GNU Debugger for Single-ISA Heterogeneous MAny-Core System (SHMAC)

Bjørn Christian Seime

Master of Science in Computer Science

Submission date:	June 2014
Supervisor:	Lasse Natvig, IDI
Co-supervisor:	Asbjørn Djupdal, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

SHMAC is an FPGA-based multicore prototype developed in a research project within the Energy Efficient Computing Systems (EECS) strategic research area. SHMAC is planned to be an evaluation platform for research on heterogeneous multicore systems. The goal of the SHMAC project is to propose software and hardware solutions for future power-limited, heterogeneous systems. The main goal of this master thesis project is to implement debugging support for SHMAC. The first part is to implement general support for GDB by developing a GDB stub for SHMAC and necessary host driver support. The second part is to facilitate kernel debugging of Barrelfish on SHMAC. If time permits, the student should also implement and document

- debugging support for multiple SHMAC cores.
- debugging support for user space programs on Barrelfish.
- debugging support for Linux kernel on SHMAC.
- more advanced debugger support of multiple cores with some form of visualisation of core usage and communication.

Supervisor: Lasse Natvig and Asbjørn Djupdal.

Abstract

Processors have historically attained performance improvements primarily by increasing frequency and the number of transistors. As the transistor density increases, keeping the power density constant gets harder. As a result, future processors will not be able to power all transistors simultaneously without exceeding the power budget. This phenomenon is coined *Dark Silicon*, referring to the part of the silicon that must be left unpowered. The issue with dark silicon can be mitigated by building heterogeneous computing systems. Such systems consist of several specialised components, each highly efficient in performing a specific task and workload.

The SHMAC project was initiated by NTNU to investigate the challenges in designing heterogeneous computing systems. The output of the project is a heterogenous processor called SHMAC, which has an architecture consisting of a grid of computing tiles. One of the available computing tile is an ARMv3 compliant CPU core. The current software for SHMAC is primarily using this tile as the target CPU.

This thesis presents the first functional debugger for SHMAC. The debugger is based on the *GNU Debugger* (GDB), a popular open-source debugger maintained by the *Free Software Foundation.* Future software development on SHMAC will greatly benefit from having a proper tool for debugging. Another contribution is the integration of the debugger with Barrelfish, the first functional operating system for SHMAC. The integration facilitates kernel debugging and debugging of user programs running on Barrelfish.

Sammendrag

Prosessorer har historisk sett oppnådd forbedringer i ytelse ved å øke frekvensen og antall transistorer. Etter hvert som transistor-tettheten øker blir det vanskeligere å holde energiforbruket per areal konstant. Resultatet er at fremtidens prosessorer ikke vil kunne bruke alle transistorer samtidig uten å overskride tillatt effektnivå. Denne trenden blir kalt for *Dark Silicon*, som refererer til den del av silisiumet som må være avskrudd. Problemene med *dark silicon* kan delvis løses ved utvikling heterogene datamaskiner. Slike maskiner består av flere spesialiserte komponenter, der hver komponent er effektiv i å løse en spesifik oppgave av en gitt størrelse.

SHMAC-prosjektet ble startet av NTNU for å undersøke utfordringene rundt utvikling av heterogene datamaskiner. En heterogen prosessor kalt SHMAC er et resultat av dette prosjektet. Denne prosessoren har en arkitektur som består av et gitter av beregningsenheter. En av de tilgjengelige beregningsenhetene er en ARMv3-kompatibel CPU-kjerne. Nåværende programvare for SHMAC er primært laget for denne beregningsenheten.

Denne hovedoppgaven presenterer den første fungerende debuggeren for SHMAC. Debuggeren er basert på *GNU Debugger* (GDB), en velkjent debugger fra *Free Software Foundation*. Fremtidig programvareutvikling for SHMAC vil ha stor nytte av å ha et skikkelig verktøy for feilsøking. Et annet bidrag er integrasjon av debuggeren mot Barrelfish, det første funksjonelle operativsystemet for SHMAC. Denne integrasjonen forenkler feilsøking av operativsystemkjernen og programmer som kjører på Barrelfish.

Preface

This report is submitted to the Norwegian University of Science and Technology in fulfillment of the requirements for master thesis.

This work has been performed at the Department of Computer and Information Science, NTNU, with Prof. Lasse Natvig as the supervisor, and Asbjørn Djupdal as co-supervisor.

Acknowledgments

Thanks to Benjamin Bjørnseth for providing outstanding technical assistance with the ARM toolchain, SHMAC and Barrelfish.

Thanks to Asbjørn Djupdal for producing an enhanced ABP tile for the only purpose of aiding this project.

Thanks to Håkon Amundsen and Joakim Andersson for providing me with their ARMv3 version of the Amber CPU core.

Contents

List of Figures

List of Tables

List of Abbreviations

APB Advanced Peripheral Bus.

CLI Command Line Interface.

CPSR Current Program Status Register.

CPU Central Processing Unit.

FIQ Fast Interrupt.

FPGA Field Programmable Gate Array.

FSM Finite-State Machine.

GCC GNU Compiler Collection.

GDB GNU Debugger.

GNU GNU's not UNIX.

ILP Instruction Level Parallelism.

IRQ Interrupt Request.

ISA Instruction Set Architecture.

LR Link Register.

OS Operating System.

PC Program Counter.

POSIX Portable Operating System Interface.

RAM Random Access Memory.

RISC	Reduced Instruction Set Computing.
SHMAC	Single ISA Heterogeneous MAny-core Computer.
SPSR	Saved Program Status Register.
SVC	Supervisor Call.
SWI	Software Interrupt.
TTY	TeleTYpewriter.
UND	Undefined.

Chapter 1 Introduction

1.1 Trends in Computer Architecture

The processor has seen an impressive increase in performance since its conception. Most of the performance gains has been achieved by increasing the Instruction Level Parallelism (ILP). Increasing the ILP benefits the single core performance, but requires the use of more transistors. The result is that the number of transistors per processor has increased by an exponential rate. This trend is known as *Moore's Law*, named after the Intel co-founder who described the trend in a paper from 1965 [M+65].

The performance increase through ILP and frequency scaling stagnated in the early 2000s. The power consumption and the core temperature reached the limit of what cooling systems could handle. The single core designs had become so complex that any increase in ILP would require a substantial increase of transistor usage and power consumption [BC11]. Higher performance had to be gained through other means. The multicore design was the answer to this issue. Multicore processors were designed by placing multiple homogenous processor cores on a single chip.

Processor designs have also relied on *Dennard scaling*, a law related to Moore's Law. Dennard scaling states that power density stays constant as transistors gets smaller [DGR+74]. As the transistor dimension shrinks, it becomes harder to keep reducing the power consumption per transistor, leading to the failure of Dennard scaling. The result is that an exponential increase in number of transistors leads to an exponential increase in total power consumption. Thus only parts of a processor can be powered on at a given time to ensure that the chip does not overheat. This phenomenon is called *Dark Silicon* [EBSA+11].

A potential strategy for mitigating this problem is heterogeneous computing [Tay12]. Instead of making homogenous multicore processors, future multicore processor may consist of several specialized components, each highly efficient in performing a specific task and workload. In theory, such processors may have only the most efficient component enabled at a time, yielding higher performance and power efficiency than traditional, general-purpose processors.

1.2 The SHMAC Project

The *SHMAC* project is a research project initiated by the Energy Efficient Computing Systems (EECS) group at NTNU [EECb]. EECS is a joint research initiative by the *Department of Electronics and Telecommunications* and the *Department of Computer and Information Science* [EECa]. The primary focus is to investigate the challenges in designing heterogeneous computing systems. A subproject of this research initiative is the *Single-ISA Heterogeneous MAny-Core System* (SHMAC). It is a generic architecture for implementing single-ISA heterogeneous multicore systems. The architecture consists of a grid of computing tiles, each implementing a specific routing interface. The most prominent tile available is an ARMv3 compliant CPU core. The current software for SHMAC is primarily using this tile as the target CPU.

1.3 Barrelfish for SHMAC

Barrelfish is an experimental operating system designed for future multi- and many-core systems [Zü, BBD$^+$09]. It is developed by ETH Zürich with collaboration from Microsoft. The motivation behind Barrelfish is the current trend in hardware design where the number of cores and hardware diversity increases. The architecture of Barrelfish is designed after the *multikernel model* which treats the machine as a network of independent cores. The machine is modelled a distributed system; there is no intercore sharing and the cores communicate solely through message passing. Benchmarks on manycore systems have demonstrated that Barrelfish scales better than conventional operating systems such as Linux and Windows.

An initial port of Barrelfish to SHMAC was developed as a result of a student project in fall 2013 [BS13]. The port proved to run stable on SHMAC, although only utilizing a single core. Barrelfish became the first stable operating system for SHMAC. The development of the port continued in spring 2014 with the target goal of adding multicore support [Bjo14].

1.4 Debugging on SHMAC

The report from the Barrelfish porting project stated that the lack of a debugger was a serious limitation with the SHMAC platform. Diagnosing errors was a challenging task that was time consuming and hard to carry out. Debugging was performed by examining the output from inserted print statements and the content of memory dumps taken during execution. These two techniques are far from ideal. Inserting print statements may accidentally alter the semantics of the program and requires recompilation whenever a new print statement is inserted. Examining memory dumps is a complicated task that requires detailed knowledge about how a program is laid out in memory.

A proper debugger will simplify software development on SHMAC. Locating bugs will be easier and less time consuming, enabling software developers to focus on implementing functionality rather than spending time on fixing bugs.

Terje Schjelderup investigated the debugging capabilities of SHMAC in a report from fall 2013 [Sch13]. The report states why having a debugger for SHMAC is important, and it also briefly describe how to add GDB support to SHMAC without having to perform any larger modifications to the SHMAC hardware.

1.5 Assignment Interpretation

The following tasks were extracted from the assignment text:

Mandatory:

T1 Add support for debugging with the GNU Debugger (GDB) on SHMAC. This step includes the development of a GDB remote stub and any host drivers required to enable GDB debugging.

T2 Facilitate debugging of the Barrelfish kernel using the software developed in `T1`.

Optional:

T3 Extend the debugger implementation in `T1` and `T2` to enable debugging of multiple SHMAC CPU cores simultaneously.

T4 Extend the debugger implementation in `T2` to facilitate debugging of user programs in Barrelfish.

T5 Extend the debugger implementation in `T1` to enable debugging of the Linux kernel.

T6 Implement a software providing visualisation of core usage and communication.

Most effort will likely be spent on `T1`. The other tasks are dependent on the output of this task. `T2` is the dependent on the success of `T1`, and cannot be started until the software in `T1` has reached a stable state. `T3` is defined as a separate task from `T1` as it is believed to be technically challenging and time consuming. It is the highest prioritized task among the optional tasks since debugging multicore programs are an essential feature on SHMAC. `T4` provides an extension of the software in `T2`. This task is believed to be a relatively simple once kernel debugging of Barrelfish is finished. `T5` is related to a master thesis working on porting Linux to SHMAC [AA14]. This task will be initiated if they are in great need of a debugger and if time permits. `T6` is not directly related to the other tasks, although it provides SHMAC with monitoring capabilities which can be useful when debugging.

1.6 Contributions

This thesis provides the SHMAC project with a fully functional debugger. The main contribution is support for the *GNU Debugger (GDB)*, an open-source debugger developed and maintained by the *Free Software Foundation*. This thesis describes the implementation of a GDB remote stub, a small software component, which linked to a SHMAC program, enables debugging using GDB.

This thesis does also provide GDB support for the Barrelfish operating system. The GDB integration enables debugging of the OS kernel and system processes.

1.7 Thesis Organization

Chapter 1: Introduction presents the motivation and goals of the Master Thesis.

Chapter 2: Background presents an overview of common debugging concepts, the SHMAC platform, GNU Debugger (GDB) and the Barrelfish operating system.

Chapter 3: GDB for Bare-Metal Programs describes the implementation of the GDB debugger for SHMAC.

Chapter 4: GDB for Barrelfish describes how the debugger was integrated into Barrelfish.

Chapter 5: Testing the debugger describes how the debugger was tested and verified.

Chapter 6: Evaluation evaluates the debugger. Limitations with the current implementation are presented.

Chapter 7: Future work suggests the direction for any future work.

Chapter 8: Conclusion contains the concluding remarks for this project.

Chapter 2
Background

2.1 Debugging

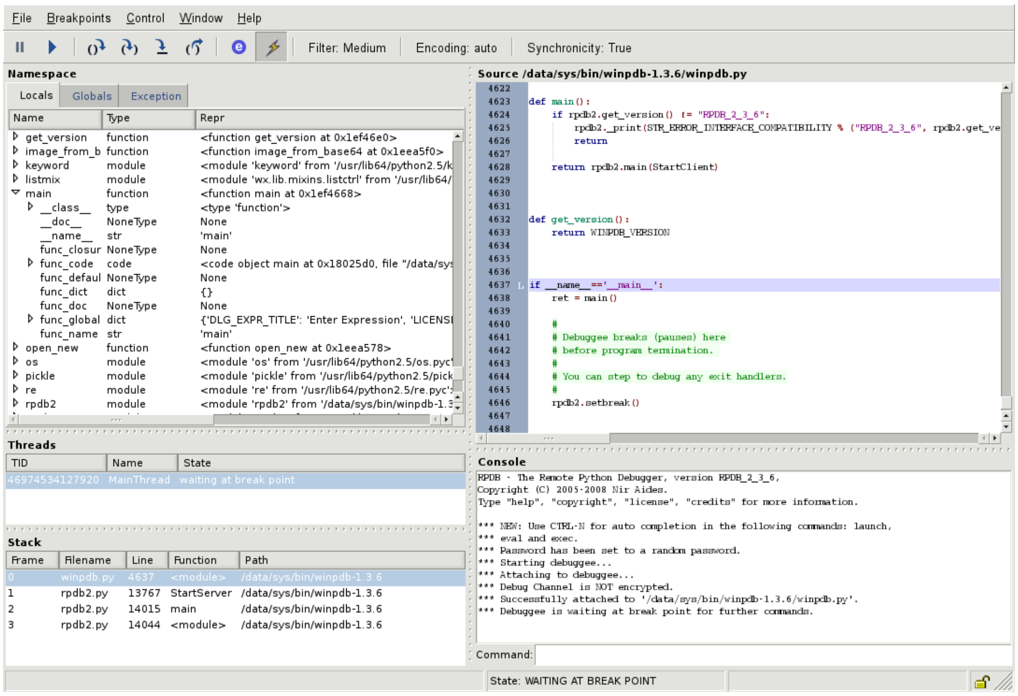
Debugging is the process of discovering and eliminating defects in a computer program. Software defects and anomalies are more commonly known as *bugs*. Debugging is an essential part of software development, as it is hard to write fault-free code on a first attempt, and finding the bugs can be challenging. Static analysis and code reviews will identify some bugs, while other bugs are not discovered until testing or running the program live. Some debugging techniques involves inspecting the state of the system after the program has finished.

2.1.1 Debugger

A debugger is a tool used to facilitate debugging of a program. A debugger provides operations for examining and modifying the program state. In addition, the debugger may control the program execution. The program might be stopped at location, and then later resumed. The program being debugged is usually compiled with a special debug flag. The flag tells the compiler to include *debug symbols*. Debug symbols provides the debugger with details about the source code. This enables the debugger to determine the location of variables in memory, and to map each instruction to lines in the source code.

Some debuggers provide a graphical interface. An example is Winpdb [Aid], which is a debugger for the Python programming language. A screenshot of Winpdb is depicted in Figure 2.1. The upper right corner of the window presents the source code of the program being debugged. The line 4637 is highlighted as it indicates the current position in the program (which is currently paused). At the upper left corner is a list containing symbols, variables and functions, local to the source file. Each symbol is presented with its current value. The call stack is shown in the bottom left corner.

Other debuggers, like GDB [Fre], are controlled through a command line interface. Figure 2.2 shows GDB debugging SHMAC, using the software components produced as part of this master thesis. The debugger is controlled by typing in commands.



Filter: Medium | Encoding: auto | Synchronicity: True

Namespace (Locals)

Name	Type	Repr
get_version	function	<function get_version at 0x1ef46e0>
image_from_b	function	<function image_from_base64 at 0x1eea5f0>
keyword	module	<module 'keyword' from '/usr/lib64/python2.5/k
listmix	module	<module 'wx.lib.mixins.listctrl' from '/usr/lib64/
main	function	<function main at 0x1ef4668>
__class__	type	<type 'function'>
__doc__	NoneType	None
__name__	str	'main'
func_closur	NoneType	None
func_code	code	<code object main at 0x18025d0, file "/data/sys
func_defaul	NoneType	None
func_dict	dict	{}
func_doc	NoneType	None
func_global	dict	{'DLG_EXPR_TITLE': 'Enter Expression', 'LICENSI
func_name	str	'main'
open_new	function	<function open_new at 0x1eea578>
os	module	<module 'os' from '/usr/lib64/python2.5/os.pyc'
pickle	module	<module 'pickle' from '/usr/lib64/python2.5/pick
re	module	<module 're' from '/usr/lib64/python2.5/re.pyc'
rpdb2	module	<module 'rpdb2' from '/data/sys/bin/winpdb-1.3

Source /data/sys/bin/winpdb-1.3.6/winpdb.py

```

def main():
    if rpdb2.get_version() != "RPDB_2_3_6":
        rpdb2._print(STR_ERROR_INTERFACE_COMPATIBILITY % ("RPDB_2_3_6", rpdb2.get_ve
        return

    return rpdb2.main(StartClient)



def get_version():
    return WINPDB_VERSION



if __name__=='__main__':
    ret = main()

    #
    # Debuggee breaks (pauses) here
    # before program termination.
    #
    # You can step to debug any exit handlers.
    #
    rpdb2.setbreak()


```

Threads

TID	Name	State
46974534127920	MainThread	waiting at break point

Stack

Frame	Filename	Line	Function	Path
0	winpdb.py	4637	<module>	/data/sys/bin/winpdb-1.3.6
1	rpdb2.py	13767	StartServer	/data/sys/bin/winpdb-1.3.6
2	rpdb2.py	14015	main	/data/sys/bin/winpdb-1.3.6
3	rpdb2.py	14044	<module>	/data/sys/bin/winpdb-1.3.6

Console

```
RPDB - The Remote Python Debugger, version RPDB_2_3_6,
Copyright (C) 2005-2008 Nir Aides.
Type "help", "copyright", "license", "credits" for more information.

*** NEW: Use CTRL-N for auto completion in the following commands: launch,
*** eval and exec.
*** Password has been set to a random password.
*** Starting debuggee...
*** Attaching to debuggee...
*** Debug Channel is NOT encrypted.
*** Successfully attached to '/data/sys/bin/winpdb-1.3.6/winpdb.py'.
*** Debuggee is waiting at break point for further commands.
```

State: WAITING AT BREAK POINT

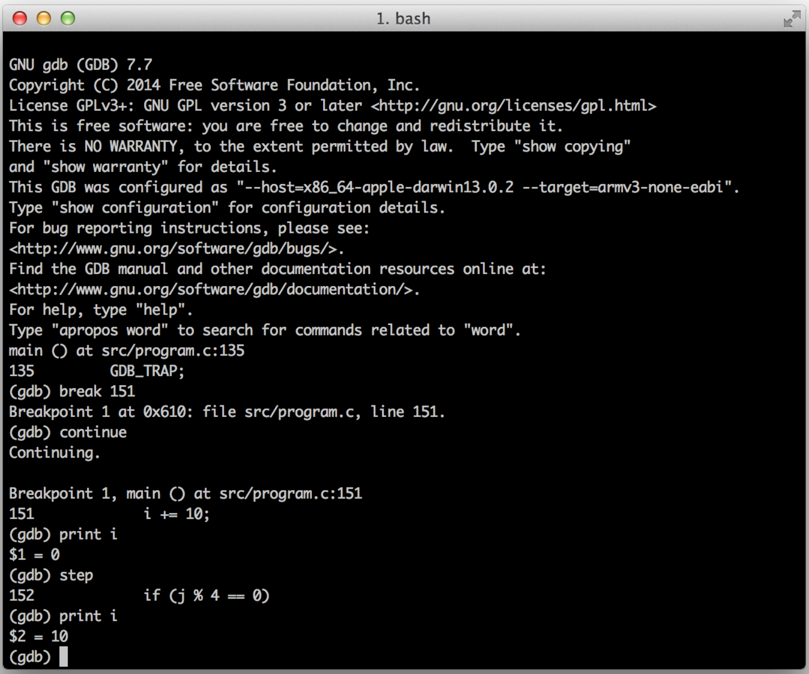
Figure 2.1: Screenshot of Winpdb, a debugger for the Python programming language. Source: Wikimedia [Wik].

2.1.2 Black Box Debugging

It's difficult to debug programs where its inner working is unknown or not available for inspection. A technique called *black box debugging* is employed in those cases [WT03]. Black box debugging collects useful information from sources external to the program. Many bugs can be diagnosed by observing the data that flows between the program and its dependencies. Relevant sources are for instance network traffic, system calls and file operations.

2.1.3 Remote Debugging

Remote debugging is debugging a program where the debugger runs on a different system than the program. The system running the debugger is called the *host*, while the system running the program is called the *target*. The host will usually connect



```
GNU gdb (GDB) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-apple-darwin13.0.2 --target=armv3-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
main () at src/program.c:135
135         GDB_TRAP;
(gdb) break 151
Breakpoint 1 at 0x610: file src/program.c, line 151.
(gdb) continue
Continuing.

Breakpoint 1, main () at src/program.c:151
151             i += 10;
(gdb) print i
$1 = 0
(gdb) step
152             if (j % 4 == 0)
(gdb) print i
$2 = 10
(gdb) 
```

Figure 2.2: Screenshot of GDB while debugging a program on SHMAC.

to the target over a network, for instance over a serial line or using an IP connection. Remote debugging is used when debugging a system which cannot run the debugger itself, typically because the system has limited processing capabilities.

2.1.4 Debugging Techniques without Debugger

Print Debugging

Print debugging is a primitive form of debugging. Print statements are used to output the program's state at different locations during the program execution. One strategy is to print all relevant local and global variables at specific locations, to verify that their values are correct. The print statements are usually removed once the debug session is finished. This strategy is easy to understand, but requires modifications to the source code. Modifying the source code is undesired, as the process might introduce new bugs. The print output can be stored in a log file if standard output is not available.

Analyse Core Dumps

A memory dump, sometimes called core dump, consists of the recorded state of the system at a specific time. A core dump of a program will usually contain the snapshot of the main memory and the values of the CPU registers. This information represents the state of the system at specific time. If the core dump was created at the time of a crash, it can be used to determine the cause.

2.1.5 Debugging Techniques using Debugger

Breakpoints

Breakpoints are used to stop the program execution at specific locations. The location can be a line in the source code or a memory location. The debugger is given control over the system when the program hits a breakpoint. The program's state may then be easily inspected as the program is paused. Breakpoints are usually managed by the debugger; the debugger insert and remove breakpoints during a debugging session. More advanced breakpoints exist in form of conditional breakpoints. Conditional breakpoints will only halt the program if certain conditions are met. The condition is usually specified as a boolean expression which includes program variables. Breakpoints are either implemented in software or in hardware.

Hardware Breakpoint
Some architectures provide support for hardware breakpoints. The breakpoint location is written to a special register file when it is inserted into the program. The processor will generate a trap when it tries to execute the instruction where the breakpoint is located. A processor will usually have a limited capacity for storing hardware breakpoints. So it is not uncommon for a debugger to use both hardware and software breakpoints simultaneously. The Intel x86 ISA has for instance only capacity for 4 hardware breakpoints [Int86].

Software Breakpoint
Software breakpoints are implemented using instructions that traps the processor when executed. The original instruction is substituted for a trap instruction when a breakpoint is inserted. Many architectures have a special trap instruction dedicated for breakpoints. The `int3` instruction in Intel x86 will explicitly call the breakpoint exception handler [Int86]. The ARMv5 ISA introduced the `BKPT` breakpoint instructions, which triggers a prefetch abort on execution [ARM01]. Some architectures, for instance earlier ARM versions, do not provide a dedicated breakpoint instruction. The common solution is to either use a *software interrupt instruction* (`SWI`) or an *undefined instruction* for breakpoints. Both instructions are good candidates as they will generate a trap which will be handled by a dedicated exception handler.

Single Stepping

It is sometimes useful to execute only a single instruction or line of code while debugging. Variables can be examined after each step to verify that the previous operation was correctly executed. In addition, single stepping can be used to determine the actual program flow during execution. Implementations of single stepping in both software and hardware exists. One way of achieving single stepping is by inserting a breakpoint at the next instruction to be executed. Like with breakpoints, Intel x86 does also provide hardware support for single stepping. Setting the `TP` (trap flag) bit in the system register will force the CPU to generate an exception after executing a single instruction [Int86].

Watchpoints

A watchpoint halts the program when the values of an expression or variable changes. Watchpoint are sometimes called data breakpoint. The Intel x86 architecture provides hardware support; the CPU can be programmed to trap when the data at a memory location is modified [Int86].

2.2 GNU Debugger

GDB, shorthand for GNU Debugger, is an open-source debugger [RSea14]. It was originally developed by Richard Stallman, now managed by Free Software Foundation. GDB supports a large variety of architectures like x86, ARM and PowerPC. Language support is provided for C, C++, D, Go, Objective-C, Fortran, Java, OpenCL C, Pascal, Modula-2, Ada and several assembly dialects. GDB is still under active development, with multiple releases every year. The latest version as of March 2014, is GDB 7.7, which was released in February 6th, 2014.

2.2.1 Debugging Target

A target is a term used to describe the program being debugged. The typical use case is to run GDB side-by-side with the target, for instance running the debugger and the program on the same machine as depicted in Figure 2.3. The debugging session is initiated by attaching GDB to an existing process, or by specifying an executable file which GDB will launch and attach to. GDB also supports debugging of recorded sessions, for instance core dumps.

2.2.2 Remote Target

GDB can debug programs running on a remote machine. This is typically the case when debugging on a platform that does not have the necessary operating system support for running GDB directly.

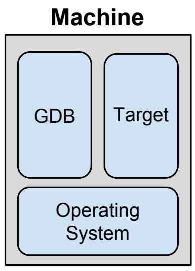


Figure 2.3: Debugging a local program.

A host machine runs GDB, launched with the path of the program file as argument. The host connects to the remote target using either a serial port or a TCP connection. The remote target is either GDBServer instance or a remote stub. The host and the remote target communicates using a text based protocol named *GDB Remote Serial Protocol*. This protocol will be described in more detail in subsection 2.2.3.

GDBServer

GDBServer is a *debug server* running on the remote machine. It appears as a server program which other GDB instances may connect to. The GDB instances debugs the program running on the remote machine through GDBServer. Figure 2.4 illustrates a remote target in combination with GDBServer. GDBServer requires the same operating system facilities as standard GDB program. Its main advantage over GDB is its size. The codebase of GDBServer is smaller, which makes it simpler to add new architecture support. A small program size is also beneficial when running the debugger on a memory-constrained system.

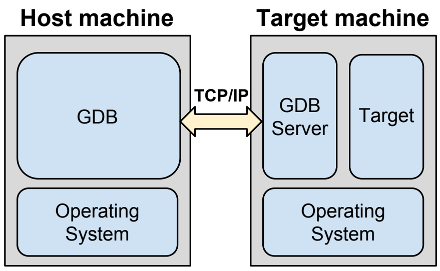


Figure 2.4: Debugging a remote program using GDBServer.

GDB Remote Stub

Another way of debugging a remote program is using a remote stub. The remote stub is a small software component that is linked with the program. It uses the same remote protocol as GDBServer. A remote stub does not require an operating system as opposed to GDBServer. It is commonly used when debugging embedded systems or when debugging the operating system itself. KGDB, the Linux kernel debugger, is an example of a remote stub. Figure 2.5 illustrates a remote target being debugged over a serial line using a remote stub.

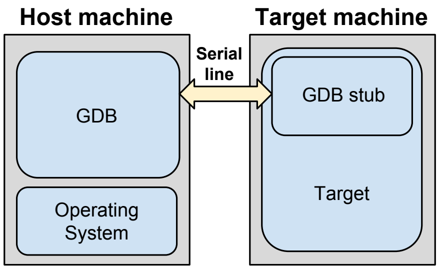


Figure 2.5: Debugging a remote program using a GDB stub.

2.2.3 GDB Remote Serial Protocol

The GDB host communicates with a remote target using the *GDB Remote Serial Protocol*. Data is transferred between host and target as *packets*. A packet has the following structure:

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

A packet starts with a `$` character and ends with a `#` character followed by a two-letter `checksum`. The packet data is usually in ASCII text to support transmissions over mediums that only support 7-bit characters. The receiver of a packet verifies the checksum and sends an acknowledgement with result. An acknowledgement is sent as a single character, not as a packet. The character `+` is replied if the checksum matches the packet data, `-` if not. A negative acknowledgement will usually result in a retransmission of the original packet. The *Remote Serial Protocol* is a request-reply protocol. The host sends packets in form of *commands*, which are replied by the remote with a *response*. Note that *protocol commands* are different from the GDB CLI[1] commands. A CLI command will usually result in several protocol command packets.

Packets

The GDB Remote Serial Protocol defines several commands with associated responses. The type of a command is determined by the first letters in the packet data. These

[1] *Command Line Interface*

letters represents the name of packet. The packet may contain any number of parameters. The parameters are usually separated from each other using either a semi-colon (`;`), comma (`,`) or colon (`:`). An example of a command with response from remote is shown below:

```
host:    $m5cc,4#c8
remote: +$0030a0e1#ea
host:    +
```

The host sends the read memory packet, `m`, requesting 4 bytes from address 0x5cc. The remote first acknowledges the packet with a `+`, then sends the response to the host. The response contains the 4 bytes requested, each encoded as a two-letter hexadecimal number. The bytes `0x00`, `0x30`, `0xA0` and `0xE1` are returned to the host.

The stub is required to return an empty response for commands which it does not support. The older packets have single letter names, while newer packets typically have longer names. All multiletter names are prefixed with one of the letters `q`, `Q` or v.

Packet starting with a lower-case `q` are categorized as general query packets. An example is `qC`, which queries the target for the identifier of the current thread. A prefixed upper-case `Q` is used for general set packets; packets which modifies the remote's state. The prefix `v` is used for all other multiletter packets which do not fit any of the aforementioned categories.

The target sends the host a *stop reply packet* when the program halts. The stop reply packet informs the host of the reason for the halt, e.g. if the program hit a breakpoint or was killed. The stop reason is encoded as a Unix signal; `SIGTRAP` is for instance reported when program traps on a breakpoint.

Obligatory Packets

A remote stub is only required to support a small subset of the available commands. These packet are shown in Table 2.1. Most of the debugging management will in that case be taken care of by the GDB instance running on the host. The host will for instance have to insert and remove breakpoints using the memory access packet if the stub does not support any of the breakpoint packets. Since the stub can support an arbitrary set of packets in addition to the minimum required, the host has to probe the stub for supported packets. Two techniques are used; the host can send the packet to stub and see if an empty response is replied, or query the stub for supported commands using the `qSupported` packet.

Table 2.1: Obligatory packets for a GDB stub. Source: [RSea14, Appendix E].

Packet format	Description	Reply
`g`	Read general registers.	`XX...` Each byte of register data is described by two hex digits. The bytes with the register are transmitted in target byte order.
`G XX...`	Write general registers. Data is in the same format as for `g` packet.	`OK`
`m addr,length`	Read *length* bytes of memory starting at address *addr*.	`XX...` Memory contents; each byte is transmitted as a two-digit hexadecimal number.
`M addr,length:XX...`	Write *length* bytes of memory starting at address *addr*. *XX...* is the data; each byte is transmitted as a two-digit hexadecimal number.	`OK`
`s [addr]`	Single step. *addr* is the address at which to resume. If *addr* is omitted, resume at current address.	*Stop reply packet* when targets halts.
`c [addr]`	Continue. *addr* is address to resume. If *addr* is omitted, resume at current address.	*Stop reply packet* when targets halts.

Interrupt Mechanism

GDB has a mechanism for interrupting a program while its running. This mechanism is triggered if the user press Ctrl+C in the GDB command prompt. The interrupt signal is transferred to the remote as the byte `0x03`. The byte does only represent an interrupt if it is not part of packet data. When the remote receives the byte, it stops the program execution and transfers the control over to the debugger. The remote will then send a *stop reply packet* to notify GDB that the program has stopped.

Multithreading Support

GDB supports debugging of multithreaded programs. Several of the packets accepts a thread identifier to specify the thread of operation. A thread identifier is represented as a positive number. An example is the `vCont` packet, where the host can specify to single step a single thread while halting the others:

```
host:    $vCont;s:1;c#c1
```

The argument `s:1` tells the stub to single step thread #1. The next argument `c` tells the stub to continue execution for the other threads.

There are two modes of controlling execution of the debugger in a multithreaded environment. The default mode is the *all-stop mode*. A more advanced alternative called *non-stop mode* was added in 2009 [GDB09]. The major difference between the modes is that in *all-stop mode*, all threads halt whenever one thread is halted. It is not possible to inspect a single thread while the others are running simultaneously. This limitation is removed in *non-stop mode*, at a cost of a substantially higher complexity, both in semantics and implementation.

Multiprocess Mode

Multiprocess mode enables GDB to attach and detach several processes during one session. This differs from normal behaviour which only supports debugging a single process during a session. The remote protocol supports packets for attaching, detaching and querying processes when in *multiprocess mode*.

2.2.4 GDB for ARM

Most modern ARM microcontrollers provides a dedicated debug port named TAP (Test Access Port), which is part of the JTAG standard [JC09]. This interface talks directly to hardware, eliminating any need for debugging software running on the target. A debugger setup using GDB and JTAG is depicted in Figure 2.6. As a dedicated hardware port is common for most architectures today, there are only a few official remote stubs available. The GDB source code comes with stubs for only 6 architectures; none of which support ARM.

2.3 The SHMAC Platform

The SHMAC platform is a generic architecture for implementing single-ISA heterogeneous multicore systems. It is a tile-based architecture, meaning that the system consists of a set of tiles, connected to a two-dimensional grid interconnect. Each tiles implements a communication interface, supporting communication with up to 4 neighbors. As the grid is implemented as a mesh network, tiles are required to forward all packets to their destinations. The SHMAC architecture is depicted in Figure 2.7.

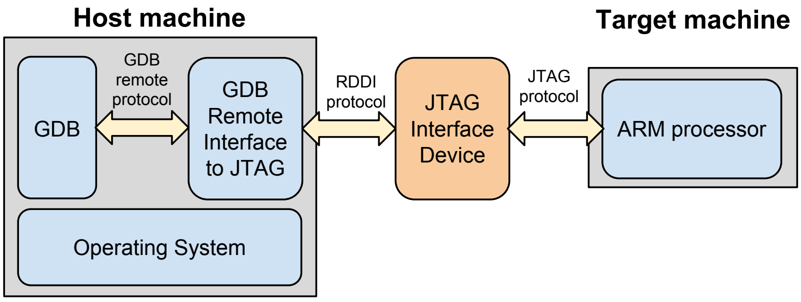


Figure 2.6: Debugging with GDB over the JTAG interface.

The current prototype is synthesized on FPGA development systems. The tile setup on SHMAC can be configured when synthesizing. All configurations consist of at least one processor tile, an APB tile and a memory tile.

The following tiles are available:

APB tile
: This tile provides communication channels to the FPGA host controller. It implements the ARM AMBA Advanced Peripheral Bus (APB) protocol.

Processor tile
: The processor tile consists of a single ARMv3 CPU core.

Main memory tile
: The main memory tile provides access to off-chip RAM. The amount of RAM available depends on the development system.

Scratchpad tile
: The scratchpad tile provides access to block RAM available on the FPGA chip.

2.3.1 Test platforms

There are currently two SHMAC test platforms available. The first test platform is based on a RealView PB11MPCore development system, depicted in Figure 2.8. It features an ARM11-based host controller and 32MB SRAM. The test platform is synthesized with a SHMAC processor consisting of 3 CPU tiles, 1 APB tile, 1 main memory tile and 1 scratchpad tiles.

The second test platform is based on the Versatile Express development system. This system features a larger FPGA chip, making it possible to synthesize SHMAC chips

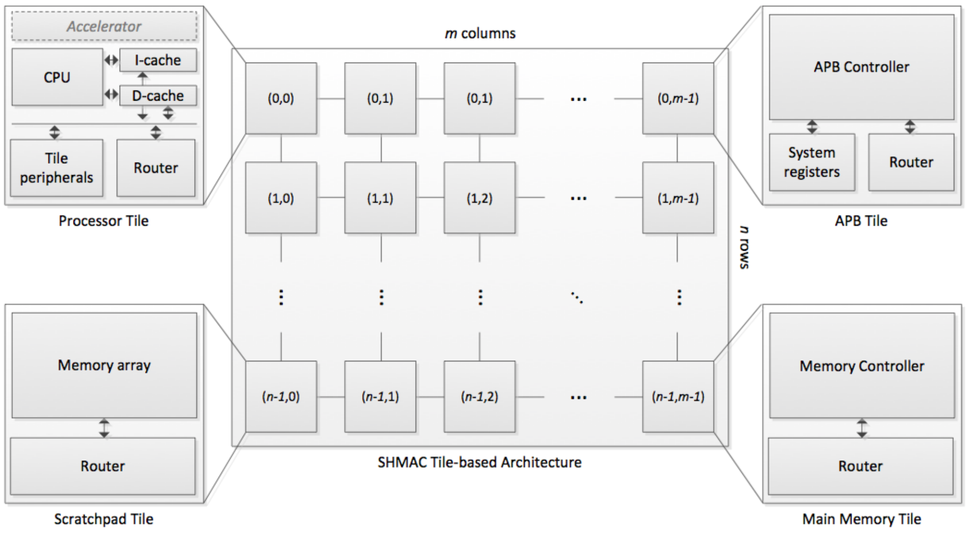


Figure 2.7: Overview of the SHMAC architecture. Source: [EECb].

with several more tiles than the older RealView system. The Versatile system provides 4GB of DDR RAM and features an ARM Cortex A9 quad core host controller. It is synthesized with a SHMAC processor consisting of 8 CPU tiles, 1 APB tile, 1 main memory tile and 1 scratchpad tile.

Both test platforms run a Linux based operating system on their host controllers. The host controller acts as a gateway between the FPGA and the world outside. Communication with the system is available through ethernet connection and RS-232 serial port. Terminal access is possible by connecting to a SSH server installed on the host controller.

SHMAC Communication

The Linux installation communicates with SHMAC using a TTY device driver and set of utility programs. There are a total of 16 TTY devices available, which enable serial communication with the SHMAC. The TTY devices uses the APB tile as its underlying channel.

The utilities are listed in Table 2.2.

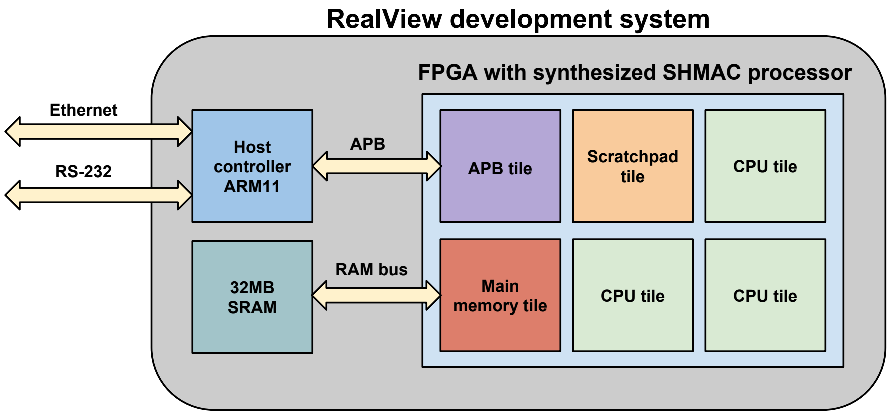


Figure 2.8: SHMAC synthesized on RealView development system.

Table 2.2: SHMAC utility programs

Name	Description
shmac_reset	Resets the SHMAC processor
shmac_program	Writes data to the SHMAC's memory
shmac_dump	Dumps a SHMAC memory region to file

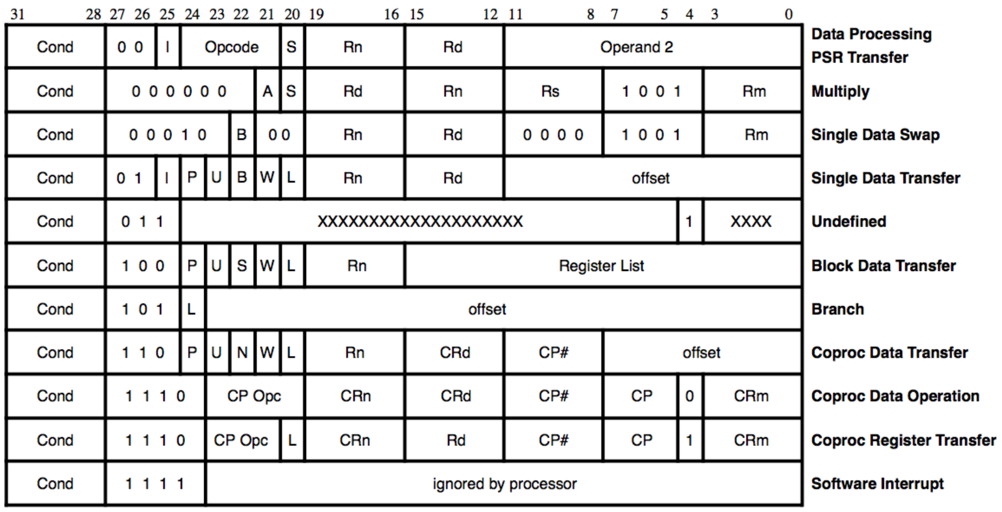
2.3.2 Processor Tile

The processor tile is based on the open-source Amber 25 CPU core [Ope13], which is an 32-bit RISC processor implementing the ARMv2a instruction set. The Amber project is hosted at OpenCores, an open-source hardware community [Ope]. It features a 5-stage pipeline together with separate data and instruction cache.

Amundsen and Andersson [AA14] enhanced the CPU tile to support a newer ARM ISA version during their master thesis. Their goal were to port the Linux 3.12 kernel to SHMAC. As ARMv2a support was removed in Linux 2.6 version, migrating the CPU core to a newer ARM ISA became a necessity. Their work had positive implications for the debugger project, as the ARMv3 ISA was a crucial requirement for some of the debugger functionality. The benefits of ARMv3 will be outlined in subsection 6.3.1.

2.3.3 ARMv3 ISA

The ARMv3 ISA is a 32-bit RISC instruction set. All instructions are 4 bytes long. Only the load and store instructions can access main memory. Every instruction is prefixed with a 4 bit condition code. An instruction is only executed if the condition code agrees with the condition code flags in the program status register. This functionality enables conditional execution without using branching, which is beneficial as branching is an expensive operation. An overview of the instruction set is presented in Figure 2.9.



31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7 5	4	3 0	
Cond	0 0		I	Opcode				S	Rn	Rd	Operand 2				**Data Processing PSR Transfer**
Cond	0 0 0 0 0 0					A	S		Rd	Rn	Rs	1 0 0 1		Rm	**Multiply**
Cond	0 0 0 1 0					B	0 0		Rn	Rd	0 0 0 0	1 0 0 1		Rm	**Single Data Swap**
Cond	0 1		I	P	U	B	W	L	Rn	Rd	offset				**Single Data Transfer**
Cond	0 1 1			XXXXXXXXXXXXXXXXXXXX									1	XXXX	**Undefined**
Cond	1 0 0			P	U	S	W	L	Rn	Register List					**Block Data Transfer**
Cond	1 0 1			L	offset										**Branch**
Cond	1 1 0			P	U	N	W	L	Rn	CRd	CP#	offset			**Coproc Data Transfer**
Cond	1 1 1 0				CP Opc				CRn	CRd	CP#	CP	0	CRm	**Coproc Data Operation**
Cond	1 1 1 0				CP Opc			L	CRn	Rd	CP#	CP	1	CRm	**Coproc Register Transfer**
Cond	1 1 1 1				ignored by processor										**Software Interrupt**

Figure 2.9: Overview of the ARMv3 instruction set. Source: [ARM94].

Modes

The ARMv3 ISA support six modes of operation. The processor may change mode because of several reasons. A mode change takes place when an exception arise, either triggered by an instruction or by an external interrupt. The processor may also be forced into a specific mode by modifying the program status register using the `msr` instruction.

A mode is either privileged or non-privileged. Code running in non-privileged mode is not allowed to execute certain instructions. Some instructions have different semantics in non-privileged mode.

The operating modes are described below:

User mode (usr)

The normal program execution state. It is the only non-privileged mode.

FIQ mode (fiq)
: Entered on fast interrupt. Designed to support a data transfer or channel process.

IRQ mode (irq)
: Entered on normal interrupt. Typically used for general purpose interrupt handling.

Supervisor mode (svc)
: Mode typically used by the operating system code.

Abort mode (abt)
: Entered on prefetch and data abort.

Undefined mode (und)
: Entered when an undefined instruction is executed.

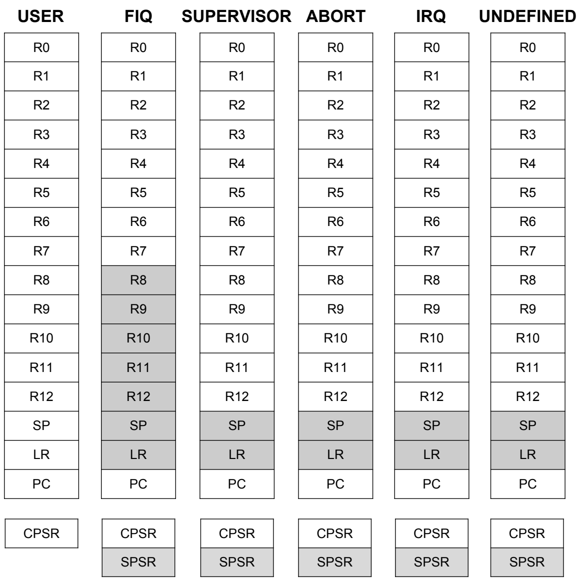


Figure 2.10: Layout of register file.

Registers

ARMv3 has 37 registers, of which 31 are general 32-bit registers and 6 are status registers. Only 16 of the general registers are visible at any time, which 16 depends on the current operating mode. The `CPSR` (Current Program Status Register) register is visible in all modes. All privileged modes has its own `SPSR` (Saved Program Status

Registers) register, used to store the old value of CPSR on mode change. A register that is only visible in a single mode is a *banked register*. Figure 2.10 presents the register layout for each mode. Note that cells with gray background represents *banked registers*.

Some of the general registers have a special purpose. The register R15 is the program counter (PC). The banked register R14, also called LR, is the link register. It receives the old value of PC when executing a branch-and-link register or when an exception handler is invoked. Another special register is R13 (SP), a banked register which stores the stack pointer for each mode.

The layout of the program status registers is presented in Figure 2.11. The condition code flags consists of the N, Z, C and V bits. The bits M0-M4 are the mode bits, which determines the current operating mode. The I and F bits disables IRQ and FIQ respectively when set high.

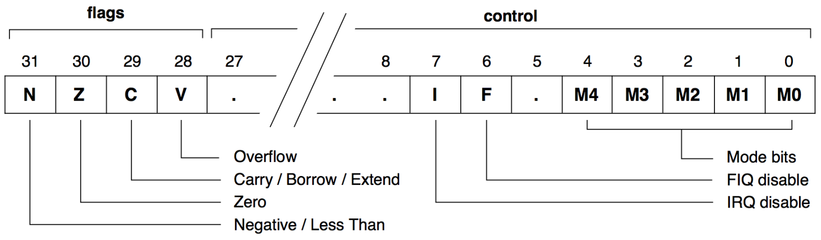


Figure 2.11: Format of Program Status Registers. Source: [ARM94].

Exceptions

Certain registers are backed up automatically on exception. The old PC and CPSR values are copied into the LR and SPSR respectively. PC is set to a location in the interrupt vector defined by the type of exception. The mode bits in CPSR are set to the new mode. Interrupts are disabled when entering the exception handler to prevent unmanageable nesting of exceptions.

The interrupt vector is located at address 0 in memory. It contains 8 entries, one for each type of exception. The layout of the interrupt vector is shown in Table 2.3. An entry is stored as a *branch-and-link instruction*, where the branch offset is the location of the respective interrupt handler.

The CPU starts executing at address 0 on reset, effectively invoking the reset handler.

Table 2.3: The interrupt vector

Address	Exception	Mode on entry
0x00000000	Reset	Supervisor (svc)
0x00000004	Undefined instruction	Undefined (und)
0x00000008	Software interrupt	Supervisor (svc)
0x0000000C	Abort (prefetch)	Abort (abt)
0x00000010	Abort (data)	Abort (abt)
0x00000014	Reserved	-
0x00000018	IRQ	IRQ (irq)
0x0000001C	FIQ	FIQ (fiq)

2.4 Barrelfish

Barrelfish is an experimental operating system developed by ETH Zürich, with collaboration from Microsoft Research [Zü, BBD+09]. Its architecture is designed for future multicore and manycore systems. The motivation behind Barrelfish is the increasing diversity in computer hardware and the growing number of cores. Todays operating systems are designed for the common case, homogenous architectures, while cores in newer systems are getting increasingly diverse. A single system can consist of multiple cores with different performance characteristics, power profiles and ISA. As the number of cores increase, cache coherency becomes more expensive, making shared memory less viable for intercore communication. Barrelfish does not rely on shared memory, and uses solely message passing for communication between cores. It is design to run on multicore system where cores are heterogeneous on ISAs level, for instance systems consisting of both ARM and x86 processors. Most of todays operating systems are designed after the *shared-memory single-kernel model*, where the operating system expects that all cores exposes the same ISA.

Barrelfish is ported to several different architectures. The official distribution supports the most common architectures such as Intel x86 and ARM. Several unofficial ports to more experimental architectures have also been developed. One example is the port to the Intel Single-Chip Cloud Computer [PSMR11], which is an experimental processor consisting of 48 x86 cores [HK10].

2.4.1 The Multikernel Model

Barrelfish's architecture is designed after the multikernel model, which is depicted in Figure 2.12. The multikernel model structures the operating systems as a distributed system of cores.

The multikernel is guided by three design principles:

1. Make all intercore communication explicit.
2. Make OS structure hardware-neutral.
3. View state as replicated instead of shared.

Each *OS node* communicates only through message passing. There is no shared state, each node has its own state replica. Replica consistency is maintained by exchanging messages. The hardware specific parts are separated out from the hardware-neutral codebase, which makes it easier to add support for new architectures.

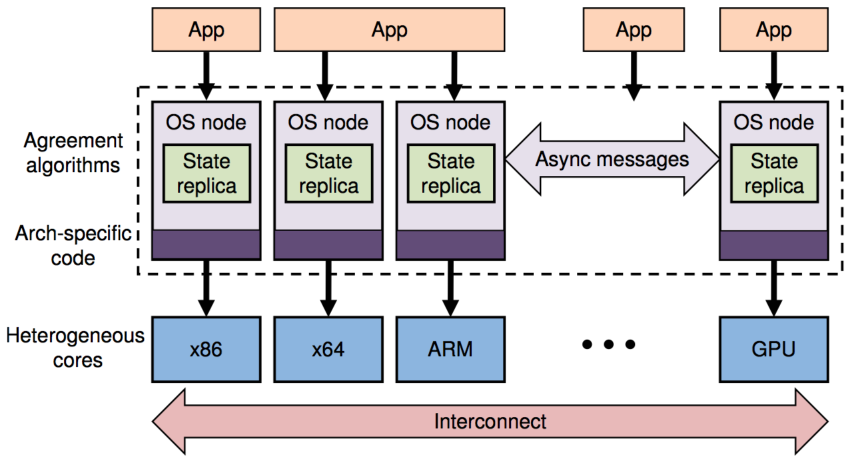


Figure 2.12: The multikernel model. Source: [BBD+09].

2.4.2 Barrelfish Architecture

The Barrelfish implementation of the multikernel model is depicted in Figure 2.13.

Each CPU core has a *CPU driver* running in privileged mode. The CPU driver is responsible managing hardware resources and timeslicing processes, similar to a microkernel. It shares no state with other cores which simplifies its design.

A distinguished user process called *monitor* runs on top of each CPU driver. The *monitors* coordinate systemwide state and updates the data structures inside each local CPU driver, e.g. the memory allocation table, which has to be kept consistent across cores. All intercore coordination is performed by the monitor processes.

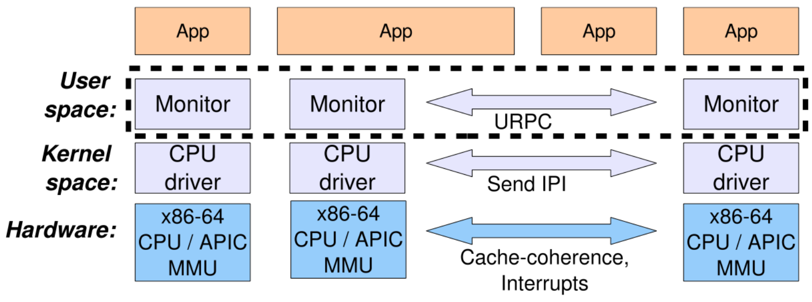


Figure 2.13: The Barrelfish architecture. Source: [BBD+09].

Process Structure

A *process* in Barrelfish is represented by a set of *dispatcher objects*, one object for each core which the process might execute on. Barrelfish does not use kernel threads; the dispatcher objects are the only schedulable units in Barrelfish. Each CPU driver is responsible for scheduling the dispatchers local to the core. Barrelfish provides multithreading capabilities similar to POSIX threads. The thread schedulers on each dispatcher co-operates to create and schedule threads. Threads can be migrated between the dispatchers, hence making it possible to move a thread from one core to another.

2.4.3 The SHMAC Port

Bjørnseth and Seime [BS13] developed a preliminary port of Barrelfish for SHMAC as part of a university project in fall 2014. It represented the first stable operating system for SHMAC. The port is described as preliminary, as it did only utilize a single core. The development of a multicore port based on the previous work was started in early spring 2014.

2.5 Related work

Terje Schjelderup investigated the current debugging capabilities of SHMAC in a report from fall 2013 [Sch13]. Based on the investigation, he described how debugging support could be added to the platform. Implementing a GDB stub was recommended as the best strategy for adding GDB support. The report further gives a rough outline on how the stub could be implemented. The fact that SHMAC did only provide a single TTY channel for communication was emphasized as a limitation of the platform. As a result, the work on implementing multiple channels was initiated by Schjelderup.

Minheng Tan implements a minimal GDB stub for Linux running on Intel x86 [Tan02]. Although the stub is for a desktop computer, the paper outlines the advantages and challenges with developing a stub for embedded devices.

Zheng and Lange implemented GDB debugging support for user program on an operating system called Kitten [ZL]. Kitten is a lightweight OS targeting supercomputers consisting of a large number of nodes. Their solution consists of a GDB stub running as a kernel module in Kitten. The paper describes the challenges related to debugging in a multithreaded environment, which is particularly relevant for implementing a debugger for SHMAC. Barrelfish is similar to Kitten as both are targeting manycore system. Consequently, their paper served as an inspiration for the Barrelfish debugger.

Sidwell et al. describes in a paper how they extended GDB with a new debugging mode [SPA$^+$08]. The new debugging mode, called *non-stop mode*, provides GDB with an alternative way of debugging multithreaded programs. GDB was already able to debug multithreaded programs using the existing *all-stop mode*, but this mode had some limitations. The paper outlines why the new mode is more suitable for debugging under certain conditions. Further, the paper describes the implementation challenges related to asynchronous event handling, single stepping and breakpoints. Hence, the paper provided useful knowledge which was helpful when adding multicore support to the stub.

Chapter 3 GDB for Bare-Metal Programs

Implementing a GDB remote stub was the recommended strategy for adding GDB support to the SHMAC platform, as outlined by Schjelderup in his report [Sch13]. The other alternative, porting the GDBServer application to SHMAC, was less practical as it is designed to run on top of an operating system. Having debugger support independent of any operating system is a big advantage, which enables debugging of any software running on SHMAC.

Integration with GDB gives a lot of functionality for free. The GDB host program will take care of several tasks such as symbol management and user interaction. As a result, implementing a GDB stub requires a relatively small effort compared to implementing a debugger from scratch. GDB itself requires no modifications as it already supports the ARMv3 architecture.

3.1 Specification

The GDB stub should provide the core debugger functionality, such as breakpoints and register inspection. The main focus is to ensure that the core functionality is stable and working as intended. Auxiliary debugger functionality, such as *tracepoints* [1], will only be implemented if time permits and is therefore not listed as a requirement.

Support breakpoints

The debugger should support inserting and removing breakpoints during a debug session. It should also be possible to insert static breakpoints in the program prior to compilation.

[1] A definition of tracepoint, taken from MSDN [MSD]: *A tracepoint is a breakpoint with a custom action associated with it. When a tracepoint is hit, the debugger performs the specified tracepoint action instead of, or in addition to, breaking program execution. One common use for tracepoints is printing a message when your program reaches a certain point.*

Support GDB interrupts

The debugger should support the interrupt mechanism described in section 2.2.3. Although not a requirement in the remote protocol specification, it is a highly convenient functionality. The feature enables the user to halt the program at any time so its current state can be inspected.

Debug code running in any mode

The debugger should be able to debug code running in any privileged mode in addition to user mode. Most system calls in an operating system are executed in supervisor mode. Exception handlers are executed in several different privileged modes. Being able to debug both the system calls and exception handlers are important for the usability of the debugger.

Debug programs running on multiple cores

SHMAC is designed for manycore computing. Only being able to debug a single core would obviously be a major limitation. This functionality is therefore highly prioritized, though it is a technical challenge to implement it correctly.

No dependencies on operating systems or external libraries

The debugger should not require the presence of an operating system or any external libraries. Not even the C standard library (*libc*) may be used, as it requires that the system provides implementation of certain system calls. The debugger cannot rely on the target program providing the system calls. Neither can the debugger provide its own implementation, as it would interfere with any system calls implemented by the target program. As a result, the debugger implementation may only be written in pure C (no use of standard library) and ARM assembly.

Simple to integrate into SHMAC programs

Integrating the debugger should only require a small amount of modification to the original program. The modifications should be easy to perform and take minimal effort. It should also be easy to disable the debugger once integrated.

Adequate performance

The debugger should provide reasonable response time for common operations. Long response time worsen the debugging experience, and may result in GDB timing out in some situations. Performance is more of a secondary objective. Any performance optimization should only be applied if it provides a significant improvement. Simple operations such a single stepping should not take more than 3 seconds. We do not require a hard limit for all operations, as some GDB commands are complex in nature and will not be able to complete in 3 seconds. Large memory transfer operations are for instance limited by the bandwidth of the TTY channel, and may take several seconds to complete.

3.2 Existing Work

A GDB stub for the SPARC architecture is part of the GDB codebase. The GDB documentation [RSea14, chapter 20.5] recommends this stub as a starting point for new stub implementations, as its codebase is claimed to be the best organized among the official stubs. This stub did only implement the minimal command set listed in Table 2.1. More advanced protocol functionality such as packet compression was missing. The stub was primarily consisting of target specific code. Its codebase was therefore used more as inspiration than a basis for the SHMAC stub. The part of the code encoding and decoding the protocol packets were of most interest. Having that code available speeded up the initial development.

3.3 Development Process

The development of the stub was an iterative process. The first prototype did only support the minimal set of commands required, and could only debug a program running on a single core. The command set was then extended to support important functionality such as breakpoints. Once the stub was stable enough and supported the most useful commands, the process of making a Barrelfish version was initiated. Once the initial Barrelfish version was finished, the work on enabling multicore debugging started. The GDB stub had to be modified so that multiple cores could be debugged simultaneously. The multicore support was later backported to the Barrelfish debugger.

3.4 Debugger Setup

Figure 3.1 presents an overview of the complete debugger setup for SHMAC. The GDB stub is linked into the target program. It communicates with the host controller through the TTY interface. The Linux installation on the host controller runs a program called `ser2net` [Sou]. `ser2net` is a serial port to network proxy which maps each TTY device to a TCP/IP socket. Connected to the `ser2net` is a GDB instance running on a host machine. The host machine can be any PC which has GDB with ARM support installed. All communication between GDB and the stub is forwarded by the `ser2net` process. The host controller could technically run GDB itself, eliminating the need for a host machine and `ser2net`. The reason for not doing so is because of its poor CPU performance. The GDB program is a resource heavy program that runs notably slower on a low-performance ARM processor.

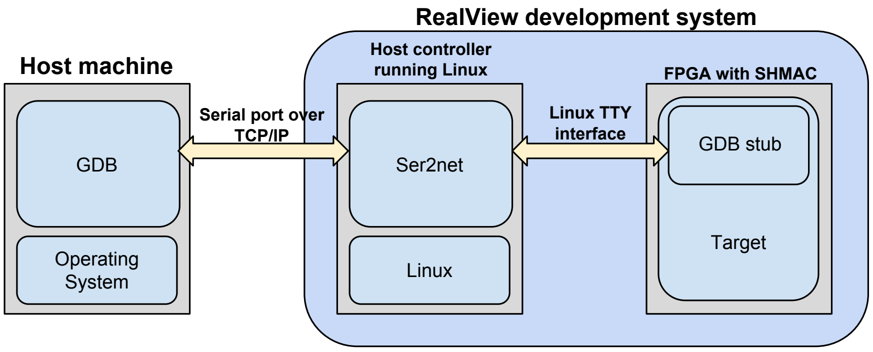


Figure 3.1: Overview of the debugging setup for SHMAC.

3.5 Implementation

3.5.1 Stub Overview

Figure 3.3 presents an overview of the GDB stub together with the target program. It shows that the *interrupt vector* has an important role in the interaction between the program and the stub. The blue boxes represent important software routines in the stub and program, and the arrows explain how they are connected together. Figure 3.2 demonstrates how the interrupt vector would look like for a program without the GDB stub. Note that both figures do not show how all entries in the interrupt vector are configured. Some entries, like the `software interrupt` entry, are ignored. Those entries are not part of the debugger integration and will, if configured, point to handler routines in the target program.

The interrupt vector is configured such that `undefined instruction exceptions` and `IRQ interrupts` are handled by the GDB stub. The `GDB exception handler` handles the undefined instruction exceptions and the `GDB IRQ handler` handles the IRQ interrupts. Both GDB stub handlers are connected to the respective handlers in the target program. Any undefined instruction exception or IRQ interrupt not relevant for the debugger is forwarded to the program's respective handlers.

The *reset handler* is responsible for initializing the C runtime environment. The most important task is to configure the program's stack pointers. A separate stack is created for each operating mode. The last task is to invoke the program's `main` function. The initializing procedure receives an extra responsibility when using the debugger; it has to initialize the GDB stub. The GDB stub provides a single initialization function that is invoked either at the top of the `main` function or in the reset handler itself.

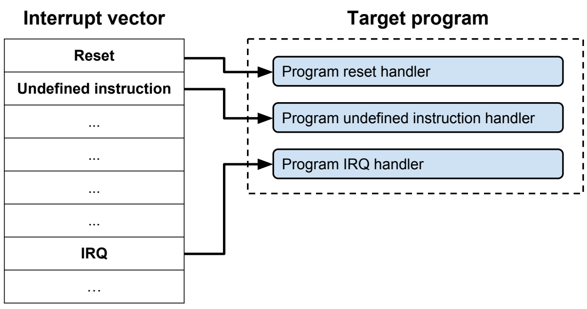


Figure 3.2: Interrupt vector configuration without GDB stub

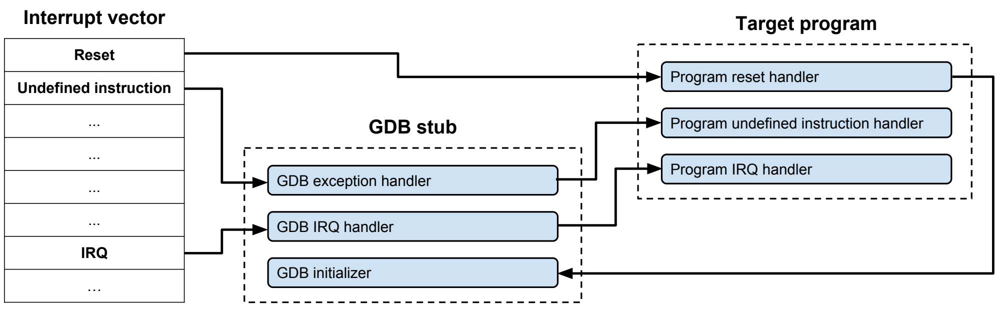


Figure 3.3: Interrupt vector configuration with GDB stub

3.5.2 Debugger Initialization

The debugger initialization is performed in two steps. The first step is to initialize the data structures managed by the stub. This is achieved by calling the function `gdb_init` as described in subsection 3.5.1. It is important to call the function as early as possible, as it is not possible to debug the program prior to initializing the stub. `gdb_init` has the following signature:

```
typedef struct {
    int tty_channel[NUM_CORES];
    void (*kill_program_handler) (void);
} gdb_config_t;

void gdb_init(gdb_config_t * config);
```

It takes a configuration struct as its only argument. This struct has two fields:

– One array declaring which TTY channels the debugger should use. This configuration is core specific; index 0 represents CPU 0, index 1 represents CPU 1 and so forth. Setting a value to -1 will disable debugging for that core.

– A function pointer to a *kill* function. This function is invoked when the debugger kills the target program.

An example of a configuration is shown below. Core #0 uses channel 1 as debug channel and core #1 uses channel 2 as debug channel. Core #2 has debugging disabled as its TTY channel value is -1.

```
gdb_config_t gdb_config;
gdb_config.tty_channel[0] = 1;
gdb_config.tty_channel[1] = 2;
gdb_config.tty_channel[2] = -1;
gdb_config.kill_program_handler = &die;
gdb_init(&gdb_config);
```

The second step is to insert a static breakpoint in the program somewhere after the call to `gdb_init`. The static breakpoint is required as `gdb_init` does not halt the program. It is up to the user to decide where to put the breakpoint. This will usually be the statement right after the call to `gdb_init`, so that the program halts as early as possible. The GDB host will only be able to connect to the stub after the program has been halted.

3.5.3 Exception Handler

The *GDB exception handler* handles all program exceptions relevant for the debugger. It is invoked when any of the following events occur:

– The program hits a breakpoint.

– The program has executed a single instruction during single stepping.

– The GDB stub has received a GDB interrupt signal.

The details on how those events are detected and handled will be described later in this chapter.

The implementation of the exception handler can be separated into two parts: a *undefined instruction handler* which is written in assembly, and a C function called `gdb_exception_handler`. The assembly routine is responsible for operations that cannot be performed in C code, such as storing register values to memory. The list below describes the sequence of operations performed by the assembly handler:

1. All *non-banked registers* (R0-R12, PC and CPSR) are stored to memory.
2. The previous operating mode is determined by reading SPSR. The CPU is then forced to the previous operating mode.
3. The *banked registers* (SP, LR and SPSR) of the previous mode are stored to memory.
4. The CPU is switched back to *und* mode.
5. gdb_exception_handler is called. A pointer to memory region containing the register values is passed as argument.
6. All *non-banked registers* are restored.
7. The CPU is forced to the previous mode.
8. The *banked registers* are restored.
9. The CPU is switched back to *und* mode.
10. The program's undefined instruction handler is invoked if the return value from gdb_exception_handler is 1.

The register values stored to memory represents the register content at the time the program halts. A pointer to the register storage is passed to the C handler. The stub may read or modify the register content as part of its operation. Note that the C handler returns a boolean value, which determines if the exception should be passed to the program's handler.

The gdb_exception_handler is more complex than the assembly handler. Algorithm 3.1 presents a greatly simplified version of gdb_exception_handler to outline its most important tasks.

Interrupts are automatically disabled by the processor when an exception occur. The C handler re-enables interrupts, so data from the GDB host can be received. This involves modifying the interrupt mask for the interrupt controllers on the CPU core. Host interrupts for the debug channel is enabled, while all other types of interrupts are disabled. The interrupt mask is restored when gdb_exception_handler returns.

Once interrupts are enabled, a single *stop reply packet* is transmitted to the host to notify that the program has halted. The function will then enter the message handling loop. Each iteration of the loop starts by waiting for a packet from the host. Once received, the packet is decoded, a specific operation is performed and a response is returned. Some of the packets, like the *continue* command, will make the `gdb_exception_handler` exit the loop and return.

The instruction cache is flushed right before leaving the exception handler. The reason is that the stub might have altered the instructions of the program, and the instruction cache will in those cases contain stale data.

Algorithm 3.1 Simplified version of `gdb_exception_handler`.

```
function GDB_EXCEPTION_HANDLER(register values)
    BACKUP_INTERRUPT_MASK()
    ENABLE_HOST_INTERRUPTS()
    SEND_STOP_REPLY_PACKET()
    while true do
        packet ← READ_PACKET()
        switch PACKET_TYPE(packet) do
            case 'p'                                  ▷ Read-all-registers packet
                ...
            case 'm'                                  ▷ Read-memory packet
                ...
            case 'c'                                  ▷ Continue packet
                ...
                return PREPARE_RETURN()               ▷ Continue execution of program
            ...
    end while
end function
function PREPARE_RETURN()
    DISABLE_INTERRUPTS()
    RESTORE_INTERRUPT_MASK()
    FLUSH_INSTRUCTION_CACHE()
    return SHOULD_FORWARD_EXCEPTION()
end function
```

3.5.4 Host Communication

The stub utilizes the TTY interface for communication with the GDB host. SHMAC provides 16 TTY channels. Which channels to use as debug channels is determined by the configuration struct described in subsection 3.5.2. The stub will handle all host interrupts coming from the debug channels. All characters received are inserted into a ring buffer. This data structure provides a read function to remove characters from the ring. The read function is used by the `read_packet` function shown in Algorithm 3.1. The ring buffer has a fixed size. If the buffer is full when a character is received, the oldest character is dropped.

3.5.5 Breakpoints

As the CPU core does not provide hardware breakpoints, breakpoint support had to be implemented entirely in software. The remote protocol defines two packets for this purpose. The packet `Z0` is used to insert a software breakpoint and the packet `z0` is used to remove a software breakpoint. GDB assumes that the software breakpoints are implemented by replacing the instruction at a given location with a trap instruction. Support for handling both packets was added to the stub.

In addition to supporting the GDB breakpoint packets, support for *static breakpoints* was also implemented. Static breakpoints uses a trap instruction to halt the program, just like standard breakpoints. A static breakpoint is inserted by placing a C preprocessor macro in the source code of the program. As it is compiled with the program, it is *static* and may therefore not be removed at runtime.

Selecting a Trap Instruction

A series of suitable trap instructions had to be selected for use as breakpoints. The ARMv3 provided major 2 alternatives; the software interrupt instruction (`swi`) or an undefined instruction. The latter was selected as the best candidate for several reasons:

– Software interrupts will trap the processor to *svc* mode, while a undefined instruction will trap to *und* mode. When the CPU enters an exception handler, it will copy the old value of `PC` into `LR`. If the previous mode is the same as the mode used by the exception handler, the old value of `LR` will be lost. Same problem also applies for the `SPSR` register. The implication is that a `swi` instruction cannot be safely executed in *svc* mode, neither may an undefined instruction be safely executed in *und* mode. As most of the operating system code is executed in *svc* mode, using `swi` as breakpoint instruction is a non-viable choice.

– Software interrupts are used in operating systems when invoking a system call. If a `swi` instruction was used for breakpoints, the debugger would need to have a dedicated mechanism for distinguishing breakpoints from system calls. Undefined instructions may also be managed by an operating system, but will typically have a less important role.

A series of `mcr` (coprocessor register transfer) instructions were selected as breakpoint instructions. A `mcr` instruction will generate an undefined instruction exception if it refers to a non-existing coprocessor. This property is the reason why the `mcr` instruction was chosen. Specific instructions are used for specific purposes. For example, static breakpoints and standard breakpoints do not use the same `mcr` instruction.

Breakpoint Insertion

This section describes the steps performed by the GDB stub when it receives a `Z0` packet. The packet has one argument; a memory address where the breakpoint should be inserted.

1. The existing instruction at the memory location is inspected. The breakpoint insertion procedure returns immediately if the instruction is a breakpoint instruction. This situation may occur if a breakpoint is already inserted at the location.
2. The memory address and the existing instruction at that location is stored to a data structure. This information is needed when removing the breakpoint later on.
3. Lastly, the breakpoint instruction is written to the memory location.

Breakpoint Removal

This section describes the steps performed by the GDB stub when it receives a `z0` packet. The packet has one argument, a memory address of an already inserted breakpoint.

1. The existing instruction at the memory location is inspected. The breakpoint removal procedure returns immediately if the instruction is not a breakpoint.
2. The stub checks if there is an entry in the breakpoint data structure referring to the given memory location. If so, the original instruction is retrieved.
3. The original instruction is written back to the program. The breakpoint information is then removed from the data structure.

Handling a Breakpoint Exception

The list below describes what happens when the program hits a breakpoint:

1. The GDB exception handler is invoked.
2. The instruction pointed by the PC register is inspected to confirm that the program halted because of a breakpoint.

3. A *stop reply packet* is sent to the host to notify that the program has halted.

Continuing the program is not straightforward. The original instruction at the breakpoint location has to be executed, but to do so, the breakpoint has to be removed. After executing the instruction, the breakpoint has to be reinserted.

The breakpoint is removed as described in section 3.5.5. The original instruction is then single stepped. Further, the breakpoint is reinserted using the steps described in section 3.5.5. Lastly, the program is continued.

The procedure is simpler for *static breakpoints*. The stub just has to ensure that the program continues execution on the instruction following the breakpoint, essentially incrementing the `PC` register before returning from the GDB exception handler.

3.5.6 Single Stepping

The remote protocol requires that the stub implements the `S` (single step) packet. When the stub receives this packet, the program will execute a single instruction and then halt. The stub then replies the host with a stop reply packet. The packet has an optional argument, the address at which to resume execution.

Design Alternatives

The CPU core does not provide hardware support for single stepping, so similarly to breakpoints, single stepping had to be implemented in software.

There are two major design choices for single stepping, both briefly described in [SPA$^+$08]:

– Simulate the behaviour of the stepped instruction inside the debugger.

– Insert a temporary breakpoint at the next instruction after the one being single stepped.

The first choice is the most complicated solution. The debugger must be able to decode any valid instruction and simulate its behaviour, which may include modifying registers and writing to main memory. The task is essentially to build a complete instruction simulator for ARMv3. Another approach to instruction simulation is to execute a displaced copy of the instruction. The latter approach is relatively easy for all instructions that do not use `PC`. For instructions modifying `PC`, simulation is required.

Table 3.1: Instructions supported by the single step instruction decoder.

Instruction	Description
`add`	Add
`b`	Branch
`bl`	Branch and link
`ldm`	Load multiple registers
`ldr`	Load single register
`mov`	Write to register
`sub`	Subtract
`swi`	Software interrupt

The second choice is simpler. The debugger inserts a breakpoint at the next instruction after the one being single stepped. This is trivial for instructions that do not modify the `PC` register: the next instruction is at location `PC + 4`. It is more complicated for instructions modifying `PC`. Those instructions have be decoded, and the effect on the `PC` register simulated to reveal the location of the next instruction.

Implementation

The second alternative described in section 3.5.6 was chosen, as it required a less complex solution.

Support for determining the next value of `PC` was added for the most common instructions generated by the *GCC* compiler. Though many instructions are technically allowed to modify `PC`, the compiler only use a small set of instructions for `PC` modification. A binary program compiled for SHMAC was decompiled to determine which instructions were commonly used. Based on the analysis, a subset of instructions was selected. These instructions are listed in Table 3.1. The instructions come in many variants, for instance some operands might either be a register or an immediate value. Support for decoding the most common variants was prioritized, while rare variants were left out.

Instruction Decoding

1. The first step is to examine the 4-bit condition code. The condition code is evaluated against the condition code flags in the status register to determine if the instruction will be executed. If that's the case, the next step is performed.

2. The second step is to determine if the instruction type is supported by the decoder. Certain bits from the instruction are extracted and compared to

determine the type.

3. The third step is to determine the destination register (if any). The fourth step is only performed if `PC` is set as destination register, or if the instruction implicitly modifies `PC`.

4. The fourth step is to calculate the new value of `PC`. Any operands and flags are extracted from the instruction. The evaluation of the operands may involve reading register values and or content from main memory. The new value is calculated based instruction type, operands and instruction flags.

Performing a Single Step

A temporary breakpoint is inserted at the location pointed to by the next `PC` value. The original instruction is saved to a data structure prior to insertion, similar to how insertion of a normal breakpoint is handled. When the program continues, it will execute the single instruction and then hit the temporary breakpoint. The stub will then restore the original instruction.

3.5.7 GDB Interrupts

The stub's IRQ handler handles the data received from host, and is therefore also responsible for detecting GDB interrupts. As described in section 2.2.3, an GDB interrupt is transmitted as the byte `0x03`. The byte `0x03` does only represent an interrupt if not part of packet data. As a result, the IRQ handler has to implement a finite state machine to be able to detect if a `0x03` is an interrupt or not. The FSM is depicted in Figure 3.4.

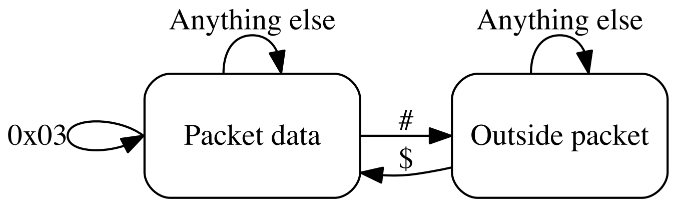


Figure 3.4: The finite state machine for GDB IRQ handler.

The GDB exception handler is not invoked directly from the interrupt handler. Instead, the program is interrupted using a temporary breakpoint. The interrupt handler knows the value of the program's `PC` register. This is the location where the temporary breakpoint is inserted. Once the interrupt handler returns, the program start executing the instruction at `PC` and will then immediately halt.

A breakpoint is not inserted if the CPU was in `und` mode at the time of IRQ interrupt. The reason is that it is very likely that the code running at that moment would be the debugger itself.

3.5.8 Disabling the Debugger

The debugger can be disabled at compile time by defining the pre-processor macro `DISABLE_DEBUGGER`. The macro has the following effect on the GDB stub:

- Static breakpoints behave like a no-op.
- The stub's IRQ handler and undefined instruction handler will forward all interrupts to the program's handlers.
- The GDB initialization function does nothing.

3.6 Performance Enhancements

Several performance enhancements were added to the GDB stub. The SHMAC CPU core has limited performance; it is clocked at only 60MHz, with both data and instruction cache disabled due to a hardware defect. With such performance limitation, extra precautions were made to ensure that common GDB operations performed well. The focus was on reducing the packet traffic required for typical GDB commands as each packet has a high cost in term of processing time and network delay. The optimizations reduces the number of packets transmitted and the size per packet. The sections below describe the optimizations implemented.

3.6.1 Range Stepping

GDB 7.7 added a feature called target-assigned range stepping [GDB14]. Range stepping is a variant of single stepping where the stub steps over multiple instructions as long as `PC` is within a specified range. The remote protocol is extended with a new packet for commanding the stub to perform a range step. The range is given as two memory addresses, a lower bound and an upper bound. The program will continue execution as long as the program counter is in the range. Once the program counter is outside the range, the program halts and the stub returns a *stop reply packet* to the host.

Range stepping is particularly useful when stepping over a single line of code, which only requires a single range step packet. GDB versions prior to 7.7 had to repeatedly issue single step commands to the stub to step over a single line, once for each instruction constituting the line.

Range stepping brought a substantial reduction in packet traffic, especially for stepping over code lines consisting of a large amount of instructions. A line consisting of 100 instructions would result in 100 single step packets and 100 *stop reply packets*.

Range stepping was implemented using single stepping. When a range step packet is received, a range step flag is set. The stub will then single step instructions as long as the range step flag is set and `PC` is in the provided address range. Once `PC` is outside the range, the flag is cleared, and the stub reports back to the host that the range step is complete. Figure 3.5 illustrates the range stepping mode as an finite state machine.

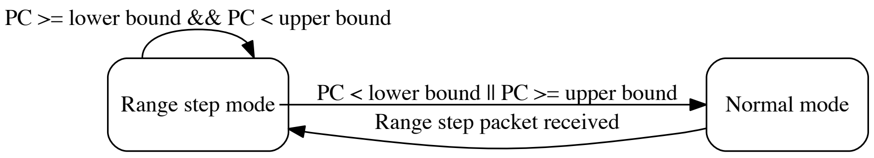

PC >= lower bound && PC < upper bound
Range step mode
PC < lower bound || PC >= upper bound
Range step packet received
Normal mode

Figure 3.5: Range stepping finite state machine.

3.6.2 Packet Compression

The remote protocol allows the remote to compress packet data in response packets. A technique called *run-length encoding* is used as compression algorithm. Sequences of identical characters are stored as a single character and a count. The compression is particularly useful when reading larger blocks of memory from host, as large sequences of zeroes are common.

3.6.3 Binary Transfer Packet

Binary data in most packets is encoded as two hexadecimal digits per byte of binary data. This allowed the traditional remote protocol to work over seven-bit connections [RSea14, p. 575]. As this encoding is wasting bandwidth on the eight-bit connections, such as the TTY channels on SHMAC, hexadecimal encoding is not used in some newer packets. One example is the `X` (binary download) packet, which is used for writing content to the remote's memory [RSea14, p. 584]. The content is transferred as raw binary data. This packet uses half as many bytes for the data part as the `M` packet, as the hexadecimal encoding encodes each byte as two bytes.

3.6.4 Disabling Irrelevant Packets

GDB queries the remote target for certain types of information after certain events, for instance when receiving a *stop reply packet* or when connecting to a target. The information is queried using query packets. The query packets are generally optional and may not be relevant for the remote target. These packets waste bandwidth if the remote does not support them. The GDB host can be configured to not send specific query packets.

Disabling the `qTStatus` packet saved a single round-trip time for every breakpoint hit or instruction single stepped. The packet is used to query the status of all trace experiments running. This packet is meaningless for SHMAC as support for *tracepoints* is not implemented, and therefore no trace experiment may exist.

3.6.5 Extended Stop Reply

A *stop reply packet* is sent whenever the target halts. The simplest stop reply packet only returns the signal explaining the reason for the halt. The remote protocol defines several other versions which provide the host with more relevant information. Bundling extra, relevant information in addition to the stop signal may save the host for querying the information afterwards. GDB needs the `PC` value to determine the location where program halted. By bundling `PC` and other special purpose registers, the host will not need to explicitly fetch their values later on. For this reason, the decision to use an extended stop reply was taken. The stop reply is implemented such that it reports the register values for `R11`, `R12`, `SP`, `LR`, `PC` and `CPSR`. These values eliminated the need for GDB to read out register values using the read register packet in many situations, such when the program halts after hitting a breakpoint.

3.7 Multicore support

3.7.1 Analysis of Implementation Alternatives

Adding multicore support to the debugger was a highly prioritized goal. The stub could only debug a single core initially. It was extended to enable multicore debugging later in the implementation phase.

The analysis revealed three different alternatives for multicore debugging. The two first alternatives relied on the thread abstraction in GDB described in section 2.2.3. *all-stop mode* represents the first alternative, and *non-stop mode* represents the second alternative. They provide different semantics, *all-stop mode* provides a simpler abstraction then *non-stop mode.*

The third and last alternative is fundamentally different, as it does not rely on a thread abstraction. Instead, each core is debugged individually using a separate GDB instance per core. This alternative gave the best trade-offs and the resulting solution uses this approach.

The three alternatives are discussed in more detail in the sections below.

All-stop Mode and Non-stop Mode

The major difference between the modes is that in *all-stop mode*, all threads halt whenever one thread halts. This allows the user to examine the overall state of the program without worrying about the other threads. This is in contrast to *non-stop mode*, where each thread may be halted and continued individually. Consequently, adding support for *non-stop mode* would result in a more complex implementation than *all-stop mode.*

The major advantage with both modes is that it provides a the user with a easier abstraction than using a separate GDB instance for per core. *Non-stop mode* is particularly useful when debugging real-time system where halting all threads might not be desired due to timing constraints.

The thread abstraction was not selected due to technical limitations with the SHMAC platform. It requires that SHMAC provides a reliable mechanism such that one core may halt the execution of the other cores. A somewhat similar mechanism was added in the third month of this project. It enabled a single core to send a soft interrupt to the other cores. This mechanism was not deemed good enough as it relied on interrupts, which would likely interfere with the program's own use of interrupts. The program might use global interrupts for its own purposes, and the program might in some situations disable all interrupts.

Both modes expects that the threads are part of the same program. As a result, the thread abstraction does not fit well in situations were some of the cores execute separate programs.

One GDB Instance per Core

In this alternative, each CPU core is debugged by a separate GDB instance. Debugging a program running on 4 cores will require that the host connects 4 separate GDB instances to the remote. The host observes the remote as 4 separate targets. Each core is debugged individually as independent targets.

One of the drawbacks with this design is the fact that the number of cores that may be debugged simultaneously is limited to the number of TTY channels available, which is 16 if the program is using none of them. Non-stop mode and all-stop mode does not have a thread count limit.

Nevertheless, this alternative has a two major advantages. First of all, it is a simpler design. The main reason is the fact that there is no need for any global coordination between the cores. This is important as SHMAC does not provide any suitable mechanism for inter-core coordination. Secondly, it is a better alternative for debugging software where each core is executing different programs or operate more independently of each other. A good example is Barrelfish, where each core runs a separate CPU driver instance.

3.7.2 Implementation

This section described the various modifications performed on the debugger to enable multicore debugging. The selected alternative required that multiple GDB instances should be able to connect to the same stub. The modifications mostly involved duplicating data structures and usage of synchronization primitives to protect critical regions.

The CPU cores share a single interrupt vector, located at address 0 in main memory. Exceptions and interrupts generated by any of the cores will be handled by interrupt handlers shared among the cores. In other words, all cores will share a single GDB exception handler instead of having its own private copy. Though the GDB stub itself is shared, every core has its own set of data structures. This effect was achieved by placing the data structures in arrays, indexed by core id.

The technique is illustrated by the following example. The data structures storing breakpoints were represented as two static variables in the original single core stub:

```c
static breakpoint_t breakpoints[MAX_BREAKPOINTS];
static breakpoint_t step_breakpoint;
```

The code below shows how the data structures are defined in the multicore version:

```c
typedef volatile struct {
    breakpoint_t breakpoints[MAX_BREAKPOINTS];
    breakpoint_t step_breakpoint;
} core_state_t;
static core_state_t state[NUM_CORES];
```

The two static variables are moved into a struct called `core_state_t`. This struct contains all data private to each core. The static variable `state` is an array of `core_state_t` with the size equal to the number of CPU cores in the system.

Debugger Initialization

Core #0 is responsible for initializing the stub. The initialization is performed as described in subsection 3.5.2, which is to call the `gdb_init` function and insert a static breakpoint. The debugger must be initialized prior to starting the other CPU cores. In that way, none of the other cores may hit a static breakpoint before the stub is properly initialized.

Breakpoints

Each core manages its own breakpoints; they can insert and remove breakpoints without consulting the other cores. Access to the functions manipulating breakpoints are mutually exclusive. Each function acquires a single global lock on entry. The same lock is released on return. The use of locks ensures that all breakpoint operations are atomic to protect against race conditions. Two debugger instances should for instance not be able to insert a breakpoint at the same position simultaneously as this may result in code corruption. The issue is demonstrated through the following scenario:

Assume there are two instances trying to insert a breakpoint on the memory location. Instance #1 may backup the just inserted breakpoint instruction of instance #2 prior to inserting the breakpoint instruction itself. If instance #1 removes the breakpoint after instance #2, the original instruction will be lost.

Host Communication

Each core uses a single TTY channel for host communication and each channel uses a separate ring buffer where characters received from the host are stored. There is no need for synchronization as none of the TTY channels are shared among the cores.

Chapter 4 GDB for Barrelfish

4.1 Motivation

Integrating the debugger with Barrelfish provided a great opportunity to test the debugger on a larger software project. Experience during the Barrelfish integration gave valuable feedback on the debugger functionality. For instance, debugging an IRQ interrupt handler was recognized as an important feature during testing in Barrelfish, and this feature was therefore added to the debugger later on.

Barrelfish was the first stable operating system for SHMAC. It represented an important milestone for SHMAC, as having an operating system simplifies software development to the platform. Adding debugging support to Barrelfish would make it even more attractive as an operating system for SHMAC.

4.2 Specification

GDB for Barrelfish was planned to adhere to the following requirements:

Kernel debugging
The debugger should be able to debug the Barrelfish CPU driver. It should be possible to debug system processes that are part of Barrelfish, for instance the processes *init* and *mem_serv*.

Debug user programs
The debugger should be able to debug user programs.

Debug multiple cores simultaneously
The debugger should be able to debug multiple CPU cores simultaneously. It should be possible to debug the CPU driver running on each CPU core individually.

Provide debug symbols
The stub should help GDB provide debug symbols for the CPU driver and processes running on Barrelfish.

4.3 Implementation

4.3.1 Overview

The debugger support for Barrelfish is provided by a modified version of the GDB stub for SHMAC. As the stub had to be tightly integrated with Barrelfish, the stub's source code was placed directly in the Barrelfish CPU driver.

As the CPU driver image is loaded to memory each time a new core is launched, one stub instance exists per launched core. Only the GDB stub in the CPU driver image of core #0 is active, the others are not used. All undefined instruction exceptions and IRQ interrupts generated on any core are handled by the stub of core #0, similar to how exceptions are handled in the standard GDB stub for SHMAC. Figure 4.1 presents an overview of the GDB stub in Barrelfish.

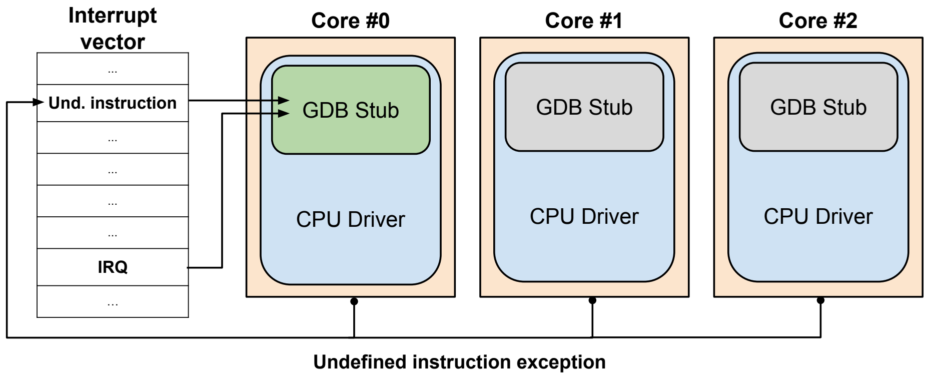


Figure 4.1: Overview of the GDB setup in Barrelfish.

4.3.2 Multiprocess Mode

The stub implements the multiprocess protocol extension required for proper multi-process mode in GDB. The extension affects the syntax of several packets. Process id (*pid*) is added as a mandatory argument for several packets, like the stop reply packets. The pid is required for GDB to detect which process is halted and to perform process specific operations.

Additionally, extra packets for attaching, detaching and other process operations were implemented. The packet `vAttach` is used to attach to a process. It takes one argument, a pid. Similarly, the packet D is used to detach a process, also taking a pid as argument.

GDB expects that the stub is already attached to a process when initiating the connections. The GDB connection is initiated at the start of the Barrelfish CPU driver. Consequently, there exist no process to attach to at that moment. The issue is resolved by pretending that the CPU driver is a process with id #0.

The debugger may only be attached to a single process at a time. GDB will explicitly detach the existing process by sending a detach packet prior to attaching a new process.

4.3.3 Barrelfish Hooks

Several hooks are inserted into Barrelfish to notify the stub when certain OS events occur. The hooks can be divided into two categories; hooks related to initiating the Barrelfish CPU driver on a CPU core, and hooks related to process life cycle events. These hooks will be described in more detail in the sections below.

The hooks are created by modifying the source code of Barrelfish. Different mechanism is used to report events depending on where the event occurred. If an event is reported from the CPU driver of core #0, a direct function call to the stub is used. This mechanism is used as the active GDB stub is the one linked to CPU driver at core #0.

For the same reason, this mechanism cannot be used if reporting from a CPU driver on a different core, or if the event is reported from a process. A different mechanism, similar to how system calls work in operating systems, is used in those situations. This mechanism is described in the next section.

The System Call Mechanism

This section describes the system call mechanism used in Barrelfish to notify the stub about relevant events.

The first step of this mechanism is to invoke the function `invoke_gdb_syscall`, passing an integer representing the event type and a pointer to a struct containing any extra arguments:

```
process_created_params_t gdb_params;
...
invoke_gdb_syscall(PROCESS_CREATED, (void *) &gdb_params);
```

The definition of `invoke_gdb_syscall` is simple, it consists of an undefined instruction and a return instruction. The undefined instruction generates a trap which is handled by *GDB exception handler*.

```
invoke_gdb_syscall:
    mcr p1, 6, R13, c9, c4, 5
    mov pc, lr
```

The stub checks if the program halted because of a system call by examining the `PC` register. The cause of halt is a system call if the instruction pointed by `PC` is a specific undefined instruction.

```
if (is_gdb_syscall(registers)) {
    handle_gdb_syscall(registers);
    return prepare_return(...);
}

static bool is_gdb_syscall(uint32_t * registers) {
    uint32_t * pc = (uint32_t *) registers[R15];
    return *pc == GDB_SYSCALL_INTERRUPT_INSTR;
}
```

The C calling convention for ARM guarantees that the two arguments of `invoke_gdb_syscall` are located at register `R0` and `R1` respectively. The stub performs a specific operation based of the content of these two registers. The implementation of the function is shown in Figure 4.2.

```
static void handle_gdb_syscall(uint32_t * registers) {
    syscall_ident_t type = (syscall_ident_t) registers[R0];
    switch (type) {
        case PROCESS_CREATED: {
            process_created_params_t * params = (
                process_created_params_t *) registers[R1];
            gdb_on_process_created(params->handle, params->
                segment_address);
            break;
        }
        case PROCESS_DESTROYED: {
            process_destroyed_params_t * params = (
                process_destroyed_params_t *) registers[R1];
            gdb_on_process_destroyed(params->handle);
            break;
        }
    }
}
```

Figure 4.2: The `handle_gdb_syscall` function.

CPU Driver Initialization

The stub is initialized similarly to how it is done for a non-Barrelfish stub. The difference is the inclusion of an extra function call to the stub, invoked by core #0 only. The function, `gdb_on_core_setup`, is called when instantiating the Barrelfish CPU driver on additional CPU cores. The function has the following signature:

```
void gdb_on_core_setup(uint32_t core_id, uint32_t text_segment);
```

This hook provides the debugger with the CPU driver location in memory. Each core has a separate image loaded into memory, where the location is determined at runtime. GDB needs the text segment location to provide proper symbol information. The details on how symbol information is provided is described in subsection 4.3.7.

Note that this function is not invoked for core #0, as its CPU driver image is always located at address 0.

Process Creation and Destruction

Hooks are inserted into Barrelfish to notify the stub when a *dispatcher object* is created or destroyed. A process is represented as several dispatcher objects, one for each core the process may run on.

The hooks for dispatcher creation events are inserted into the code loading ELF files to memory. The dispatcher handle and the text segment location is stored to a data structure in the stub. In addition, the stub uses a counter to generate a unique id to each dispatcher. This identifier is used by the GDB host to determine which process to attach to. The text segment location is required for symbol information in GDB.

The hooks uses the system call mechanism described in section 4.3.3:

```
process_created_params_t gdb_params;
gdb_params.handle = handle;
gdb_params.segment_address = vbase + reloc_distance;
invoke_gdb_syscall(PROCESS_CREATED, (void *) &gdb_params);
```

A similar hook notifies the stub when a dispatcher is destroyed. This hook ensures that references to destroyed dispatchers are removed from the internal data structures.

4.3.4 Listing Available Processes

The remote protocol defines a packet called `qRcmd` which enables stub to implement custom commands. The user can invoke custom commands by typing `monitor` `[command-with-arguments]` in the GDB CLI. The command with arguments is transferred to the stub as a single text string with the `qRcmd` packet.

The stub implements a custom command to list available processes. This commands is invoked by typing `monitor processes` in the GDB CLI. The stub will return a string containing the name and process id for each active process.

4.3.5 Attaching a Process

The stub implements the `vAttach` packet. The packet takes a single argument, a process id (pid), which is the process to attach to. The pid might refer to an existing process and or future process (by guessing the pid of the desired process). The GDB assumes that the remote responds with a stop reply packet once the process is attached.

If the pid refers to an unknown dispatcher, the actual attaching is performed once the stub gets notified about its creation.

The following steps is performed to attach to a dispatcher:

1. The dispatcher handle is retrieved by looking up the pid in the internal data structures.
2. The dispatcher object is retrieved from Barrelfish using the dispatcher handle.
3. The dispatcher object holds the location to resume execution next time the dispatcher is scheduled. The stub inserts a temporary breakpoint at that location.
4. The debugger returns and Barrelfish resumes execution.
5. Once Barrelfish schedules the dispatcher, it will immediately hit the breakpoint, and return control to the debugger.
6. The stub sends a stop reply packet to notify the host that the dispatcher attachment is complete. This step completes the attachment process.

4.3.6 Obtaining the Currently Executing Process

The global variable `dcb_current` in the Barrelfish CPU driver holds a reference to the dispatcher object of the currently active dispatcher. The stub accesses this variable to determine the current process, which is reported back to GDB host in stop reply packets. The location of the variable has to be calculated if the core running the debugger is not core #0. The relative offset of `dcb_current` is added together with the text segment location of the current core's CPU driver. The resulting value is the correct address of `dcb_current`. The C code for the calculation is shown below:

```
struct dcb * current_disp = *((struct dcb **)(((uint32_t)&dcb_current)
    + text_reloc));
```

The calculation of `current_disp` consists of the following steps:

1. Get the relative address of the `dcb_current` variable.
2. Add the CPU driver text segment location to the relative address found in step 1.
3. Cast the resulting value to a pointer to the `dcb_current` variable. Note that `dcb_current` itself is a pointer.
4. Dereference the pointer to get current core's `dcb_current` value.

Once the dispatcher handler is found, the pid is determined by a lookup in the internal data structures.

4.3.7 Providing Debug Symbols

The GDB stub uses two mechanism to ensure that GDB provides correct debug symbols. GDB provides source code mapping, such as the location of instructions and variables in memory, when debug symbols are available. The requisite is the presence of the symbol file, e.g. the ELF file of the program, and the memory location where the text segment is loaded.

Debug Symbols for CPU Driver

The CPU driver's ELF file is passed as argument when launching GDB. This file provides the debug symbols for the CPU driver. The text segment location is provided by the stub. The GDB host queries the stub for relocation offsets at the beginning of the session. The host sends a `qOffsets` packet. The stub responds with the text string `TextSeg=xxx`, where *xxx* is text segment location. Note that this location is provided by Barrelfish using the hook described in section 4.3.3.

Debug Symbols for Processes

The remote protocol defines a mechanism so a remote can notify GDB when a shared library is loaded or unloaded. The stub halts the program and sends a stop reply packet marked with a special flag on a shared library event. The host may then query the list of shared library using the `Xfer:libraries:read` packet. The remote responds with a XML structure listing the library file path and segment address of all shared libraries.

The XML code below presents an example:

```
<library-list>
  <library name="/lib/libc.so.6">
    <segment address="0x10000000"/>
  </library>
</library-list>
```

This mechanism is used to provide the path of the process symbol file and its text segment location to GDB. The file path is the location of the program's ELF file relative to the Barrelfish build directory. When GDB is launched from this directory, it will be able to automatically load debug symbols for all running Barrelfish processes.

Chapter 5

Testing the Debugger

5.1 Testing Methodology

This chapter describes the testing scheme employed to verify the correctness of the debugger. The standalone debugger and the debugger for Barrelfish were tested separately, and they will be described in separate sections through this chapter.

The debugger for Barrelfish is technically very similar to the standalone debugger. The major difference is that the Barrelfish stub also implements the multiprocess extension required for the Barrelfish integration. For this reason, the tests for the Barrelfish debugger focus on the functionality that is specific for Barrelfish. They will not cover functionality that is also present in the standalone debugger.

The test scheme focuses on functional testing to ensure that debugger meets the functional requirements. The debugger is mostly tested at a high level, with the exception being the unit tests described in subsection 5.1.1. The tests were performed by running a complete debugger setup and executing GDB commands on the host. Such testing is known as *system testing* as the complete system is tested.

A single GDB command may result in several packets being transmitted back and forth, and thus a single command will test several parts of the debugger. A good example is the single step command, which utilizes several packets like the `vCont` for range stepping and `m` for reading memory. As a result, most parts of the stub can be tested by only executing a few different commands. The behaviour of an operation is observed by inspecting packet content and state of the SHMAC system.

5.1.1 GDB for Bare-Metal Programs

This section explains the testing of the standalone GDB stub. The stub was tested using both high-level tests (*system testing*) and low-level tests (*unit testing*). The system tests cover the overall functionality, while the unit tests cover the instruction decoder used for single stepping.

System Testing through GDB

A dedicated test application was developed for the single purpose of testing the debugger. The tests are peformed by debugging this program application it is running on SHMAC. As described in section 5.1, the tests are performed by executing GDB commands on the host and examining the result.

The application is designed to cover all the required scenarios defined by the test plan. Such scenarios include for instance the use of timed interrupts and software interrupts. The code utilizes several types of C flow control statements (loops, branches) so that single stepping and breakpoints can be tested thoroughly.

The source code of the test application can be located by consulting Appendix A.

Unit Testing Single Stepping

The correctness of the instruction decoder, which is part of the single stepping mechanism, was verified using unit testing. The instruction decoder is responsible for detecting and decoding instructions that may modify the `PC` register. Once such instruction is decoded, the future value of `PC` is calculated. It is important the calculated value is correct, so that the single step breakpoint is inserted at the subsequent instruction. As several different instructions may modify the `PC` register, and decoding these instructions are rather complex, the code for the instruction decoder became a major part of the overall codebase. For this reason, manually testing the instruction decoder was deemed as too time consuming. Instead, the unit testing was chosen as it was considered more time effective, in addition to testing the decoder more thoroughly.

The unit tests are designed to run on SHMAC itself. As parts of the instruction decoder expected 32-bit address space, testing it on a 64-bit personal computer was practically not possible.

The source code of the unit tests can be located by consulting Appendix A.

5.1.2 GDB for Barrelfish

The tests covers functionality that is specific to the GDB stub for Barrelfish, mainly the commands which are part of the multiprocess extension. The tests were performed on the multicore port of Barrelfish. This port was designed to run on a more powerful Versatile Express development system featuring 4GB RAM. The SHMAC variant with multiple TTY channels was at the time of testing only compatible with the older RealView development system featuring only 32MB RAM. The GDB stub for Barrelfish required the presence of multiple TTY channels, and could therefore not run on the Versatile Express system. In order to run Barrelfish on only 32MB of RAM, certain modifications were performed on its codebase. System processes that were not required for booting were disabled. In addition, Barrelfish was restricted to boot two cores only, leaving the third core unused. With these modifications, Barrelfish was able to boot the CPU driver and two system processes on two cores. Note that these modifications were performed for testing purposes only. The SHMAC variant with multiple TTY channels will in the future be compatible with the Versatile development system, so that Barrelfish can be debugged without requiring any modifications.

5.2 Test Plan

5.2.1 GDB for Bare-Metal Programs

System Testing through GDB

The test cases for system tests are listed below.

1. Verify that a GDB host is able to connect to the stub, using one GDB instance per core. Each core should be connectable through the TTY channel specified in the configure struct. The stub should send an initial stop reply packet once the first static breakpoint is hit.

2. Verify that GDB is able to kill the program on SHMAC. The stub should call the *kill* function specified in the configure struct.

3. Verify that disabling debugging on individual cores works as expected. The program running on those cores should not halt on static breakpoints, but continue execution to the subsequent instruction. If a disabled core hits a breakpoint inserted by a different core, it should halt execution. Once the breakpoint is removed, the core should continue execution.

4. Verify that the read and write register packets work correctly.

5. Verify that the memory write and memory read packets work correctly.

6. Verify that range stepping behaves correctly. A range step is performed using the *step* command in GDB, which steps over a single line of code. This test also verifies the instruction single step operation the and continue operation, as range stepping implementation in the stub uses both those operations. Stepping should behave correctly on all types of flow control statements in the C language (if statements, switch statements, while/for/do-while loops, function call, function return).

7. Verify that the stub handles static breakpoints correctly. The program should halt on all static breakpoint. The stub should be able to continue the program afterwards.

8. Verify that the stub handles normal breakpoints correctly. The program should halt on the locations where breakpoints are inserted. Removing a breakpoint should restore the original instruction. The stub should be able to continue the program afterwards. Breakpoints should behave correctly when inserted on all types on flow control statements in the C language.

9. Verify that a GDB interrupt is able to halt the program. The stub should be able to halt the program at any time given that interrupts are enabled and the current mode is not *und* mode. Once halted, the stub should notify the host that it was halted by a GDB interrupt. The stub should be able to continue the program afterwards.

10. Verify that stub is able to debug code running in *irq* and *svc* mode. The test is performed by inserting a breakpoint inside the software interrupt handler and the IRQ interrupt handler. The program should halt on the breakpoint every time a software interrupt or IRQ interrupt occur. The stub should be able to continue execution after hitting any of the breakpoints. Note that *fiq* mode is not tested as it is currently not supported by the CPU core.

11. Verify that the stub only enables interrupts for the debug channels when entering the exception handler. This verification is performed by examining the `CPSR` register and status registers of the interrupt controller.

Unit Testing Single Stepping

The test suite for the instruction decoder consists of 21 unit tests. Each test verifies the `PC` calculation for a single instruction. One of the unit tests is presented below as an example. This test verifies the decoding of the instruction `add pc, r0, r1`. The hexadecimal literal (*0xe090f001*) is the machine code representation of the instruction.

```
registers[R0] = 0x100;
registers[R1] = 0x8;
uint32_t addr2 = handle_add_pc_instr(0xe090f001, &registers[0]);
uint32_t expected_addr2 = registers[R0] + registers[R1];
assert_equals(expected_addr2, addr2, "add_pc_test_2");
```

The unit tests are listed in Table 5.1.

Table 5.1: Unit tests for instruction decoder.

Name	Type	Instruction
swi_test_1	swi	swi 42
mov_pc_test_1	mov	mov pc, lr
mov_pc_test_2	mov	movs pc, lr
mov_pc_test_3	mov	mov pc, r0
branch_test_1	b	b 134
branch_test_2	b	bl 134
add_pc_test_1	add	add pc, r0, #4
add_pc_test_2	add	add pc, r0, r1
add_pc_test_3	add	addls pc, pc, r2, lsl #2
sub_pc_test_1	sub	sub pc, lr, #4
sub_pc_test_2	sub	subs pc, lr, #4
sub_pc_test_3	sub	sub pc, r0, #8
ldr_pc_test_1	ldr	ldr pc, [r3, #4]
ldr_pc_test_2	ldr	ldr pc, [r0, r1]
ldr_pc_test_3	ldr	ldrls pc, [pc, r3, lsl #2]
ldm_pc_test_1	ldm	pop {r0, r1, r2, pc}
ldm_pc_test_2	ldm	ldm r0, {r1, r2, pc}
ldm_pc_test_3	ldm	ldmib sp!, {r0, r1, r2, pc}
ldm_pc_test_4	ldm	ldmdb sp!, {r0, r1, r2, pc}
ldm_pc_test_5	ldm	ldmda sp!, {r0, r1, r2, pc}
ldm_pc_test_6	ldm	pop {lr, pc}

5.2.2 GDB for Barrelfish

The test cases for system tests are listed below.

1. Verify that the GDB host can connect to the stub, using one GDB instance per core. The stub should provide the host with the correct text segment location for each CPU driver. The values are verified by comparing each core's `PC` register with the respective text segment location.

2. Verify that the GDB host is able to attach to an existing dispatcher. The stub should insert a temporary breakpoint at the correct location inside the dispatcher. Once the Barrelfish schedules the execution of the attached dispatcher, the dispatcher should halt. The stub should then report back to the host that the attachment is complete.

3. Verify that the GDB host is able to attach to a dispatcher that will be created in future. The stub should attach to the correct dispatcher based on the pid. The temporary breakpoint should be inserted as soon at the stub is notified about the dispatcher creation.

4. Verify that the stub notifies the GDB host once a new dispatcher is created. The dispatcher name and the reported text segment location for each dispatcher should be verified.

Note that both user programs and system processes use the same internal representation in Barrelfish, specifically as a set of dispatcher objects. This is the reason why the test plan does not test debugging of user programs and system processes separately.

5.3 Test Results

5.3.1 GDB for Bare-Metal Programs

The debugger was able to pass the tests listed in section 5.2.1. Although the tests do not cover all possible scenarios, it gives a strong indication that the debugger operates correctly.

All unit tests listed in section 5.2.1 passed. This indicates that the instruction decoding works correctly for the instructions supported. It does not test single stepping directly as only the decoding is tested. The range stepping test in section 5.2.1 covers the single step command. The combination of the unit tests and the range stepping test gives strong indication that single stepping operates correctly.

5.3.2 GDB for Barrelfish

The debugger was able to pass the tests listed in subsection 5.2.2. The results give a strong indication that the Barrelfish debugger operates correctly. Note that it was only possible test a modified instance of Barrelfish running a limited set of system processes. As the modified version does not significantly alter the internals of Barrelfish, but mainly reduces the number of programs started, it is unlikely that the debugger does not work similarly well with a complete Barrelfish setup.

Chapter 6 Evaluation

6.1 Performance Enhancements

Measuring the performance increase introduced by the enhancements in section 3.6 is not straightforward. Different operations takes different amount of time based on many factors such as the number of instructions in the current line.

As single stepping is one of the most frequent operations performed during a debug session, it was decided that measuring the duration of stepping a line of code was a good indicator for the performance comparison. The test used the following line of code:

```
i = i * 10 - 3;
```

Prior to the optimizations, single stepping over the code line above took 8 seconds. The optimizations reduced the duration down to 1.5 seconds, giving a 5 times speedup. Different code lines will of course result in different speedups.

6.2 Limitations

6.2.1 Limitations with the Single Stepping Implementation

Single stepping is implemented in software and relies on proper detection and decoding of instructions that modify the `PC` register. The stub has to insert a temporary breakpoint at the subsequent instruction so that the program halts after executing the single instruction. If the temporary breakpoint is inserted at an incorrect location, the program will continue execution until it is halted for some other reason. The current implementation only handles the instruction variants discovered during the analysis of the instructions generated by the GCC compiler for ARM. There might be other `PC` modifying instructions typically generated which were not discovered. If such instruction is single stepped, it may lead to the temporary breakpoint being inserted at `PC + 4` instead of the correct location. Hand-coded assembly code may for instance use unusual instructions for branching for optimizations purposes, which may also lead to faulty single stepping.

While the instruction decoder can be further developed to handle all types of instructions, adding hardware assisted single stepping is the preferred approach for proper single stepping. Handling all instructions would essentially require a complete instruction simulator, which will enable correct single stepping for all kind of instructions. The disadvantage with such solution is the implementation complexity. Implementing and testing the simulator would take a considerable amount of time.

A better solution is to implement single stepping in hardware. The Amber CPU core can be modified to include support for hardware assisted single stepping. Doing single stepping in hardware is believed to require less effort than implementing an instruction simulator. As this thesis focus on the software support for debugging, modifying the Amber CPU was out of scope.

The single stepping mechanism can be implemented such that the it traps the program after executing the instruction at a given memory address. Activating single stepping could be as simple as writing the memory address to a dedicated co-processor register.

Imperfect single stepping can be mitigated by using breakpoints. The user of the debugger can insert a breakpoint at subsequent instruction when single stepping an instruction which the stub is known to not handle correctly. This workaround will require that the user manually inspects the problematic instruction and is able to predict where the program will branch to.

6.2.2 Using Breakpoints in Multicore Programs

Breakpoints are inserted to the program by substituting the original instruction with a trap instruction. This approach has a few implications if this instruction is part of a code segment shared by multiple cores. First of all, any breakpoint inserted will trap all cores that will execute that instruction. This may or may not be intentional by the user, but as the cores are debugged individually, it might not be the best fitting behaviour.

A major issue with the breakpoint implementation is the fact that GDB removes all breakpoints once the program halts (and reinserts them prior to continuing the program). Once the owner of the breakpoints hits it, the breakpoint will be removed and any other cores will execute the original instruction instead of hitting the breakpoint. This behaviour is confusing for the user. GDB uses all-stop mode per default and expects that all threads halt whenever one thread halts. That is the reason why GDB assumes that it is safe to remove all breakpoints on halt.

The proper solution of all the aforementioned issues is the use of hardware breakpoints. Each core would then have a separate set of breakpoints. Inserting a hardware breakpoint does not require substituting the original instruction with a trap instruction. Instead, the CPU core is configured to trap when `PC` reaches a specific value. Adding support for hardware breakpoints would require modifications of the Amber core, which is out of scope for this project.

Shared breakpoints could be implemented if the debugger was modified to utilize the GDB multithread abstraction (either all-stop mode or non-stop mode). Though, as described in subsection 3.7.1, modifications to the SHMAC hardware would be required.

6.2.3 GDB Interrupts

The GDB host will not be able to interrupt a core using Ctrl+C if that core has interrupts disabled. The implementation of GDB interrupt relies on host interrupts being enabled. If the core has interrupts disabled, it will not be notified about data being sent through the debug channel. Consequently, it will not receive and detect the interrupt token from the host. Note that the host will still be able to halt the core using either standard or static breakpoints. This limitation affects Barrelfish as its CPU driver is designed as an uninterruptable kernel.

6.3 The SHMAC Platform

The SHMAC platform was enhanced during the development of the stub. The CPU tile received an update adding a newer ARM instruction set, and the APB tile was upgraded with 15 new TTY channels. These two updates were beneficial for the debugger implementation.

6.3.1 Benefits of ARMv3

The Amber CPU core was improved in the early phase of this project. The ISA was upgraded from ARMv2a to ARMv3. The ARMv3 represents a major upgrade over ARMv2a by migrating from a 26-bit to a 32-bit program counter. What benefitted most was the introduction of a new operating mode; undefined (*und*) mode. ARMv2a used *svc* mode for the undefined instruction handler, effectively making it impossible to debug code running in *svc* mode. The issue is discussed in section 3.5.5.

6.3.2 Benefits of Multiple TTY Channels

The SHMAC test platform available during the two first months of this project had limited communication capabilities. The original APB tile did only provide a single TTY channel to the host controller. As a result, the debugger and program output had to share a single channel. The solution to this problem was to wrap all program output in an `O` (program console output) packet [RSea14, p. 587]. The packet takes a single argument, an ASCII string encoded in the hex format. The parameter is decoded and printed to the GDB terminal when packet is received on GDB host. Programs using the debugger had to use dedicated print functions supplied by the GDB stub for console output. This requirement made it harder to integrate the debugger, as it would likely involve larger modifications to a program's source code. In addition, it would be much harder to implement multicore support using the current approach, as all the debug channels would have to be multiplexed over a single TTY channel.

The need of special treatment of program output was eliminated once the APB tile was extended to 16 TTY channels.

6.4 Comparing the Implementation with The Specification

This section evaluates the functionality of the implementation against the requirements listed in section 3.1 and section 4.2.

6.4.1 GDB for Bare-Metal Programs

Support breakpoints

The debugger supports both static breakpoints and standard breakpoints. Insertion and removal of standard breakpoints is supported through the software breakpoint packets defined in the remote protocol. Testing has verified that the breakpoint functionality is working correctly.

Support GDB interrupts

GDB interrupts are supported by the debugger and testing has shown that is working correctly. Though, the current implementation requires that the CPU core has interrupts enabled. GDB is not able to interrupt code running in *und* mode. As this mode is rarely used, the limitation is not a big issue.

Debug code running in any mode

The debugger is able to debug code running in all modes except *und* mode. *Und* mode is not supported for technical reasons as explained in section 3.5.5. Testing has shown that the debugger is able to debug code running in *svc*, *irq* and *usr* mode. *Fiq* mode is not tested as it is not supported by CPU core.

Debug programs running on multiple cores

The debugger is able to debug multicore programs by debugging each core individually with a separate GDB instance. This approach was determined to be the best alternative, having the most suitable trade-offs.

No dependencies on operating systems or external libraries
The debugger has no dependencies on libraries or operating systems. It is written in C without standard library and ARM assembly. As a result, it is easy to integrate the stub into SHMAC programs.

Simple to integrate into SHMAC programs
Integrating the stub to a program only requires small modification to the interrupt vector and the program's main function. These modification should be relatively simple to perform on most program. Performing these modifications on Barrelfish were for instance straightforward.

Adequate performance
Typically, single stepping over a single line of code takes less than two seconds. Exceptions include stepping over long function calls, instructions with multiple memory references, or complicated arithmetic expressions. Operations that are more complex, such as calling a program function through GDB, will take considerable more time. The performance is briefly discussed in section 6.1.

6.4.2 GDB for Barrelfish

Kernel debugging
The debugger is able to debug the CPU driver and system processes. The stub implements the multiprocess extension for the remote protocol, which provides commands for attaching and detaching any processes on SHMAC.

Debug user programs
As stated in the above item, the implementation of the multiprocess extension provides the necessary support for debugging any process in Barrelfish. This also includes user programs, as they use the same internal representation as system processes.

Debug multiple cores simultaneously
The debugger is able to debug multicore programs by debugging each core with a separate GDB instance.

Provide debug symbols
The debugger provides automatic lookup of debug symbols for the CPU driver and processes running on Barrelfish.

6.5 Evaluating Assignment Tasks

This sections reviews the result in terms of the tasks identified in section 1.5.

T1 Add support for debugging with the GNU Debugger (GDB) on SHMAC.
GDB debugging for SHMAC is enabled by the GDB remote stub described in chapter 3. The debugger is a software solution, although it does require a SHMAC test platform with certain characteristics. It is designed for ARMv3, and require the presence of the enhanced APB tile with 16 TTY channels. The remote stub supports the core debugger functionality such as breakpoints. Its functionality is tested and verified, as described in chapter 5. The debugger is ready for use as it is today.

T2 Facilitate debugging of the Barrelfish kernel using the software developed in `T1`.
The GDB remote stub has been integrated with Barrelfish. The remote stub for Barrelfish is extended to support the GDB multiprocess extension, which enables debugging of processes and the CPU driver. Additionally, it is designed to help GDB provide proper debug symbols, such that GDB will automatically give source code mapping when attaching to a process. Its functionality is tested and verified, as described in chapter 5.

T3 Extend the debugger implementation in `T1` and `T2` to enable debugging of multiple SHMAC CPU cores simultaneously.
The solution for `T1` and `T2` is able to debug multiple cores simultaneously. The challenges related to multicore debugging, and how the implementation addresses those challenges, is described in section 3.7.

T4 Extend the debugger implementation in `T2` to facilitate debugging of user programs in Barrelfish.
The debugger for Barrelfish supports debugging of user programs in Barrelfish. Barrelfish does not distinguish between system processes, i.e. processes that are part of the operating system, and user programs. Both are represented as a set of dispatcher objects. Consequently, when debugging support for system processes was added, it also included support for debugging user programs.

T5 Extend the debugger implementation in `T1` to enable debugging of the Linux kernel.
This task was never initiated. The team porting Linux did not have a great need for a debugger. In addition, the above tasks were prioritized thereby not leaving enough time for this task. Linux provides its own GDB remote stub, although it is not known if it supports older ARM ISAs such as ARMv4.

T6 Implement a software providing visualisation of core usage and communication.
As this task was lower prioritized than the above tasks, time did not permit the execution of this task.

Chapter 7 Future Work

7.1 Adding ARMv4 Support

The modified SHMAC CPU core is in progress of being upgraded to the ARMv4 instruction set [AA14]. Once the new CPU core is finished, it will become the default CPU tile for SHMAC. For that reason, it is preferable that the debugger is updated to support ARMv4 once the core is finished.

Adding ARMv4 support should only require minor modifications to the stubs. The details are as following:

- ARMv4 introduced a new operating mode called *system mode*. The code responsible for storing and restoring register values has to be modified to handle traps coming from code executing in the new mode.
- The CPU core also supports the `bx` instruction even though it is part of the ARMv4T (ARMv4 with Thumb) instruction set. The instruction was included for better compatibility with the toolchain used for compiling Linux. As this instruction is commonly used for branching, the instruction decoder used for single stepping should be updated to support the new instruction.

7.2 Adding Debugging Capabilities to SHMAC CPU Core

7.2.1 Breakpoints and Single Stepping

The current SHMAC CPU core does not provide any dedicated debugging functionality. Having hardware assisted single stepping and breakpoints would solve many of the limitations with current debugger implementation, as described in subsection 6.2.2 and subsection 6.2.1. If hardware breakpoints are added to the CPU, it should provide a high enough capacity for breakpoints such that software breakpoint would not be needed at all.

7.2.2 Range Stepping

Range stepping is another operation that would be greatly improved with proper hardware support. The current stub implements range stepping by single stepping multiple instructions, which is terribly ineffective as the program is halted for each instruction executed. With hardware assisted range stepping, the program would only be halted after the range step operation is complete. The range step operation could be initiated by writing range (start and stop address) to co-processor registers.

7.2.3 Dedicated Debug Interface

The CPU core could be further enhanced by adding a dedicated debug interface. The host controller should be able to debug and control each CPU core through this interface. The debug interface should preferably implement a request-reply protocol similar to the GDB remote serial protocol. The GDB stub would then be modified to run as a separate program on the host controller, controlling a CPU core through the debug interface.

The protocol should implement necessary commands for basic debug operations:

1. Read/write SHMAC memory
2. Read/write registers
3. Stop/continue instruction execution
4. Single stepping and range stepping
5. Insert/remove breakpoints

In addition, the protocol should enable the CPU core to report exceptions as they occur.

One of the advantages with this design is that it enables debugging of any program running on SHMAC. The current design with a GDB stub is less practical as the program has to be modified and recompiled to enable debugging. Having the debugger in software makes it more fragile. If the program modifies the interrupt vector during runtime, the debugger may malfunction. With the debug interface, there is no need to link a GDB stub to the program, thereby removing many of the limitations with the current design.

A disadvantage with the debug interface is that it will require larger modifications to both the CPU core and the SHMAC infrastructure. New device drivers for Linux has to be developed to enable communication over the new interface.

Chapter 8 Conclusion

This thesis provides the SHMAC project with a software debugger based on the GNU Debugger (GDB). Future software development on SHMAC will greatly benefit from having a proper debugging tool. As the debugger is based on GDB, it provides the SHMAC developers with a familiar product that has a rich feature set.

The debugger has been integrated in Barrelfish, the first functional operating system for SHMAC. This integration enables debugging of the OS kernel and processes running on Barrelfish. The combination of a debugger and an operating system represents a powerful software platform for SHMAC, greatly simplifying future software development.

The thesis describes the implementation of a GDB remote stub, a software component which, when linked to a SHMAC program, enables debugging through GDB. The thesis discusses the challenges with multicore debugging and why the current implementation achieves the best trade-offs. The integration of the remote stub with Barrelfish demonstrates that the stub can easily be integrated with larger programs. Various performance optimizations were performed to make the debugger more responsive, and the evaluation presents the substantial speedup attained. Future work discuss the benefits with hardware assisted debugging, and describes how current limitations with single stepping and breakpoint implementation can be solved by enhancing the SHMAC CPU core. The last contribution is the user guide in the appendix, which explains how to use the debugger.

The debugger has been extensively tested to ensure it is working correctly. The test results are positive and concludes that debugger is stable. Overall, the thesis has accomplished its primary goal, which is to provide SHMAC with a fully functional debugger.

References

[AA14] Håkon Amundsen and Joakim Andersson. Linux for SHMAC. Master Thesis, June 2014.

[Aid] Nir Aides. Winpdb - A Platform Independent Python Debugger. http://winpdb.org/.

[ARM94] ARM. *ARM7DI Data Sheet*, 1994.

[ARM01] ARM. *ARM7EJ-S Technical Reference Manual*, 2001.

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[BC11] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[Bjo14] Benjamin Bjornseth. Towards energy efficiency measurements on the SHMAC (working title). Master Thesis, December 2014.

[BS13] Benjamin Bjørnseth and Bjørn C Seime. Porting and Evaluating Barrelfish for SHMAC. TDT4501 - Computer Science, Specialization Project, December 2013.

[CLO] CLOC - Count Lines of Code. http://cloc.sourceforge.net/.

[DGR+74] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[EBSA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[EECa] The EECS research initiative. http://www.ntnu.edu/ime/eecs.

[EECb] The SHMAC Project. http://www.ntnu.edu/ime/eecs/shmac.

[Fre] Free Software Foundation. GDB: The GNU Project Debugger. https://www.sourceware.org/gdb/.

[GDB09] GDB 7.0 released! https://sourceware.org/ml/gdb-announce/2009/msg00001.html, October 2009.

[GDB14] GDB 7.7 released! https://sourceware.org/ml/gdb-announce/2014/msg00001.html, February 2014.

[HK10] Jim Held and Sean Koehl. Introducing the Single-Chip Cloud Computer. *Intel White Paper*, 1(4):5–3, 2010.

[Int86] Intel. *INTEL 80386 PROGRAMMER'S REFERENCE MANUAL*, 1986.

[JC09] Randy Johnson and Stewart Christie. JTAG 101 IEEE 1149.x and Software Debug. Technical report, Intel Corporation, January 2009.

[M$^+$65] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[MSD] Microsoft Developer Network: Breakpoints and Tracepoints. http://msdn.microsoft.com/en-us/library/ktf38f66(v=vs.90).aspx.

[Ope] OpenCores. http://opencores.org/.

[Ope13] Opencores.org. Amber ARM-compatible core. http://opencores.org/project,amber, November 2013.

[PSMR11] Simon Peter, Adrian Schüpbach, Dominik Menzi, and Timothy Roscoe. Early experience with the Barrelfish OS and the Single-Chip Cloud Computer. In *MARC Symposium*, pages 35–39, 2011.

[RSea14] Stan Shebs Richard Stallman, Roland Pesch and et al. *Debugging with gdb*, Tenth Edition edition, February 2014.

[Sch13] Terje Schjelderup. Debugging Environment for SHMAC. TDT4501 - Computer Science, Specialization Project, December 2013.

[Sou] SourceForge. Ser2net - Serial port to network proxy. http://sourceforge.net/projects/ser2net/.

[SPA$^+$08] Nathan Sidwell, Vladimir Prus, Pedro Alves, Sandra Loosemore, and Jim Blandy. Non-stop Multi-threaded Debugging in GDB. In *GCC Developers' Summit*, page 117, 2008.

[Tan02] Minheng Tan. A minimal GDB stub for embedded remote debugging. *Columbia University*, 2002.

[Tay12] Michael B Taylor. Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1131–1136. ACM, 2012.

[Wik] Wikimedia Commons. Winpdb 1.36 screenshot. http://en.wikipedia.org/wiki/File:Winpdb-1.3.6.png.

[WT03] James A Whittaker and Herbert H Thompson. Black box debugging. *Queue*, 1(9):68, 2003.

[ZL] Angen Zheng and Jack Lange. A GDB Stub for the Lightweight HPC OS-Kitten.

[Zü] Systems @ ETH Zürich. The Barrelfish Operating System. http://www.barrelfish.org/.

Appendix A

Source code overview

This chapter presents an overview of the source code for the standalone debugger and debugger for Barrelfish.

A.1 GDB for Bare-Metal Programs

The GDB stub + test program source code is located at https://bitbucket.org/bjorncs/gdb-project. The source code can be downloaded manually from website or by using git:

```
git clone https://bjorncs@bitbucket.org/bjorncs/gdb-project.git
```

The source code is also available as an attachment at DAIM (*Digital Arkivering og Innlevering av Masteroppgave*).

The root folder contains 3 folders. Each of them are described in the table below:

Folder	Description
gdb-stub	The GDB stub source code.
test-program	The test application source code. This application was used during the functional testing.
single-step-tester	Contains the unit tests for the single stepping instruction decoder. The tests

The following table shows the total lines of code. The statistics are generated by CLOC (Count Lines of Code) [CLO]. The line counts does not include blank lines or comments.

Language	Files	Lines of code
C	11	1636
C Headers	13	352
Assembly	3	420
Sum	27	2228

A.2 GDB for Barrelfish

The source code for Barrelfish with the GDB stub integrated is located at a private repository owned by Benjamin Bjørnseth and Bjørn C. Seime.

The source code for the Barrelfish GDB stub alone is available as an attachment at DAIM.

The GDB stub files are located in the following directory relative to root of the Barrelfish codebase:

Type	Location
Implementation files	kernel/arch/shmac/gdb
Header files	kernel/include/arch/shmac/gdb

The following table shows the total lines of code. The statistics are generated in the same way as the source code for the standalone stub. Note that the line counts are lower as the counts for the standalone stub also includes the source files for the test application and the unit tests.

Language	Files	Lines of code
C	7	1610
C Headers	9	104
Assembly	1	145
Sum	17	1859

Appendix B
User Guide: Debugging Programs on SHMAC

This is a guide that cover the necessary setup for using GDB for SHMAC. Note that it does not explain how to use GDB in general.

The debugger is compatible with any SHMAC configuration having the following properties:

- ARMv3 compatible CPU cores (e.g. *arv3*).
- The updated APB tile from Asbjørn (with 16 TTY channels).

B.1 Install GDB 7.7 for ARM

The stub is designed for GDB 7.7 or newer. Older versions are not tested, but might still work. The GDB installation must have ARM as target platform. The easiest way to obtain the correct version is to compile GDB from source. You may install GDB either on your computer or on the host controller on SHMAC.

1. Download and extract the source code (http://ftp.gnu.org/gnu/gdb/gdb-7.7.tar.gz).
2. Compile GDB using the following commands:

```
mkdir gdb-build
cd gdb-build
CFLAGS=-Wno-error=deprecated-declarations [path to source code]/
     configure --target=armv3-none-eabi
make
```

B.2 Setup *ser2net* on SHMAC Host Controller

This step is not necessary if you intent to run GDB on host controller.

1. Install ser2net from package manager:

```
sudo apt-get install ser2net
```

2. Edit the file `/etc/ser2net.conf` and remove the default configuration. Add the following lines to the bottom of the file:

```
4242:raw:600:/dev/ttySHMAC0:115200 8DATABITS NONE 1STOPBIT
4243:raw:600:/dev/ttySHMAC1:115200 8DATABITS NONE 1STOPBIT
4244:raw:600:/dev/ttySHMAC2:115200 8DATABITS NONE 1STOPBIT
4245:raw:600:/dev/ttySHMAC3:115200 8DATABITS NONE 1STOPBIT
4246:raw:600:/dev/ttySHMAC4:115200 8DATABITS NONE 1STOPBIT
4247:raw:600:/dev/ttySHMAC5:115200 8DATABITS NONE 1STOPBIT
4248:raw:600:/dev/ttySHMAC6:115200 8DATABITS NONE 1STOPBIT
4249:raw:600:/dev/ttySHMAC7:115200 8DATABITS NONE 1STOPBIT
4250:raw:600:/dev/ttySHMAC8:115200 8DATABITS NONE 1STOPBIT
4251:raw:600:/dev/ttySHMAC9:115200 8DATABITS NONE 1STOPBIT
4252:raw:600:/dev/ttySHMAC10:115200 8DATABITS NONE 1STOPBIT
4253:raw:600:/dev/ttySHMAC11:115200 8DATABITS NONE 1STOPBIT
4254:raw:600:/dev/ttySHMAC12:115200 8DATABITS NONE 1STOPBIT
4255:raw:600:/dev/ttySHMAC13:115200 8DATABITS NONE 1STOPBIT
4256:raw:600:/dev/ttySHMAC14:115200 8DATABITS NONE 1STOPBIT
4257:raw:600:/dev/ttySHMAC15:115200 8DATABITS NONE 1STOPBIT
```

   This configuration will map */dev/ttySHMAC0* to IP port 4242, */dev/ttySHMAC1* to 4243, etc.

3. Restart the ser2net service using the following command:

```
sudo /etc/init.d/ser2net restart
```

B.3 Compile the GDB Stub

The GDB stub + test program source code is located at https://bitbucket.org/bjornсs/gdb-project. The source code can be downloaded manually from website or by using git:

```
git clone https://bjorncs@bitbucket.org/bjorncs/gdb-project.git
```

Prerequisites:

– A GCC toolchain for ARMv3. The GDB stub needs the *libc* headers. The test program requires the *libc* itself in addition to the headers. The repository contains a guide for setting up the toolchain required for compiling the GDB stub and Barrelfish.

The header file *defs.h* must be configured prior to compilation. It is located in the directory `gdb-stub/include`. Modify the definitions defining the number of cores and TTY channels:

```
#pragma once

#define NUM_CORES 3
#define NUM_TTY_CHANNELS 16
```

Navigate to the source code root directory. You may build the test program and the stub with `make`. If you only want to build the stub, run `make` from the directory `gdb-stub`. The resulting static library is (*libgdbstub.a*) located in `gdb-stub/build/libs/`.

B.4 Integrate GDB Stub with Program

B.4.1 Insert Interrupt Hooks

1. Modify the interrupt vector so that the the GDB stub handles *irq interrupts* and *undefined instruction exceptions*:

```
vector:
    b reset_handler
    b gdb_undef_instr_handler
    b swi_handler
    b . // Prefetch abort
    b . // Data abort
    b . // Reserved
    b gdb_irq_handler
    b . // FIQ
```

2. Label the entry point for the program's *irq handler* and *undefined instruction handler* with `gdb_irq_handler_program` and `gdb_undef_instr_handler_program` respectively:

```
gdb_irq_handler_program:
irq_handler:
    // Save regs
    push {r0-r3,r12,lr}
    ...

gdb_undef_instr_handler_program:
undef_instr_handler:
    // Save regs
    push {r0-r3,r12,lr}
    ...
```

B.4.2 Insert Call to `gdb_init`

Insert a call to the `gdb_init` function at the start of program. It takes a `gdb_config_t` struct as only argument. `gdb_config_t` is defined in header file *gdb.h*. Example:

```c
#include "gdb.h"
...
void die(void) {
    ...
}
...
int main(void) {
    int cpu_id = *TILEREG_CPUID;
    if (cpu_id == 0) {

        gdb_config_t gdb_config;
        gdb_config.tty_channel[0] = 3;
        gdb_config.tty_channel[1] = 4;
        gdb_config.tty_channel[2] = 5;
        gdb_config.kill_program_handler = &die;
        gdb_init(&gdb_config);
    }
    ...
}
```

The config above configures core 0, 1 and 2 to use TTY channel 3, 4 and 5 respectively. The function `die` will be invoked by the debugger when killing the program. It is up to the program to decide what this function should do.

IMPORTANT: Make sure that `gdb_init` is called by a single core only (in case the main function is run by multiple cores).

B.4.3 Insert the Initial Static Breakpoint

Place a static breakpoint somewhere after the call to `gdb_init`. The static breakpoint is defined as a C pre-processor macro (`GDB_TRAP`). The program will initially halt at the location of this breakpoint.

```c
...
GDB_TRAP;
...
```

B.4.4 Compile the Program with the GDB Stub

Compile the program with the static library *libgdbstub.a*. The necessary header files are located at `gdb-stub/include`. Make sure to compile with debug symbols. It is recommended to compile without optimizations.

B.5 Starting a Debugging Session

1. Start the program on SHMAC using `shmac_program` and `shmac_reset`.
2. Launch GDB on the host machine. Type the following commands to the GDB CLI to attach to the program running on SHMAC:

```
set remotetimeout 12
set remote trace-status-packet 0
set remote query-attached-packet 0
set can-use-hw-watchpoints 0
file [path to program elf file]
target remote [shmac host ip]:[shmac host port]
```

3. GDB is now ready to debug the program.

B.6 Using the Debugger

B.6.1 Interrupting the Program

You may halt the program by pressing Ctrl+C in the GDB CLI.

B.6.2 Static Breakpoints

You may insert static breakpoints to the program as described in subsection B.4.3.

B.6.3 Disabling the Debugger

You may disable the debugger by compiling the stub (*libgdbstub.a*) with the flag `-D DISABLE_DEBUGGER`. Another way of disabling the debugger is to disable each core individually in the `gdb_config` struct, as shown in subsection B.4.2.

B.6.4 Printing the SPSR Register

You can print the content of the SPSR register with the following command:

```
p $fps
```

Appendix C

User guide: Debugging Barrelfish on SHMAC

Prerequisites:

– A GCC toolchain for ARMv3. The repository contains a guide for setting up the toolchain required for compiling Barrelfish on SHMAC.

– An installation of GDB 7.7 for ARM as described in section B.1.

– A *Ser2net* installation on the development system as decribed in section B.2

– The source code of the Barrelfish for SHMAC.

C.1 Enabling and Disabling Debugging

The debugger can be disabled by defining the pre-processor macro `DISABLE_DEBUGGER` in the file `kernel/include/arch/shmac/gdb/enable.h`:

```
#pragma once
#define DISABLE_DEBUGGER
```

C.2 Compiling Barrelfish

Create directory where to build Barrelfish and navigate into it. Then execute the following commands:

```
[barrelfish-src]/hake/hake.sh -s [barrelfish-src] -a shmac
make
```

C.3 Starting a Debugging Session

1. Connect to the host controller using *ssh* and upload the following files to a directory:

   – shmac/shmacfish/header_segment.bin

   – shmac/shmacfish/load_barrelfish.sh

   – shmac/shmacfish/modules_segment.bin

   – shmac/kernel/cpu.bin

   The paths are relative to the Barrelfish build directory.

2. Start Barrelfish using the bootloader script `load_barrelfish.sh`.

3. Launch GDB on the host machine. Type the following commands to the GDB CLI to attach to Barrelfish:

```
set remotetimeout 30
set remote trace-status-packet 0
set can-use-hw-watchpoints 0
set stop-on-solib-events 1
file [barrelfish-build]/shmac/kernel/cpu_symbols.elf
target extended-remote [shmac host ip]:[port]
```

   With the *ser2net* setup described in section B.2, core 0 is debugged from port 4243, core 1 from 4244, etc.

C.4 Using the Debugger

C.4.1 Listing Available Processes

Use the following GDB command to list all available processes in Barrelfish:

```
monitor processes
```

The processes are listed with name and process id.

C.4.2 Attaching to a Process

Use the standard `attach` command in GDB to attach to a process. The required process id is a positive integer. Use the command described in subsection C.4.1 to determine the process id. The process id is determined by a global counter. Each time a process is created, the counter is incremented, and the resulting value is used as the process id for the newly created process. You may use this property to attach to a process that will be created in future. The attach command will in that case not return until the process has been created.