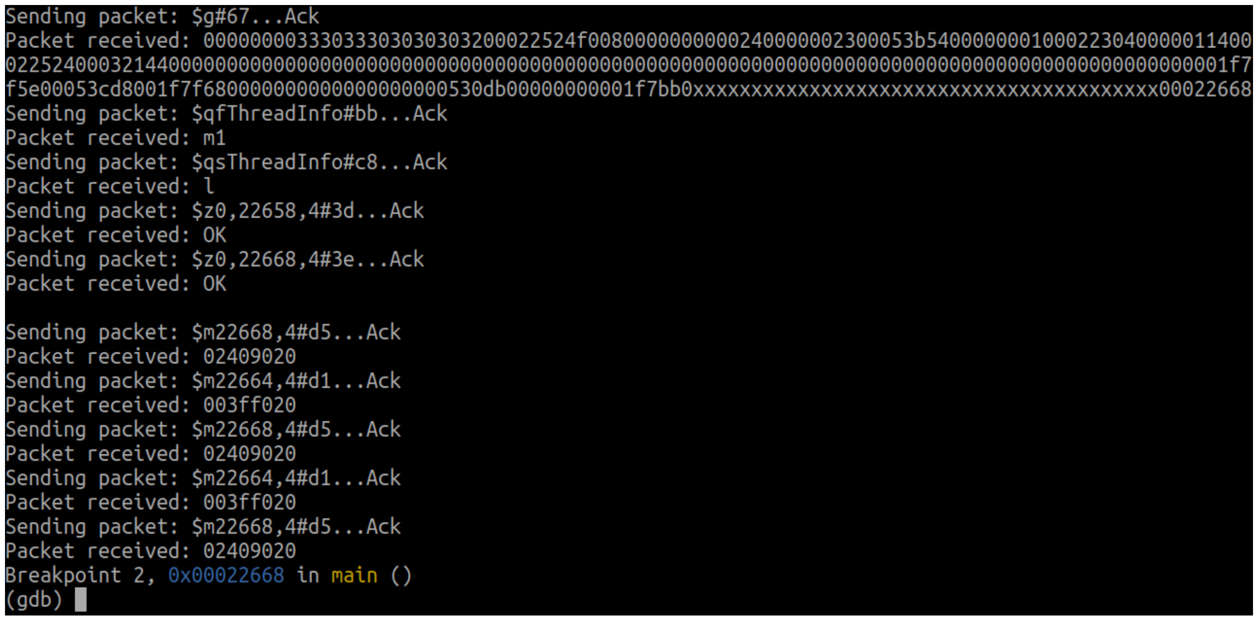
```
(gdb) break *0x22668
Sending packet: $m22664,4#d1...Ack
Packet received: 003ff020
Sending packet: $m22524,4#cc...Ack
Packet received: 03000008
Sending packet: $m22528,4#d0...Ack
Packet received: 02520038
Sending packet: $m22668,4#d5...Ack
Packet received: 02409020
Breakpoint 2 at 0x22668
(gdb) 
```

Figure 12: Screen capture of the GDB program when executing the `break *0x22668` command.



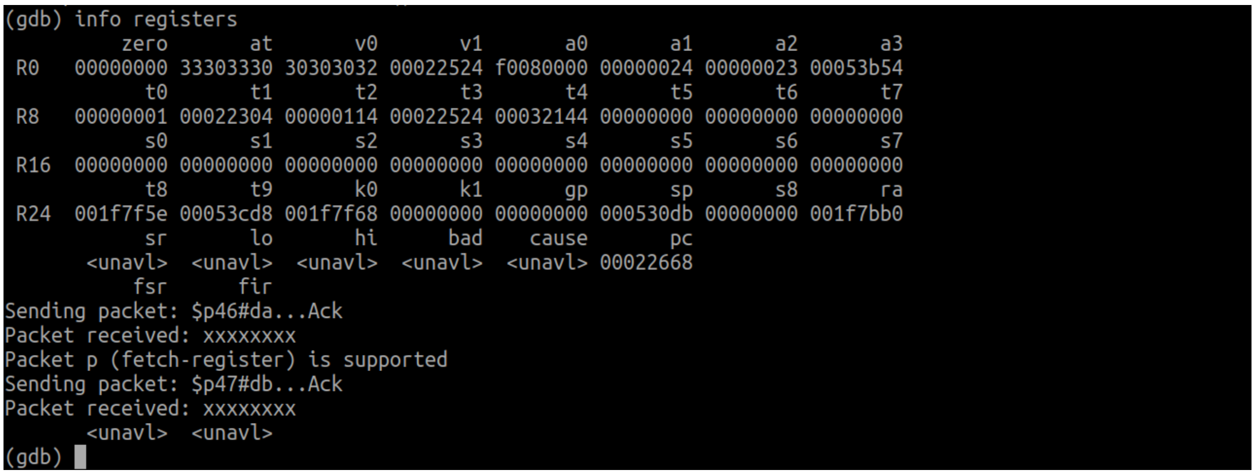
```
(gdb) step
Single stepping until exit from function main,
which has no line number information.
Sending packet: $Z0,22668,4#1e...Ack
Packet received: OK
Sending packet: $m22658,4#d4...Ack
Packet received: 02409027
Sending packet: $m22658,4#d4...Ack
Packet received: 02409027
Sending packet: $m2265c,4#ff...Ack
Packet received: 02920000
Sending packet: $Z0,2265c,4#48...Ack
Packet received: OK
Sending packet: $Hc1#dc...Ack
Packet received: OK
Sending packet: $c#63...Ack
Packet received: S05
Sending packet: $g#67...Ack
Packet received: 00000000001f7cb80005303000022524f0080000000000002400000023000053b5400000000100022304000011400
0225240003214400000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001f7
f5e00053cd8001f7f6800000000000000000000530db00000000001f7b90xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx0002265c
Sending packet: $z0,2265c,4#68...Ack
Packet received: OK
```

Figure 13: Screen capture of the GDB program when executing the `step` command (beginning) – part 1.



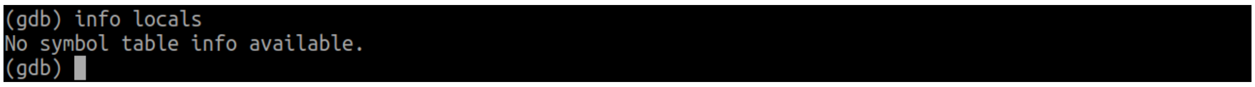
```
Sending packet: $g#67...Ack
Packet received: 000000003330333030303032000022524f0080000000000002400000023000053b5400000000100022304000011400
0225240003214400000000000000000000000000000000000000000000000000000000000000000000000000000000000000000001f7
f5e00053cd8001f7f6800000000000000000000530db00000000001f7bb0xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx00022668
Sending packet: $qfThreadInfo#bb...Ack
Packet received: m1
Sending packet: $qsThreadInfo#c8...Ack
Packet received: l
Sending packet: $z0,22658,4#3d...Ack
Packet received: OK
Sending packet: $z0,22668,4#3e...Ack
Packet received: OK

Sending packet: $m22668,4#d5...Ack
Packet received: 02409020
Sending packet: $m22664,4#d1...Ack
Packet received: 003ff020
Sending packet: $m22668,4#d5...Ack
Packet received: 02409020
Sending packet: $m22664,4#d1...Ack
Packet received: 003ff020
Sending packet: $m22668,4#d5...Ack
Packet received: 02409020
Breakpoint 2, 0x00022668 in main ()
(gdb) 
```

Figure 14: Screen capture of the GDB program when executing the `step` command (final part) – part 2.



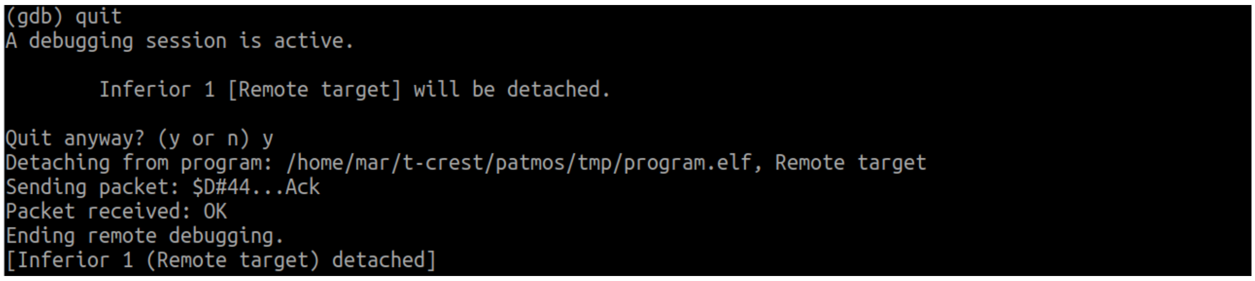
```
(gdb) info registers
          zero       at       v0       v1       a0       a1       a2       a3
 R0   00000000 33303330 30303032 00022524 f0080000 00000024 00000023 00053b54
            t0       t1       t2       t3       t4       t5       t6       t7
 R8   00000001 00022304 00000114 00022524 00032144 00000000 00000000 00000000
            s0       s1       s2       s3       s4       s5       s6       s7
 R16  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
            t8       t9       k0       k1       gp       sp       s8       ra
 R24  001f7f5e 00053cd8 001f7f68 00000000 00000000 000530db 00000000 001f7bb0
            sr       lo       hi      bad    cause       pc
       <unavl>  <unavl>  <unavl>  <unavl>  <unavl> 00022668
           fsr      fir
Sending packet: $p46#da...Ack
Packet received: xxxxxxxx
Packet p (fetch-register) is supported
Sending packet: $p47#db...Ack
Packet received: xxxxxxxx
       <unavl>  <unavl>
(gdb) 
```

Figure 15: Screen capture of the GDB program when executing the `info registers` command.



```
(gdb) info locals
No symbol table info available.
(gdb) 
```

Figure 16: Screen capture of the GDB program when executing the `info locals` command.



```
(gdb) quit
A debugging session is active.

        Inferior 1 [Remote target] will be detached.

Quit anyway? (y or n) y
Detaching from program: /home/mar/t-crest/patmos/tmp/program.elf, Remote target
Sending packet: $D#44...Ack
Packet received: OK
Ending remote debugging.
[Inferior 1 (Remote target) detached]
```

Figure 17: Screen capture of the GDB program when executing the `quit`command.