NTNU – Trondheim
Norwegian University of
Science and Technology

Linux for SHMAC

Håkon Furre Amundsen
Joakim Erik Christopher
Andersson

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

Abstract

For several years it has been possible to improve processor performance by taking advantage of the ever increasing transistor density. Recently, the power demand of processors has exceeded their power budget, so it is no longer possible to utilize all parts of the processor at the same time. The heterogeneous processor architecture is suggested in order to circumvent this effect. By containing specialized processor cores that are optimized for certain types of tasks, the single thread performance can still be increased while staying within the power budget.

The SHMAC project provides a research framework for heterogeneous architectures, in which different hardware components acts as interchangeable tiles. Currently, the only processor tile available in the SHMAC project is the ARMv2a ISA compliant Amber processor tile.

The only operating systems currently available for SHMAC is the experimental research operating system Barrelfish. Hence, most software written for SHMAC runs without an operating system, directly above the hardware. As the SHMAC hardware is frequently modified, maintaining compatibility between software and hardware is a tedious job, imposing limitations on the efficiency for SHMAC software development.

This thesis presents the process of porting the Linux kernel to SHMAC. The Linux kernel provides a powerful abstraction layer which allows the researchers to be more efficient when writing software to SHMAC.

The Amber processor tile was upgraded to support the ARMv4T ISA before porting. Once Linux was ported to this new processor tile, a toolchain was generated and a large set of user applications was built.

The final result of this project is a familiar Linux environment with over 200 UNIX applications made available through a standard UNIX shell.

Sammendrag

I flere år har det vært mulig å forbedre prossessorytelsen ved å ta i bruk den stadig økende transistortettheten. I de siste årene har strømkravet til prosessorene overgått strømbudsjettet deres, så det er ikke lenger mulig å benytte alle delene av prosessoren samtidig. Den heterogene prossesorarkitekturen er foreslått for å overkomme dette problemet.

SHMAC-prosjektet tilbyr et forskningsrammeverk for heterogene prosessorarkitekturer, hvor forskjellige maskinkomponenter fungerer som utskiftbare moduler. For tiden er den ARMv2a kompatible Amber kjernen den eneste prosessormodulen tilgjengelig.

Det eneste operativsystemet tilgjengelig for SHMAC er det eksperimentelle forskningsoperativsystemet Barrelfish. Som et resultat av dette kjører mesteparten av koden skrevet for SHMAC uten noe operativsystem, direkte på maskinvaren. Siden maskinvaren til SHMAC ofte blir byttet ut vil jobben med å holde programvaren kjørbar være tidkrevende, og gjøre programvareutviklingen til SHMAC mindre effektiv.

Denne rapporten presenterer arbeidet gjort for å tilpasse Linux-kjernen til å kjøre på SHMAC. Linux kjernen gir programvare et kraftig abstraksjonslag som lar utviklerne være mer effektive når de skriver programvare til SHMAC.

Amber prosessormodulen ble oppgradert til å støtte ARMv4T instruksjonssettarkitekturen før Linux-kjernen ble modifisert. Da Linux var modifisert til å kunne kjøre på denne nye prosessormodulen ble det laget et verktøysett for å lage kjørbare programmer, samt et stort sett med brukerprogrammer.

Det endelige resultatet av dette prosjektet er et kjent Linux-miljø med over 200 UNIX programmer, tilgjengeliggjort gjennom et standard UNIX-skall.

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 Lasse Natvig and Asbjørn Djupdal for all help with the technical work and the report. Thanks to Anders Akre and Sebastian Bøe for help with reusing the testbench from the Amber project, for providing us with an improved multiply unit, and for giving us an accelerated processor tile that could be used when performing the benchmarking necessary for this project. Thanks to Stian Hvatum and Terje Runde for help with proofreading the thesis. Also thanks to Benjamin Bjørnseth for technical assistance.

Hello Tobias!

Problem Description

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 multi-core 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 port a recent version of Linux to run on SHMAC. As discovered in the autumn project conducted by Håkon Amundsen and Joakim Andersson, the SHMAC hardware needs some modifications to support Linux v2.6 and later. This project will therefore involve both Verilog design and Linux kernel programming.

Main tasks included in the project are:

- Implement support for the ARM v3 ISA in the Amber SHMAC CPU tile.
- Create a Linux port for the new CPU tile.
- Test and benchmark individual components and the finished system

If time permits, the students can also:

- Enable multiprocessor support.
- Investigate possibilities for mass storage support through the SHMAC host driver.

Contents

List of Tables

List of Figures

Abbreviations

AAPCS ARM Architecture Procedure Call Standard.

ABI Application Binary Interface.

AIC Amber Interrupt Controller.

ALU Arithmetic Logic Unit.

CPSR Current Program Status Register.

FIQ Fast Interrupt Request.

FPGA Field Programmable Gate Array.

ILP Instruction Level Parallellism.

IRQ Interrupt Request.

ISA Instruction Set Architecture.

LDT Linux Device Tree.

MMU Memory Management Unit.

MPU Memory Protection Unit.

PC Program Counter.

PSR Program Status Register.

SHMAC Single-ISA Heterogeneous MAny-core Computer.

SMP Symmetric Multiprocessing.

SoC System on Chip.

SPSR Saved Program Status Register.

TLP Thread Level Parallellism.

Chapter 1
Introduction

During the lifespan of the computer processor, various techniques have been used to increase its performance. These techniques have helped improved the performance of the processor 25,000 fold [22]. As the processor evolves, some of these techniques stop being efficient, and new limitations are discovered. This provides the computer architects with ever new challenges. In recent years, the biggest problem facing the computer architects has been related to power consumption.

1.1 Computer Architecture Trends

From the introduction of the RISC architecture in the early '80s to the late '90s, the most important fields of improvement for processors has been Instruction Level Parallellism (ILP) and memory system techniques [22].

ILP refers to the potential overlap among machine level instructions. There are several ways of exploiting this overlap. Processor pipelining, out of order execution, branch prediction and superscalar execution are all techniques that increase processor performance by exploiting the ILP in the intsruction stream [25, 22]. Memory system techniques aim to decrease the latency and increase the bandwidth of instructions and data from memory to the processor. These two fields of improvement have worked hand in hand with Moores Law, which states that the number of transistors available for a processor doubles every 18 to 24 months [22]. This allows the researchers to simply add wider buses, more registers, longer pipelines, more advanced branch predictors and generally making the processor more complex.

In the early 2000s, extracting any more ILP from the instruction stream reached a point of diminishing returns. The processors had become too complex, and their power consumption was too much for the available cooling systems to handle. In 2004, Intel canceled all of its high-performance uniprocessor projects as a result of this. They decided to focus all attention on the new architecture trend, multiple processor cores per chip [22].

The *multiprocessor* architectures exploit the Thread Level Parallellism (TLP) in the instruction stream. By looking at typical workloads for a processor, it is

evident that several of the execution threads are independent of each other. These independent threads can be executed in parallel on two or more separate processing units, allowing an increase in performance. TLP was exploited by including several processor cores on a single chip.

One of the reasons why it was now possible to have multiple processor cores on a single chip, was the increase in transistor density. This scheme relied not only on Moores Law, but also on Dennard Scaling, which states that as number of transistors per unit area increase, the power consumption remains constant [20]. Unfortunately, the Dennard Scaling stopped applying, while Moores law continued [21]. The power usage per unit area increased as the transistor density increased, making it difficult to keep the processor temperature at an acceptable level. Consequently, it was no longer possible to utilize all parts of the processor at the same time. This effect is known as "dark silicon".

One of the suggested strategies for mitigating this effect is the use of heterogeneous multicore processor architectures, which contains several processing elements with different performance and power characteristics.

1.2 SHMAC

The SHMAC project is an effort from NTNUs Energy Efficient Computing Systems (EECS) strategic research area to investigate challenges posed by heterogeneous computer architectures [12]. It proposes a tile based architecture of generic computing units connected in a network. There exists different types of computing units. At the time of writing there exists processor tiles, memory tiles and communication tiles. The current processor tile is a slightly modified version of the open source Amber processor core, which supports the ARMv2a Instruction Set Architecture (ISA). This ISA was used in the Acorn computers during the late 80s and early 90s [1, 2, 3, 4]. Using an ISA this old impose restrictions to what software can be compiled and run on SHMAC. A Field Programmable Gate Array (FPGA) prototype of this architecture has been developed [26].

1.3 SHMAC Operating System Project

This master thesis builds on the work described in the report "SHMAC Operating System" [17]. The project was an effort to port an operating system to run on the SHMAC hardware. It investigated several candidates for porting before deciding that Linux should be the target operating system. The 2.4.28 version of the Linux kernel was ported.

The ported kernel executed on one core, and was able to boot, mount the root filesystem and execute a single userland application with a limited set of working system calls.

The Amber core used in this project supported the ARMv2a ISA, this project faced several problems as a consequence. During this project, it was found that this ISA was not supported in any recent versions of the Linux kernel. Linux 2.4 was the last kernel version to include support for the ARMv2a ISA, which is why this was the Linux version used. The main research field in the SHMAC project is multicore architectures, however it was found that the Linux 2.4 kernel did not scale to multicore environments. The ARMv2a support in the Linux kernel was in such a state that performing ports to systems with this architecture was cumbersome. The toolchain used in this project had several shortcomings and put severe restrictions on software compatibility.

The proposition of the project was to modify the Amber processor tile to support a more recent version of the ARM ISA, allowing more recent Linux versions to be ported.

1.4 Assignment Interpretation

The following tasks were interpreted from the assignment text:

Mandatory:

T1 Modify the Amber processor tile so that it supports the ARMv3 ISA.

T2 Port a recent version of the Linux kernel to this new processor tile.

T3 Provide a toolchain which is compliant with the ported version of the Linux kernel.

T4 Test and Benchmark the new processor tile and the ported version of the Linux kernel.

Optional:

T5 Build a minimal Linux distribution capable of user interaction.

T6 Enable multicore support in the ported Linux kernel.

T7 Investigate the possibility of mass storage through the SHMAC host controller.

The tasks **T1** and **T2** have been directly extracted from the assignment text. The task **T3** has been interpreted from the assignment text and made mandatory since, although user applications are not part of the Linux kernel, a Linux kernel without the ability to run user applications would not be a usable system. Also, a Linux version without the ability to run user applications would be very hard to test. The task **T5** has been added as an optional task to increase the productivity of SHMAC developers by providing a familiar environment to run applications in.

This makes it easier to run programs in an interactive environment and to run a set of applications on different hardware configurations. The tasks **T6** and **T7** are optional and something to be looked at when the Linux port was considered to be working correctly.

1.5 Contributions

This project provides the SHMAC project with a new processor tile, Rav, compatible with the ARMv4T ISA. The Linux 3.12.13 kernel has been ported to support SHMAC running a single ARMv4T compatible processor tile. A toolchain which is fully compliant with the ported Linux kernel is provided, alongside a large set of applications which has been built using this toolchain. A complete Linux distribution containing a large set of standard UNIX tools and an interactive shell has been built to provide a familiar Linux experience. Tools and documentation are provided to make it easy to install and run Linux on SHMAC.

1.6 Report Outline

Chapter 1: Introduction gives an introduction to the motivation behind heterogeneous computer architectures. It presents the SHMAC project, assignment interpretation and a summary of contributions.

Chapter 2: Background introduces concepts related to this project.

Chapter 3: Rav gives a detailed description of the process of modifying the Amber processor tile to support the ARMv4T instruction set architecture.

Chapter 4: Porting Linux to SHMAC describes the process of porting the Linux 3.12.13 kernel to SHMAC.

Chapter 5: Userland Toolchain presents the process of building a toolchain for the ported Linux kernel.

Chapter 6: Building a Linux Distribution describes the process of integrating BusyBox with the ported Linux kernel in order to provide a complete Linux distribution.

Chapter 7: Benchmarking contains a set of benchmark results.

Chapter 8: Discussion presents a detailed discussion about the project process, the choices made and the results achieved.

Chapter 9: Conclusion provides concluding remarks.

Chapter 10: Further Work gives a set of suggestions for further work on this project.

Chapter 2
Background

This chapter introduces information required to understand the work done in this project.

2.1 SHMAC

SHMAC is a hardware prototype of a Single-ISA Heterogeneous MAny-core Computer [12]. It is an ongoing research project within the Energy Efficient Computing Systems (EECS) research area at the Department of Computer and Information Science, and Department of Electronics and Telecommunications at NTNU. The goal of the SHMAC project is to propose hardware and software solutions for future computer systems in order to tackle the Dark Silicon effect.

2.1.1 SHMAC Architecture

The SHMAC architecture specifies a grid of generic computing tiles, as seen in Figure 2.1. The tiles are connected in a 2D mesh network topology.

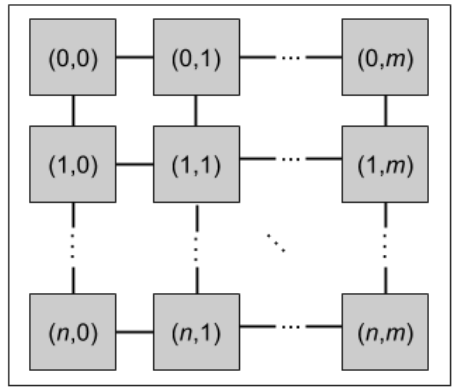

Figure 2.1: SHMAC tile architecture overview.

It is required that all of the tiles are able to forward packets to other tiles. This forwarding is performed by a router which is included in all SHMAC tiles, see Figure 2.2

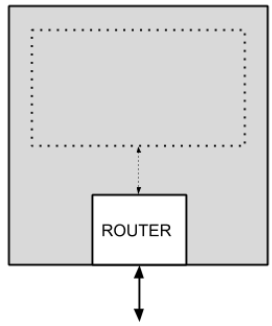

Figure 2.2: Architecture of a generic SHMAC tile.

As previously stated, each of the tiles can be made out of different hardware. Currently, the following tile types are available:

- Communication tile.
- Block RAM, l6 kB.
- External RAM, 32 MB.
- Amber25 processor tile.

The communication tile deals with the serial communication with the host. The block RAM provides memory which is implemented on the FPGA, whereas the external ram tile integrates a larger, slower, off chip memory. The block RAM is small and fast while the external RAM is large and slow.

The SHMAC architecture uses a 32-bit address space to provide access to the its various components, as seen in Figure 2.3. It should be noted that only a subset of the available addresses is mapped to actual devices.

2.1.2 Amber Processor Tile

The Amber processor tile is based on the five stage pipeline version of the Amber Core, described in section 2.3. The Amber core is slightly modified to be compatible with the SHMAC architecture. Since the Amber project is optimized for synthesis, it does not include a reset signal. This is added to the SHMAC version. The interrupt controller has been updated into using a generic bus for its sources, not one line for each source as was the case for the original Amber core.

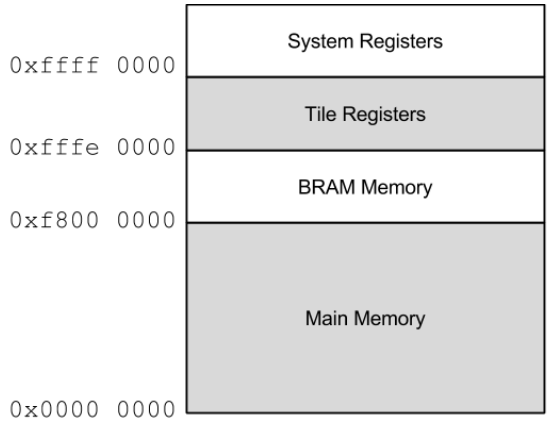

Figure 2.3: SHMAC memory map.

2.1.3 SHMAC Development Environment

The SHMAC prototype is implemented on a Xilinx Virtex 5 XV5VSX330 FPGA located on an ARM RealView Versatile platform. This platform contains an FPGA and a host controller. This host controller runs a Linux distribution, and provides the developers with an interface to the SHMAC hardware through a set of kernel modules, described below.

shmac_program write a file to SHMAC memory.

shmac_dump dump SHMAC memory to a file.

shmac_reset reset the SHMAC hardware.

shmac_ko exposes a serial device for communication with the SHMAC hardware.

2.2 ARM Instruction Set Architecture

An Instruction Set Architecture (ISA) defines the interface between the hardware and the lowest-level of software [25]. Since its first use in the Acorn computers introduced in the '80s, the ARM Instruction Set Architecture has seen widespread use throughout different markets segments [5]. Over the years the architecture has been upgraded several times. This section will present the changes in two of the upgrades of the ARM ISA, from ARMv2a to ARMv3 and from ARMv3 to ARMv4.

2.2.1 Comparing the ARMv2a and ARMv3 ISA

The ARMv2a ISA was used in the ARM2 and ARM3 family of processors which was used by Acorn Computers in their Archimedes line of computers [23]. The ARMv2a ISA is a load store architecture with 16 general purpose 32-bit registers. One of these registers is the Program Counter (PC) register. The program counter value is the memory address of the next instruction to be executed, and is incremented each cycle or modified by instructions. In addition to containing the program counter, the PC register also keeps track of the execution state of the processor, as seen in Figure 2.4. This execution state consist of 8 bits, leaving 24 bits for the program counter value. The program counter is word aligned, this implies that the program counter has a 26-bit address space. The 8 bits used for execution state is 4 bits for status flags, 2 bits for the current execution mode, and 2 bits for interrupt masking.

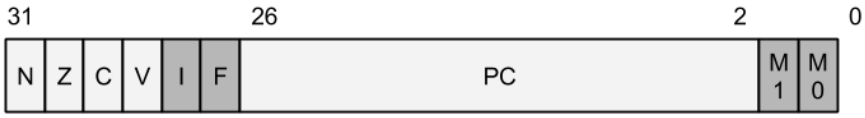

Figure 2.4: The 26-bit PC scheme of ARMv2a.

The status flags are described in Table 2.1. These are set by the Arithmetic Logic Unit (ALU) output only when the executed instruction explicitly states that the status bits should be updated.

Identifier	Meaning
N	**N**ot Equals
Z	**Z**ero
C	**C**arry
V	o**V**erflow

Table 2.1: ARMv2a status flags.

ARMv2a has 4 execution modes: supervisor, interrupt, fast interrupt and user. User mode is the only non-privileged execution mode. The other modes are exception modes and are privileged. The current mode is stored using the two least significant bits of the PC register, M1 and M0, as seen in Figure 2.4, using the values from Table 2.2. Each privileged mode have their own banked link register and stack pointer. This means that the value read when loading the link register or stack pointer differs for each of the modes, as shown in Figure 2.5.

The interrupt mask bits makes it possible to disable/enable interrupts at any given time. Privileged modes are able to freely change mode and interrupt masks while user mode is prevented from this. The interrupt mask bits are also updated by

the processor itself when context switches takes place. As an example, when an Interrupt Request (IRQ) is triggered, the IRQ mask is set to **1** automatically, ensuring that no other interrupts take place while the current interrupt is being handled.

M[1:0]	**Mode**
00	User
01	Fast Interrupt Request (FIQ)
10	IRQ
11	Supervisor

Table 2.2: ARMv2a processor mode bits.

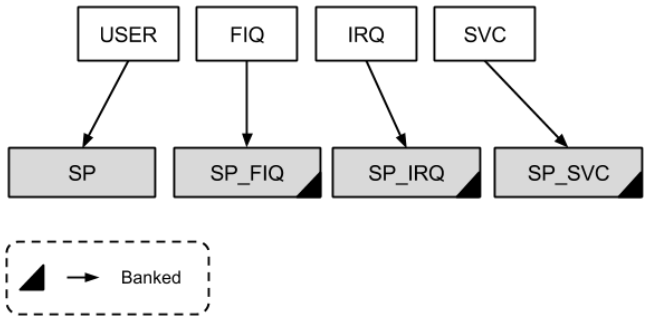

Figure 2.5: Banked register overview.

In order to modify the status bits, the `tstp` and `teqp` instructions are used. These instructions perform an AND or XOR compare operation, respectively, and store the status bits in the PC register. An example of this can be seen in Figure 2.6, where the mode is changed, the mask bits are set, and the condition flags are controlled.

The 26-bit program counter scheme puts some restrictions upon further development. Having 26 bits for addressing instructions limits the program counter to only address 64 MB of memory. Also, increasing the amount of bits used for storing the current execution mode, status bits, or interrupt mask bits would decrease size of the program counter.

When ARM designed ARMv3 they split the 26-bit PC scheme into a 32-bit PC register and a separate Current Program Status Register (CPSR) used for storing the status bits, as seen in Figure 2.7.

In addition they added a Saved Program Status Register (SPSR) which saves the CPSR when the processor performs a context change. The SPSR register is banked for each exception mode. These registers are not general purpose and can only be accessed by the new instructions `mrs` and `msr`, provided for reading and writing

```
/* Switch to IRQ Mode */
mov     r0, #0x00000002
teqp    pc, r0

...

/* Switch to User Mode */
/* and unset interrupt mask bits */
mov     r0,   #0x00000000
teqp    pc, r0

...

/* Check that the condition flags are still 0xf */
mov     r4, pc
and     r7, r4, #0xfc000000
cmp     r7, #0x08000000
movne   r10, #70
bne     testfail
```

Figure 2.6: Examples of PC usage in ARMv2a.

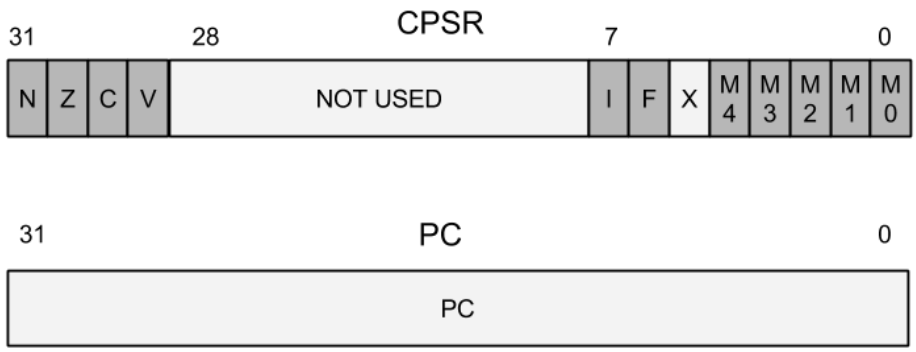

Figure 2.7: The CPSR scheme of ARMv3.

the status bits registers. ARMv3 also adds two new exception modes: “abort” and “undefined”.

The Address Exception

The address bus in ARMv2a is 26 bits wide [23]. When calculating memory addresses, it is therefore important to take care not to produce illegal addresses in which one of the top six bits is non-zero. This will trigger an Address Exception.

In the case of ARMv3, any 32-bit value is a legal memory address, hence there exists no Address Exception in ARMv3 or later ARM ISAs.

2.2.2 Comparing the ARMv3 and ARMv4 ISA

ARMv4 extended ARMv3 with half word data transfer, signed data transfer, long multiply and a new execution mode.

Half Word Data Transfer

Since high level programming languages use data structures of different sizes, it is desirable to be able to provide instructions which allows you to fetch the correct amount of bits from memory. The default behavior in the ARM architecture is to fetch an entire word (32 bits). ARMv2a and ARMv3 also supports byte transfer (8 bits). ARMv4 adds support for half word (16 bits) data transfer. This is made possible by the `ldrh` instruction for loading a halfword from memory, and `strh` for writing a halfword to memory.

Signed Data Transfer

When loading a half word sized or byte sized value from memory, it will be placed in a 32 bit register. This has implications for the transfer of signed values, since they need to be sign extended in order to retain their value. ARM represents signed values as two's complement. In the two's complement scheme, the most significant bit is used to tell if the value is positive or negative. When extending a value represented as two's complement, the most significant bit needs to be repeated in the extra bits, as seen in Figure 2.8

```
1  /* Signed representation of the value
2     10000 using two's complement using 16 bits*/
3  0010 0111 0001 0000
4
5  /* Sign extended representation of the value
6     10000 using 32 bits */
7  0000 0000 0000 0000 0010 0111 0001 0000
8
9  /* Two's complement  representation of the
10    value -10000 */
11 1101 1000 1111 0000
12
13 /* Sign extended representation of the value
14    -10000 using 32 bits */
15 1111 1111 1111 1111 1101 1000 1111 0000
```

Figure 2.8: Sign extending values represented as two's complement.

ARMv4 adds two new instruction, `ldrsb` and `ldrsh`, to automate this process for byte values and half word values respectively.

Long Multiply

Both ARMv2a and ARMv3 supports multiplying the 32-bit values of two registers. The result of such a multiplication will often be bigger than what it is possible to represent using 32 bits. In the case of ARMv2a and ARMv3 the result is stored in a single 32-bit register. If the result is bigger than what it is possible to represent with 32 bits the value is truncated.

ARMv4 provides long multiplication in which the result is stored in two separate 32-bit registers, making it possible to store the result of any 32-bit multiplication.

System Mode

ARMv4 adds the new privileged execution mode, "system". While the other privileged modes all have banked registers, the system mode share all of its registers with the user mode. The only difference between the unprivileged user mode and the privileged system mode is that system mode is able to change the current execution mode.

2.2.3 ARMv4T Extension

The T-extension to ARMv4 adds the 16-bit Thumb execution state. This execution state implements a subset of the ARM instruction set using 16-bit representation. A new instruction, `bx` (Branch and Exchange), is added in T-variants of ARMv4, and is used to switch between regular 32-bit execution state and the 16-bit Thumb execution state.

2.2.4 ARM Architecture Procedure Call Standard

When a procedure calls another procedure, they need to agree on how the arguments will be transferred, how the result of the called procedure will be stored, and what state the called procedure should leave the system in when it is done.

Arguments can be transferred in several ways, using the stack or by using the registers. In order to make it possible for separately compiled and assembled code to work together, a standard for transferring arguments needs to be defined, this is the ARM Architecture Procedure Call Standard (AAPCS) [19].

The AAPCS defines how registers are used during a procedure call, as seen in Table 2.3. It also defines how to return from a procedure call.

The Role of Branch Exchange in AAPCS

The AAPCS states that the `bx` instruction should be used when returning from a procedure [[19],p.21]. `bx` takes a register as argument, and jumps to the address specified by that register. If the last bit of that address is **1**, the processor should change to Thumb execution state (16-bit instruction width). Even though this

Register	Purpose
r0-r3	Holds the arguments passed to a sub procedure. r0 also holds the result from the subroutine.
r4-r11	Holds local variables. These registers needs to be retained during the procedure call, which is done by storing them on the stack while the procedure is ongoing.
r12	The Intra-Procedure-Call scratch register.
r13	The stack pointer.
r14	The link register.
r15	The program counter.

Table 2.3: AAPCS register convention.

instruction is heavily associated with the Thumb instruction set, it is also used on processors without support for Thumb instructions. In the case of ARMv4, which does not support the `bx` instruction, a GCC flag, `fixv4bx`, is provided in order to change all of the `bx` calls into equivalent `mov` calls. ARMv4 is therefore incompatible with the AAPCS. In ARMv5 or later, trying to change to Thumb execution state when this is not supported will trigger an undefined exception.

2.3 Amber

Amber is an open source project which contains a complete embedded computing system. This system consists of the Amber processor core, hereby called the "Amber core", and a set of peripherals as shown in Figure 2.9.

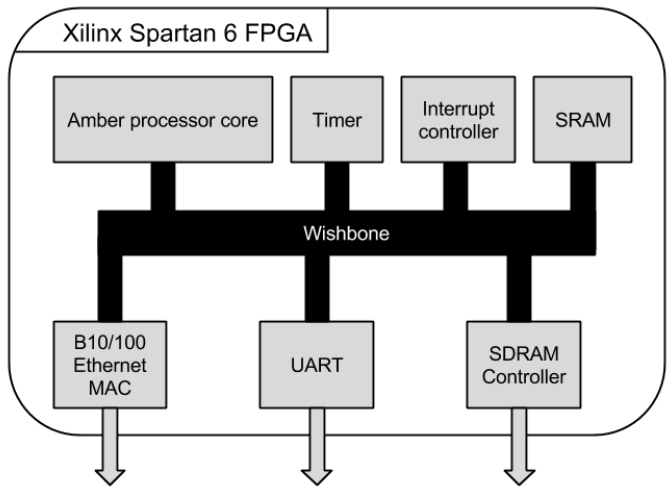

Figure 2.9: Amber system overview.

The Amber project focuses on FPGA synthesis rather than hardware implementations. An example of this is that the Amber cores does not have any reset logic, since this is not necessary in an FPGA environment. The peripherals makes it possible for the user to interact with the system, and to execute applications of a relatively large size. A brief explanation of the peripherals most relevant for this project is given below.

UART deals with communication with the users of the system.

Timer generates timed interrupts.

Interrupt Controller is a general purpose interrupt controller which is able to send interrupts from various modules.

SDRAM Controller contains the memory of the system.

2.3.1 Amber Core

The Amber project contains two versions of the Amber core: one with a three stage pipeline and one with a five stage pipeline, called *A23* and *A25* respectively. The Amber processor tile contains a modified version of the five stage pipeline, A25.

Pipeline Design

The Amber core is a scalar, in-order processor core with a classical five stage RISC pipeline [22]. The pipeline stages are fetch, decode, execute, memory and write-back.

Fetch reads the instruction to be executed from memory and forwards it to the decode stage.

Decode looks at the instruction, and decides what needs to be done in the remaining pipeline stages to perform the tasks requested by the instruction. This is arguably the most complex of the five stages, as it needs to be able to parse all legal instructions, and set all of the requested control signals.

Execute performs the actions which is required to change the register values according to the instruction being executed. It is controlled by the decode stage, and is also used for calculating addresses which is sent to the memory stage. In the case of the Amber core, the execute stage contains the register bank. This implies that the register values are updated at the end of the execute stage. Since most instructions read some value from a register, the output from the register bank is wired to the input of the execute stage, creating a circular data flow, seen in Figure 2.10

Memory reads data from or writes data to the memory.

Write-Back saves the data read from memory to the register bank.

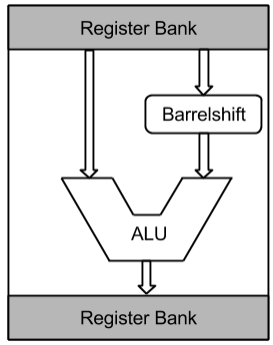

Figure 2.10: Conceptual placement of register bank in execute stage.

Instruction Set Architecture

The Amber core supports the ARMv2a instruction set architecture. This makes the Amber core compatible with the GNU toolset, and old versions of Linux (ARMv2a support was removed in version 2.6).

2.3.2 Test Framework

When designing hardware it is important to perform testing early in the development process. Once the design has been placed on an FPGA it is cumbersome to execute tests and verify the result, hence it is preferred to tests the design in a simulator. In a simulator, the state of the entire system is accessible.

Instruction Tests

An instruction test is a test that verifies that the processor behaves correctly for a single assembly instruction. The Amber project contains 59 instruction tests. The tests are executed using the Xilinx ISE simulation tool. Xilinx tools are also used for compiling the Amber system and initializing the memory to be used during the tests.

The test framework is able to execute the same set of code as the FPGA implementation of the Amber system. This is achieved by using a top level design which is developed specifically for simulation.

The test framework contains several scripts which makes it simple to execute a set of tests. These scripts also makes it easy to extend the existing test framework with new tests.

System Tests

In addition to the above-mentioned instruction tests, the Amber project contains a set of system tests.

The biggest system test contained within the Amber project is a ported version of the Linux 2.4.28 kernel, which is compatible with the Amber core. When verifying a processor design, it is not enough to verify each of the supported instructions in an isolated environment. Some bugs might not be triggered unless the processor is in a specific state which might only be triggered by a specific sequence of instructions. It is hard, if not impossible, to cover all possible code sequences using a set of small test programs. However, by running an application as big as the Linux kernel the confidence in the correctness of the design is increased.

2.4 Operating Systems

The operating system has been an important part of computers since its inception. The variation of complexity and size between operating systems, 45 million lines of code in Windows XP against 16,500 in FreeRTOS, suggests that the term "operating system" is a vague definition.

2.4.1 Operating System Definition

An operating system is a layer of software whose job is to "*provide user programs with a better, simpler, cleaner, model of the computer and to handle managing all the resources...*" [29]. In other words, an operating system is not required to interact with the *user* of the system, but with the *user programs*. However the term operating system is often used to describe system that do interact with the user, *e.g* Microsoft Windows and Apple OS X.

The motivation behind an operating system is twofold. It manages hardware complexity by providing applications with a simpler and cleaner interface. It also handles resource sharing in the system, allowing several programs to execute simultaneously on the same system.

2.4.2 Terminology Used in this Thesis

In the thesis, the terms "Linux", "Linux kernel" and "Linux operating system" are used to refer to the Linux kernel. The term "userland" refers to all code that runs outside of the kernel. The term "Linux distribution" refers to a system with the Linux kernel and a set of userland applications which enables direct user interaction.

2.5 Linux

Linux is an open source operating system kernel which has been ported to a large set of computer architectures. It is used in various computing domains, with its highest popularity in the web server and supercomputer domains [13, 15].

2.5.1 Linux Distribution Overview

The Linux kernel provides abstractions and services to user *applications*. The term Linux, however, is often used to describe a Linux distribution. A Linux distribution is the Linux kernel together with a set of applications and tools that provides abstractions and services to the *user*.

A Linux distribution is a layered system where each layer provides services to the layers above it while relaying on the services provided by the layer below it, as seen in Figure 2.11. The Linux kernel runs on top of the hardware, and is the only piece of software which runs in privileged mode. The library layer runs on top of the kernel and provides access to the kernel system call by wrapping system calls in library procedures. This enables applications to call system calls as if they were regular library routines. Which applications are included in a Linux distribution varies between distributions, from only a shell and some basic utilities to full fledged GUI systems for use with personal computers.

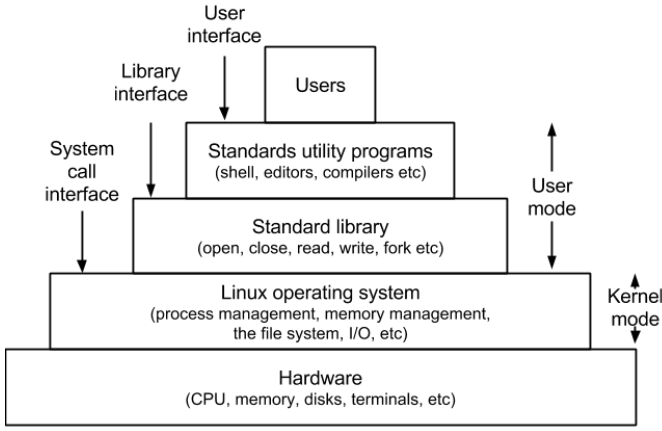

Figure 2.11: Linux distribution layers. (From [29])

2.5.2 Linux Kernel Overview

The kernels job is to manage the hardware resources in the system and expose access to these resources through a simplified interface. The simplified interface that the kernel exposes is the *system call interface*. Internally, the kernel is divided into three major components, as seen in Figure 2.12. The *memory management component* is responsible for managing the memory available in the system by providing virtual memory and paging functionality. The *process management component* is responsible for providing the process abstraction, scheduling which process should run and handling interprocess communication with signals. The *I/O component manages access* to all the input output devices in the system through a unified virtual file system known as the root file system. Resources are made available as files in the root file system.

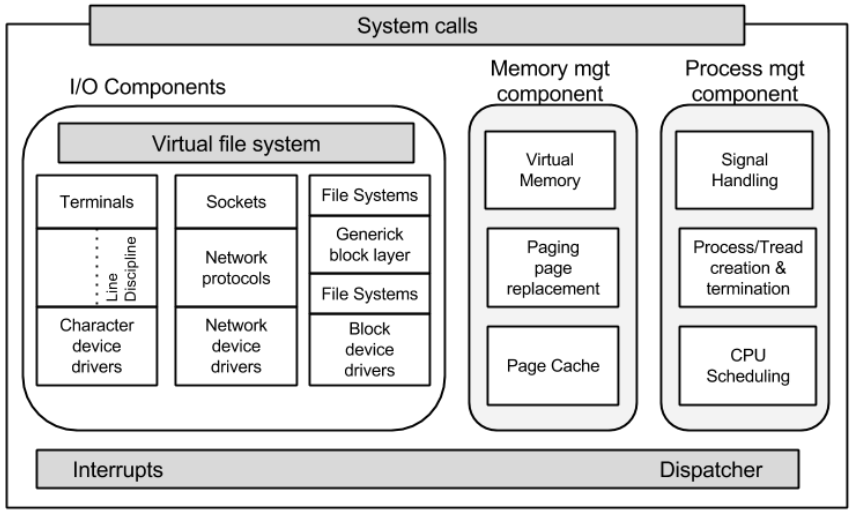

Figure 2.12: Linux kernel components. (From [29])

2.5.3 Porting Linux

The initial versions of Linux were not portable [30]. However, during its lifetime, Linux has come to be one of the most portable and ported operating systems.

For an operating system to be portable, the job of modifying it so that it is able to execute on a previously unsupported computer architecture should be as simple as possible. There are two perspectives one can use when talking about portability.

The first of these two perspectives is that of *hardware specific code*. The number of places where hardware specific code is found should be limited to a bare minimum, as Figure 2.13 shows.

The code in Figure 2.14 is an example of hardware specific code. This code excerpt shows the function used for turning off a single interrupt within an interrupt controller. The offset from the memory base of the interrupt controller to the control signal used for clearing interrupts, `IRQ_ENABLE_CLEAR`, would vary between different interrupt controller chips. The base address used for the interrupt controller device could vary between two architectures that used the same interrupt controller device.

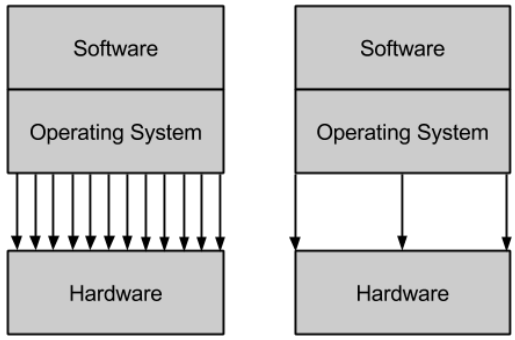

Figure 2.13: Operating system hardware touch points.

```
#define IRQ_ENABLE_CLEAR  0x0c

/* Mask (turn off) an interrupt */
static void irq_mask(struct irq_data *d)
{
    /* Find the HW-IRQ-nr by looking at the domain */
    unsigned int irq = irqd_to_hwirq(d);

    unsigned int pos = ffs(irq)-1;

   /* Write to IRQ control register to mask interrupt */
    writel((1 << pos),base + IRQ_ENABLE_CLEAR);
}
```

Figure 2.14: Example of hardware specific code.

In addition to the hardware specific code perspective, the *upward interface* from the hardware specific code is important. This interface decides how the other operating system components interact with the hardware specific code. By making sure that this interface is well defined, it is possible to use a single implementation of a higher level module with any implementation of this interface, Figure 2.15 illustrates this.

Porting an operating system quickly becomes cumbersome if care is not taken when designing it. Linux is highly portable, it has a layered design which ensures that hardware specific code is limited to the device drivers, and as long as they are compliant with the standard driver interface, no change is required in any higher level modules.

Linux Device Drivers

When performing a port, a task that needs to be performed no matter how portable the operating system, is to write drivers for all unsupported hardware. Linux

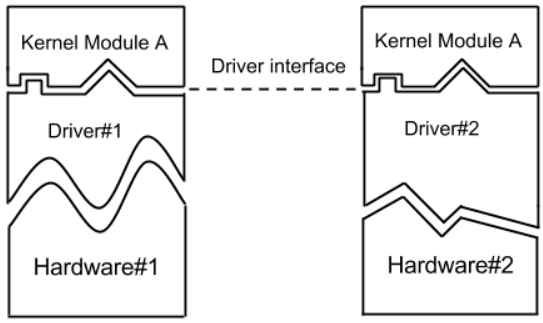

Figure 2.15: Operating system driver interface illustration.

provides a standard interface for most domains of hardware devices.

It is not required for a driver to implement all of the functions in the interface. What functions needs to be implemented is determined by the configuration of the kernel. What drivers should be compiled and included in the kernel is also set in the kernel configuration. To choose which drivers that should be used during execution, and how these should be configured, Linux provides a functionality called *Linux Device Tree*.

Linux Device Tree

The Linux Device Tree (LDT) is a hardware description format used in the Linux kernel. The introduction of the LDT simplified the porting process. It also made it easier to build kernel binaries which were compatible with a set of hardware platforms instead of only one.

The LDT describes which hardware components are included in a specific architecture. The LDT information is separated from the kernel, so it is possible to modify this information without having to recompile the kernel. The only time one would have to recompile the kernel is to add support for new hardware.

The LDT is given to the kernel by the bootloader. As the Linux kernel initializes, it parses the entry in the LDT that corresponds to the architecture type it has discovered. Once the LDT has been parsed, a set of compatibility flags and required device drivers has been found. These are mapped against the drivers contained within the kernel, and the drivers which matches are loaded and initialized.

2.5.4 Writing Software for Linux

The services provided by an operating system are made available by the system call interface. In the case of the Linux kernel, there exists 380 system calls. To

utilize these system calls, a *toolchain* that is compliant with the target Linux version is required. Briefly explained, a toolchain is a set of tools which is used when transforming source code to executable files. In cases where the target architecture, the architecture that will execute the program, differs from the host, the one on which the code is being written and compiled, a *cross compiler* is required as a part of the toolchain.

Once equipped with a toolchain that is compatible with the target Linux version, it is possible to compile code that utilizes the Linux system calls. In addition to containing the header definitions of the Linux system calls, the toolchain needs to generate code which is compatible with the Application Binary Interface (ABI) supported by the target Linux version.

ABI is the interface between the operating system and the applications at the machine code level. It specifies the binary format of applications, the procedure call standard, and how system calls are performed.

2.6 The uClinux Project

Several embedded computing systems lack a Memory Management Unit (MMU), and can only use physical addresses directly in their code. This has major implications for the programming model. The uClinux project provided support for MMU-less system in the Linux kernel.

The uClinux project provided a set of patch files which made the Linux kernel portable to MMU-less systems. The two projects existed in parallel for a few years before the uClinux project was merged into mainline Linux.

In addition to the patch files for the Linux kernel, the uClinux project also contained tools for adapting userland applications to work on MMU-less architectures. The uClinux project also contained a standard C library implementation, *uClibc*, which was built specifically for MMU-less Linux systems with limited memory resources.

2.7 GNU Operating System

GNU was launched in 1983 and set out to offer a completely free UNIX-compatible operating system. Many computer users today run a modified version of the GNU system without even realizing it. This is because what many users consider to be "Linux" is in fact the Linux kernel together with the free GNU system. By the time that Linux was released the GNU project had put together almost a complete operating system. What they still lacked was a UNIX-like kernel. When the Linux kernel was released, the GNU project was combined with the Linux kernel to build a complete operating system known as GNU/Linux. The GNU project is still working on an operating system kernel while the combined system GNU/Linux has achieved major success.

2.8 BusyBox

While GNU/Linux is a popular operating system for desktop computers BusyBox/Linux is a good alternative for embedded systems. BusyBox combines tiny versions of the most common UNIX utilities into a single executable. The implementations of these utilities usually has less features than their GNU counterparts while still providing a very similar behavior. Since BusyBox targets embedded systems, it has been written with size-optimization and small memory footprint in mind. It is also customizable, and it is possible to configure which features should be included at compile time. BusyBox also support running on Linux without MMU support.

BusyBox provides both a shell and an entire execution environment, which enhances the Linux kernel with powerful user interaction that makes a system behave like a regular PC with the GNU/Linux terminal.

2.9 Toolchains

A toolchain is a set of software components used for building applications. A typical toolchain contains a compiler, a linker, binary utilities and an assembler. Libraries are often included to simplify the job of the software developers using the toolchain. The binary utilities contained within the toolchains provide helpful tools when working with binary files. With these tools, it is possible to transform high level language source code to executable files.

In the embedded world, the *target* architecture is often different from the *host* architecture. That is: the processor for which the application is written has a different architecture than the computer generating the executable application files. It is often necessary to build a toolchain specifically for a single system environment.

Building an entire toolchain can be a cumbersome process. There are often dependencies between the various components, restrictions as to what configurations they might have, and which versions that are compatible with each other. When building the toolchain, it is important that the components are configured such that they are compatible with each other, and with the ABI of the target system. There exists several build systems for simplifying the toolchain building process.

These build systems simplify the building and configuration process by providing a single configuration interface, and automating the build process. Most build systems also ensures that no dependencies are violated during the configuration of the toolchain. What these build systems provide varies, a brief description of some popular build systems is given below.

Buildroot
In addition to being able to generate toolchains, Buildroot has the capability of building operating system images, bootloader images, root file systems and applications for the target architecture [8]. This is done using a set of build

scripts and configurations for the GNU *make* utility. Buildroot does not have support for ARM architectures without an MMU [7].

PTXdist
The most feature rich build system evaluated for this project is PTXdist. PTXdist largely supports the same features as Buildroot, and uses the same tools to achieve this. Compared to Buildroot, PTXdist provides a simpler interface, and has support for MMU-less ARM architectures. PTXdist also lets the user provide patch files to apply to the Linux kernel before compiling, which is often necessary in the embedded world.

Crosstool NG
This is the simplest build system of the three evaluated. Crosstool NG only targets toolchain generation, and does not provide functionality for building Linux or bootloaders like Buildroot and PTXdist. It is simpler to use than both Buildroot and PTXdist. As with PTXdist, it supports MMU-less ARM architectures.

2.9.1 Standard C Library

In software development, a library is a collection of function implementations made available to the developer through a well defined interface. One of the most important C libraries is the *standard C library*. This library defines a set of functions expanding the basic C functionality. It contains functions that *"... provide input and output, string handling, storage management, mathematical routines, and a variety of other services for C programs"* [24].

Developing useful applications in C without using the standard C library could be difficult. For a C program to be able to utilize the functionality offered by the standard C library, the toolchain used when building the application must include an implementation of it. There exists several implementations of the standard C library.

The ANSI definition only specifies the standard C library interface and functionality, not its implementation. Depending on the hardware support on the target architecture, it might not be possible for a standard C library implementation to provide all functions specified in the standard C library interface. The features supported by a standard C library implementation can also depend on operating system support.

Standard C Library on Bare Bones Platforms

The term "bare metal" refers to a platform which does not have an operating system. Standard C Library implementations for use with bare metal target architectures are implemented using only standard C. One example of a bare metal implementation of the standard C library is *newlib* [10].

Standard C Library on Platforms with Operating Systems

Once a system is equipped with an operating system, the platforms interface towards any software executing on top of it will be enriched by the operating systems system calls. In the case of Linux, this means support for threads, processes, files, time, concurrency, kernel modules, devices and much more.

Standard C library implementations which utilizes system calls are tightly coupled with the operating system on which they execute, placing strict requirements to any toolchain that wishes to generate executable code.

An example of a standard C library implementation that can be connected to the Linux system calls is *uClibc* [14]. uClibc specializes in embedded platforms, and supports MMU-less Linux.

2.10 Multicore Operating Systems

Since the introduction of the multicore processors, operating systems have been adapted to fully utilize the processing power offered by a multicore architecture. Designing an operating system for a multicore architecture requires several design features not required for a single core operating system.

2.10.1 Symmetric Multiprocessing

The cores in the SHMAC architecture all support the same ISA and have shared memory. This makes it possible to use the SMP model. The SMP model places the operating system in shared memory, and allows for any core to run it [29]. Allowing all of the cores to access the operating system introduces synchronization problems. What is the result when two cores attempts to update the same variable simultaneously? This problem can be solved with inter core synchronization.

2.10.2 Locking Primitives

There are several ways to implement inter core synchronization, locking is a necessity for most of them. A lock protects an entity, and ensures that only one actor (core, computer or thread) can access this protected entity at a time. This entity could be anything from a piece of critical code, code which only one core can execute simultaneously, to an entire device, such as the hard drive. For this to be possible, it is important that the locking mechanism is *atomic*. This means that when an actor locks an entity, the system behaves as if it was an instantaneous operation. A lock such as the one described above, which can be either in locked or unlocked state, is known as a *mutex* [29].

Consider the situation in which several actors requires access to a locked entity. If they cannot get exclusive access, they will go to sleep, and try to get access again after some time. In a situation like this, it would be desirable to maintain a list of actors that want access to the locked entity, and wake them up explicitly once

it's their turn. A single mutex is only capable of ensuring exclusive access, not maintaining a queue such as this. The solution is to introduce a counter which can be atomically read, incremented and decremented. A counter like this is known as a *semaphore* [29].

2.10.3 SMP in Linux

Linux provides multicore support through SMP. A problem with using locks as a synchronization mechanism is that the entities under protection of a single lock can not be simultaneously accessed once that lock is taken. Hence it is desirable that the entity which a lock protects is as small as possible, while still ensuring correct execution with multiple cores. As described in [17], it was not until Linux version 2.6 that the size of the entities protected by the SMP locks was reduced to a scalable level.

Chapter 3
Rav

This chapter describes in detail the changes made to the ARMv2a compliant Amber core in order to make it ARMv4T compliant. The new core was named *Rav* - the Norwegian word for Amber.

3.1 Motivation

Upgrading the Amber core to support a more recent ISA would result in better support in software tools, simplifying the development process for SHMAC software. ARMv2a used a 26-bit program counter scheme, requiring special handling in any software package that was to support it. This was needed since the architecture had a 32-bit ALU and a 32-bit PC register of which only 24 bits is the program counter. Given that only a handful of computers actually used ARMv2a ISA, it becomes clear that it is unattractive to add support for this ISA in modern software projects.

ARMv3 was the first ARM ISA to move away from the 26-bit PC scheme, resulting in increased support by software tools. However, the AAPCS is not supported by ARMv3.

To make Rav compatible with the AAPCS, the target ISA was chosen to be ARMv4T without Thumb support.

3.2 Execution Stage Schematic

A simplified schematic of the final execution stage of Rav is shown in Figure 3.1. The gray boxes represents either modifications or additions to the execution stage that was added in Rav. The trapezoid shapes represents multiplexers, their control signals has been omitted when the control signal is named the same as the multiplexer. Some details have been omitted from the schematic to make it more readable. Information that has not been modified, *e.g.* co-processor, or would complicate the schematic, *e.g.* stalling logic is not included.

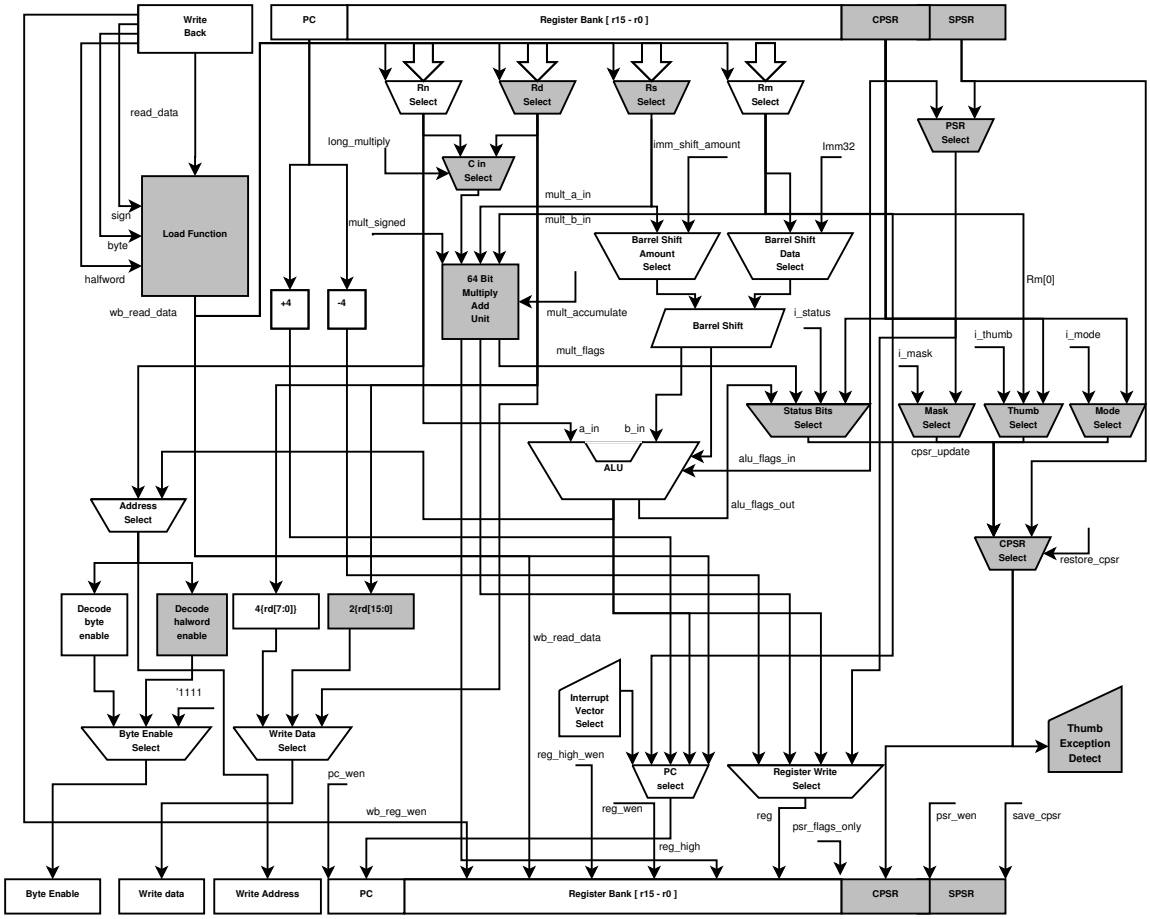

Figure 3.1: Rav schematic execution stage.

3.3 Implementing Rav with ARMv3 Support

The specification used for implementing ARMv3 support in Rav is the documentation for ARM7DI, an ARM7 family processor implementing the ARMv3 ISA [18]. This documentation was compared with that of the Amber core [27] to come up with a set of changes that needed to be implemented. These changes involved separating the status bits from the program counter and putting them in a special purpose register, introducing two new exception modes, and adding support for two new instructions to read and write the status bits registers. It was also necessary to add logic for when the status bits registers are updated during execution of programs, and exception handling to save and restore processor state. The Address Exception was removed since it cannot occur in ARMv3 or later versions. Rav was implemented by modifying the decode and execute stage of the five stage Amber processor tile.

3.3.1 Program Status Registers

A set of new register were added to contain the program status bits. The register bank was expanded by adding 6 Program Status Registers. One to hold CPSR and 5 to hold SPSR, one for each exception mode.

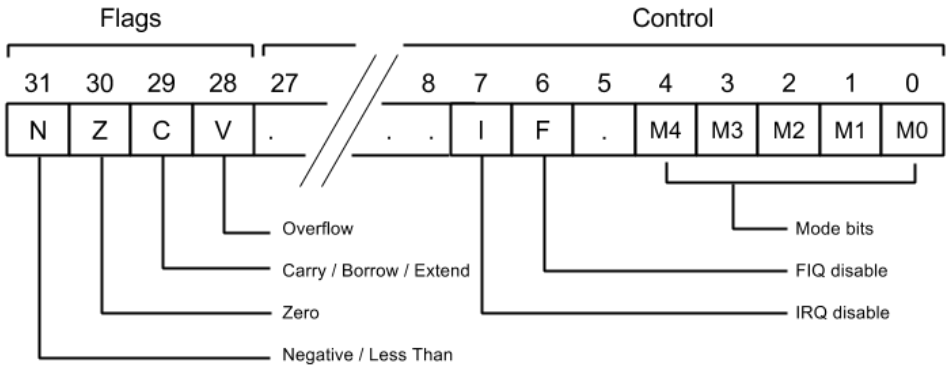

Figure 3.2: Program status register format. (From [18].)

The program status registers consists of status flag bits and control bits, as seen in Figure 3.2. The flag bits store information about previously executed instructions, and are used for conditional execution. A description of the different status flag bits is given in Figure 3.2. Any register operation is able to update the flags by having the S-bit set. The control bits consists of five bits to specify the current processor execution mode, as shown in Table 3.1, and two interrupt disable bits for normal and fast interrupts. The unused bits are reserved and cannot be written.

M[4:0]	Mode
10000	User
10001	FIQ
10010	IRQ
10011	Supervisor
10111	Abort
11011	Undefined

Table 3.1: ARMv3 processor mode bits.

3.3.2 New Processor Modes

ARMv3 adds two new exception modes to ARMv2a, undefined and abort. Undefined mode is entered when the processor finds an instruction it is unable to decode. Abort mode is entered whenever the processor receives a exception from either the instruction cache or the data cache. Both of these processor modes have their

own link and stack pointer register as well as a saved program status register. The resulting register bank now contains a total of 37 registers as seen in Figure 3.3, the gray registers are banked registers, and are exclusive to the mode.

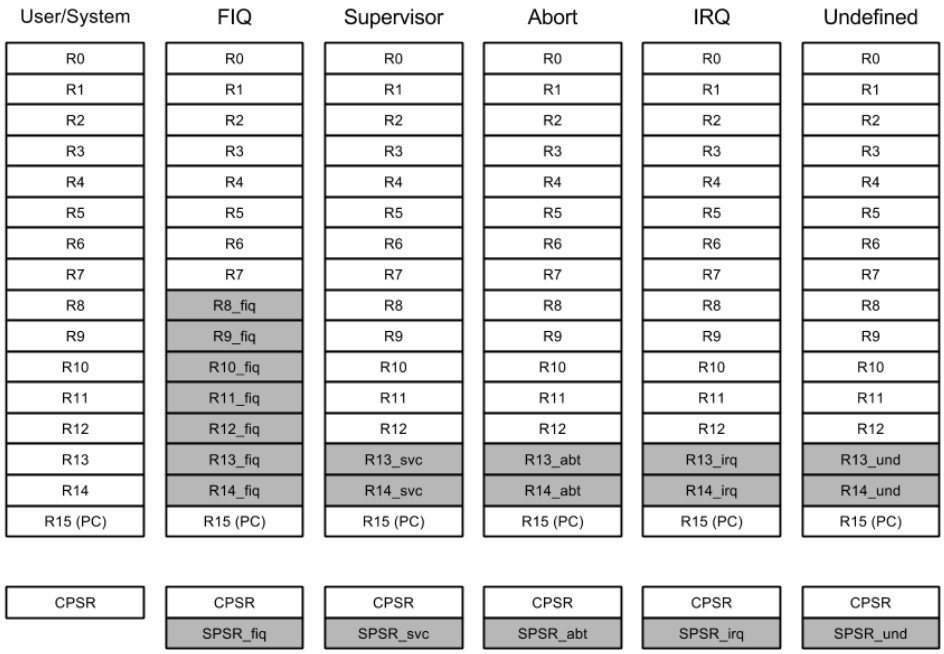

Figure 3.3: ARMv3 and ARMv4T register bank. (From [18].)

3.3.3 Program Status Register Transfer Instructions

The Program Status Registers (PSRs) can only be accessed through the `mrs` and `msr` instructions. The `mrs` instruction writes the content of either CPSR or SPSR_<mode> to a general register. The `msr` instruction writes either the value of a general register or an immediate operand into either CPSR or SPSR_<mode>. It is possible to specify that only the flag part of the target program status register should be updated when using `msr`. This behavior is forced in user mode.

The decode stage is responsible for decoding the current instruction and determining the proper control signals to give the execute stage. For an `mrs` (read PSR register) instruction, the decode stage will set the **psr_select** signal so that either CPSR or SPSR is chosen from the PSR Select multiplexer, see Figure 3.1. The **register_write_select** signal will be set to select the output from the PSR Select multiplexer as input to the register bank.

For an `msr` (write PSR register) instruction, the decode stage will set the **register_write_select** to select the **alu_out** value, since the value that should be

written to the selected PSR register is found here. The **psr_wen** signal will be set to 10 for writing to SPSR and 01 for writing to CPSR. If the msr instruction wishes to only update the status flag bits, or the instruction is running in user mode, the **psr_flags_only** signal will be set, causing only the status flag bits of the selected PSR register to be updated.

3.3.4 Execution Context

CPSR is constantly updated during execution to reflect the state of the processor. The signal **cpsr_update** holds a candidate for the updated value of CPSR. During normal execution, this value is written to CPSR every cycle. The individual parts of **cpsr_update** are controlled by four multiplexers. Each of these multiplexers has the default behavior of keeping the original value from CPSR. Alternatively, the new value can be set from the decode stage, using the **i_flags**, **i_mask** and **i_mode** signals.

Status Flags

The status flag bits are controlled by the Status Bits Select multiplexer. If the executing instruction is a register operation with the S-bit set, the new value of the status flags bits is selected from the **alu_flags_out** signal. If the executing instruction is a multiplication with the S-bit set, then the negative-bit (N) and the zero-bit (Z) are selected from the **mult_flags** signal, while the rest of the flag bits keep their original value.

Saving CPSR

When an exception is triggered, CPSR is updated. The new value of the mode part of CPSR will be set by the **i_mode** signal. In every exception mode triggered, interrupts (IRQ) will be masked to allow the exception handler to run without being interrupted. This is done by setting the value of **i_mode** to mask interrupts. If the exception is a fast interrupt (FIQ), then fast interrupts will also be masked. When an exception is triggered, the CPSR register needs to be preserved in the SPSR register. This is done by setting the **save_cpsr** signal to the register bank. This will save the value of the CPSR register into the banked SPSR register of the new exception mode.

Restoring CPSR

When returning from an exception handler, CPSR needs to be restored to the original value preserved in SPSR. When an exception return instruction is executed, the **restore_cpsr** signal will select the SPSR register value as the new CPSR value in the CPSR Select multiplexer. An exception return instruction can only be triggered in an exception mode. It is either a register instruction or a load multiple instruction that writes to the program counter with the S-bit set.

3.4 Implementing Rav with ARMv4T support

The changes from ARMv3 to ARMv4 includes: a new execution mode, supporting a new instruction, long multiplication, halfword memory access operations and signed memory access operations. The changes from ARMv4 to ARMv4T consists of a new branch instruction, a new bit in the PSRs, and triggering an undefined exception when trying to enter Thumb state.

3.4.1 System Mode

ARMv4 introduces the system execution mode. System mode shares its registers with user mode, and does not have any banked registers. It is a privileged mode, and is therefore able to modify the program status registers. Since it shares its registers with the user mode, adding support for system mode does not require as many changes as that of abort mode or undefined mode. The only change required is to allow system mode to modify the entire PSR.

3.4.2 Long Multiplication

ARMv4 adds long multiplication and long multiply accumulate instructions. These instructions use 32-bit registers as operands, and stores the 64-bit result in a pair of 32-bit registers, whereas ARMv3 stores the result in a single 32-bit register.

The Amber processor tile used an implementation of Booths multiplication algorithm to perform 32-bit multiplication [27]. This implementation consist of 33 cycles of shifts and additions. If it is a multiply accumulate instruction there is one extra addition cycle. This multiplication unit was replaced with an FPGA specific multiplication unit which is able to perform 64-bit multiplication in one cycle. Although the multiplication unit is able to do multiplication in one cycle, the new design was unable to meet timing constraints, and the clock frequency of the processor was lowered. The multiplication unit was therefore changed to do multiplication in two cycles instead, in order to maintain the processor clock frequency.

The Amber processor tile was only able to read 3 register values each cycle, while a long multiplication accumulate instruction requires to read 4 registers in a single cycle. In order to support this instruction, Rav adds support for 4 register read. This has been done by splitting a single multiplexer which was used for both Rd and Rs into two multiplexers, as seen in Figure 3.4.

The new implementation of the multiplication unit performs a signed 33-bit multiplication with two registers, and optionally a 64-bit addition with the 64-bit multiplication result, producing a 64-bit result.

Two register are used to store the 64-bit result. The register bank was therefore modified to allow two register writes simultaneously. The **reg_high** and **reg_high_wen** signals handles the writing of this second register.

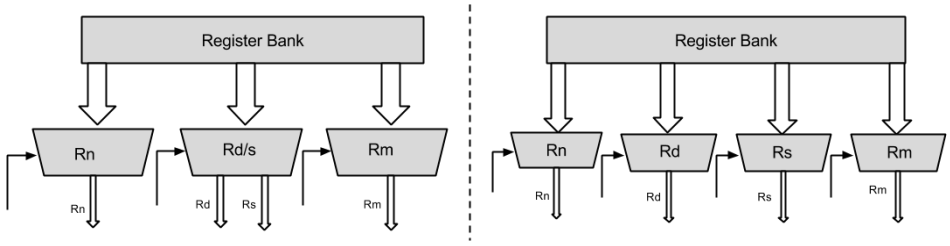

Figure 3.4: Going from three register read support (left) in ARMv3, to four register read support (right) in ARMv4.

The reason for using 33-bit multiplication is due to the fact that the FPGA multiplication unit always performs a signed multiplication. The use of an extra bit in the argument is to force unsigned multiplication behavior. For an unsigned long multiplication, `umull`, or unsigned long multiply accumulate, `umlal`, the register arguments are padded with an extra 0 as the most significant bit. For signed multiplication, `smull` and `smlal`, both register arguments are sign extended by replicating the most significant bit, to produce two 33-bit arguments.

The new multiplication unit is also able to handle the old word sized `mul` and `mla` instructions. This is done by discarding the top bits of the result and only write the 32 low bits. If **long_multiply** signal is high Rn and Rd are concatenated to form the 64-bit argument, else Rd is sign extended into a 64-bit argument. This argument is only added to the result when the **mult_accumulate** signal is high.

3.4.3 Halfword Load and Store

ARMv4 adds new load and store instructions. These are load and store halfword, load signed halfword and load signed byte, `ldrh`, `strh`, `ldrsh` and `ldrsb` respectively.

The Amber processor tile memory interfaces writes a word, 4 bytes, at a time. It uses a 4-bit signal, **byte_enable**, to select which of these 4 bytes that should be written. The **byte_enable** signal is, in the case of halfword memory access instructions, decoded from the memory address. This is done by looking at the second least significant bit. If it is **1**, the two upper bytes in the word should be written, if it is 0 the two lower bytes in the word should be written.

In order to load a value into a register, the Amber processor tile memory interface loads a word from memory. The value then passes through the load function, which processes the value based on three input values. The **byte** signal tells the load function if a single byte is to be loaded. The desired byte is selected based on the two least significant bits in the memory address. The **halfword** signal tells the load function if a halfword is to be loaded. The desired bytes are selected by looking at the second least significant bit in the memory address. The **sign** signal tells

the load function if it should sign extend the value. This value cannot be **1** unless either **halfword** or **byte** signal is **1**. If neither the **byte** nor the **halfword** signal is high, the memory transaction is a regular load and the entire word is written to a register.

3.4.4 Branch and Exchange

In order to be ARMv4T compliant, the `bx` instruction needs to be supported. It is a branch instruction which also allows the user to change the execution state from regular ARM state to Thumb state.

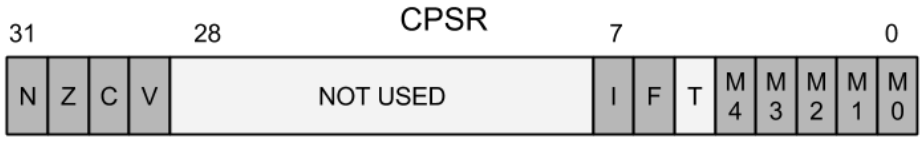

Figure 3.5: The program status register format in ARMv4T.

Since the value in the PC register is word aligned, the least significant bit of any value being written is ignored. The T-bit is added to the PSRs as shown in Figure 3.5 This bit is used by `bx` for setting Thumb state, of CPSR. If this bit is **1** the processor enters the 16-bit thumb execution state. Otherwise the processor stays in 32-bit ARM execution state. As the Thumb execution state is not supported by Rav, writing **1** to the T-bit of CPSR triggers a special thumb exception. The decode stage will then upon the next instruction trigger an undefined exception and put the processor back in 32-bit ARM execution state. This exception can then be checked for by looking at the T-bit in SPSR in the undefined exception handler.

3.5 Testing Rav

When making changes to a system as complex as the Amber processor tile, it is important to verify that it still behaves as expected once the changes are in place. Since the goal of Rav was to extend the functionality of Amber, it was possible to reuse most instruction tests contained in the Amber project. The test framework contained in the Amber project was simple to understand and to extend. It also provided a basic framework for how the design should be tested. This simplified the process of verifying the design. New tests were implemented for all new features added by Rav.

3.5.1 Instruction Tests

The Amber project contains 59 instruction tests, each of which verifies that the Amber processor tile behaves correctly for a certain ARMv2a instruction. These

instruction tests are small assembly programs which ensure that the results produced by the processor are correct, an example can be seen in Figure 3.6

Since a large portion of the Rav core is identical to the Amber processor tile, most of these tests could be reused without modifications.

```
1  main:
2          mov     r1, #3
3          mov     r2, #1
4          add     r3, r1, r2
5          cmd     r3, #4
6          movne   r10, __LINE__
7          bne     testfail
8
9          b       testpass
10
11 testfail:
12         ldr     r11, AdrTestStatus
13         str     r10, [r11]
14         b       testfail
15
16 testpass:
17         ldr     r11, AdrTestStatus
18         mov     r10, #17
19         str     r10, [r11]
20         b       testpass
```

Figure 3.6: Excerpt from an instruction test for the `add` instruction.

Reusing the Amber Instruction Tests

All of the tests which interacted with the status flag bits, the execution mode or performed context switches needed to be modified. Since one of the changes made to the Amber processor tile was how the status bits were stored and modified, tests that performed this explicit change of state needed to be modified to use the newly added `msr` and `mrs` instructions.

Adding New Instruction Tests

In order to verify that the new functionality of Rav was correctly implemented, a set of instruction tests was added to the Amber test framework. What these tests verified is described below.

bx : It is possible to perform a function call return using the `bx` instruction.

ldrh : The `ldrh` instruction correctly loads a 16 bit value to the correct memory location. It is possible to perform a single call to `str` (store word) and then fetch the entire word produced by two `ldrh` calls.

ldrsh : When loading a signed half word from memory using `ldrsh`, the value is correctly sign extended and stored in the target register.

ldrsb : When loading a signed byte from memory using `ldrsb`, the value is correctly sign extended and stored in the target register.

mla_long : The `umlal` instruction correctly multiplies the two source registers, adds the sum of the two target registers and saves the result correctly. When the result of the multiplication is zero, but the value accumulated from the target registers is not zero, the zero flag is not set. When the result of the multiplication is zero, and the value accumulated from the target registers is zero, the zero flag is set.

mul_long : The `umull` instruction correctly multiplies the two source registers and saves the result correctly in the two source registers. When the result is zero, the Z-bit is set in CPSR.

mla_long_hazard : For each of the registers used as argument to the `umlal` instruction, if it is loaded from memory in the instruction preceding the `umlal` instruction, the processor will be stalled, and the correct value will be used.

mul_long_hazard : The same as for *mla_long_hazard* only with the `umull` instruction.

mrs : The `mrs` instruction correctly copies the value of the CPSR to the target register.

msr_flags_only_i : `msr` with the `flag_only` bit set only updates the flag bits of the CPSR register, and that it is possible to use an immediate value as operand.

msr_flags_only_r : The same as *msr_flags_only_i*, but uses a register as operand instead of an immediate value.

msr_immediate : `msr` with an immediate value as argument correctly updates the entire CPSR register.

shift_cpsr_carry : The barrel shift unit is able to use the carry bit from the CPSR register.

smla_long : Doing signed multiplication with `smlal` works correctly. If the result is negative, the N-bit of the CPSR is set, otherwise it is not set. The value from the target registers is correctly added to the multiplication result, for both positive and negative values.

smul_long : Doing signed multiplication with `smull` works correctly. If the result is negative, the N-bit of the CPSR is set, otherwise it is not set.

spsr : It is possible to write to and read from the SPSR register using the `msr` instruction.

spsr_change_mode : The CPSR register is correctly backed up in the SPSR register of the correct execution mode during context switches. When returning to user mode, the CPSR value is correctly restored.

spsr_immediate : `msr` with an immediate value as argument and an SPSR register as target correctly updates the entire SPSR register.

strh : Storing a half word using `strh` works correctly. It is possible to load a value written by two `strh` calls by using `ldr`.

system_mode : It is possible to change the execution mode from system mode. The same registers are available from system mode and user mode.

thumb_exception : Trying to enter Thumb state by setting the T-bit in CPSR will trigger an undefined exception.

In addition to these new tests, several of the tests updated to work with Rav also tested the new functionality. One of the tests which verified a large set of the new functionality was the **ldm5** test. This test verifies that when an `ldm` instruction is called with R15 as one of its target registers and the S-bit set, the correct SPSR value is copied to CPSR.

3.5.2 System Testing

Running large applications is often done to test that the behavior of an entire processor is correct.

Bare Bones C Application

Since Rav will replace Amber as the processor tile in the SHMAC architecture, it is important that any test applications used in the SHMAC project can be executed on Rav. The SHMAC project contains a small C application which shows developers how to correctly initialize the processor tile, and how to work with interrupts. It starts off by setting up the stack for all of the various execution modes. This code needed to be modified for Rav in a similar manner to that of the instruction tests which changed execution mode. The new version can be seen in Figure 3.7.

```
mov r0, =top_stack

// Change to IRQ mode
msr cpsr, #0xD2
nop

// Set stack - 1MB
mov sp, r0
sub r0, r0, #0x100000

// Change to UND mode
msr cpsr, #0xDB
nop

// Set stack - 1MB
mov sp, r0
sub r0, r0, #0x100000
```

Figure 3.7: Setting up the stack pointers of various execution modes for Rav.

In this application, the interrupt controller is configured to trigger a `TIME` interrupt each second, and a `HOST` interrupt when the system receives a packet over UART. Once the stacks of the execution modes are initialized, the interrupt handlers are defined and the interrupt controller is configured, the `main` function of the application can be called. The application print its processor tile ID once, for each `TIME` interrupt it prints a "." and for each `HOST` interrupt it echoes the character received back to the host.

This application verifies that context switches works properly, that interrupts are properly managed in the SHMAC environment and that the status of the execution modes are treated correctly. It helps verify parts of the processor that had been modified during the implementation of Rav.

3.5.3 Performance

The motivation behind upgrading the SHMAC processor tile was increased software compatibility, not performance. However, it was desirable to verify that the performance had not been reduced by the upgrade. This was done using the instruction tests and the SHA1 benchmark.

The Rav design met the same timing constraints as Amber and can therefore use the same clock frequency. Since both processor tiles used the same frequency, the number of *cycles* spent during an instruction test could be used to find the execution time. 50 of the original 59 instruction tests were used. Tests that used multiplication instructions or that used the cache differently were not included.

Results from these instructions test show that the Rav processor tile is on average 1.7% faster, Appendix C contains the results used for this calculation. This small speedup can be attributed to less stalling when the status bits are in their own register, and more efficient instructions for manipulating the status bits registers.

The multiplication tests had an average of 721% speedup. This speedup is attributed the new two cycle multiplication unit.

The SHA1 benchmark [31] was performed on both the Amber and Rav processor tile. This benchmark was run with cache. The original Amber processor tile had a throughput of 1249 kB/s while the Rav processor tile had a throughput of 1368 kB/s.

Chapter 4
Porting Linux to SHMAC

This chapter describes the steps taken when porting the Linux 3.12.13 kernel to SHMAC with the ARMv4T compatible processor tile. It provides helpful information for porting Linux to a new ARM platform, especially for an MMU-less platform.

4.1 Porting Steps

In subsection 2.5.3 it was stated that Linux is a portable operating system kernel, and the criteria it meets in order to be called portable. The portability of the ARM specific code has undergone major changes during the last years in order to deal with the increasing variety of ARM System on Chip (SoC). These changes involve modifications in the code infrastructure to create a common framework for adding support for new ARM SoCs. During the Embedded Linux Conference Europe 2012, Thomas Petazzoni held a presentation where he described the changes and the current best practices for adding ARM SoCs support [6]. This presentation "The ARM SoC Checklist" is the basis for the steps taken when porting Linux to SHMAC.

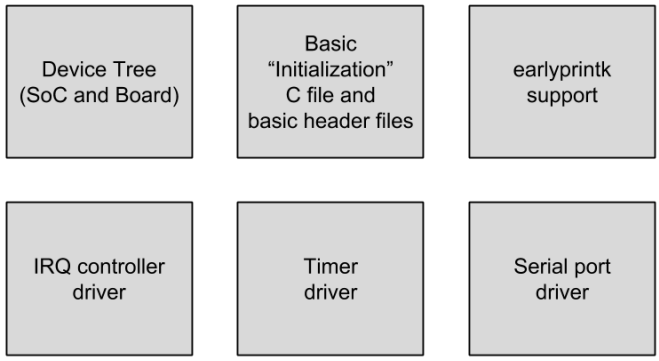

Figure 4.1: ARM SoC porting steps. (From [6].)

Figure 4.1 shows an illustration of the porting steps. These steps were all performed

when porting the Linux kernel to SHMAC. It should be noted that no kernel code was modified during this project.

4.2 Linux Device Tree

When creating a Linux kernel image, it is desirable that the image can be executed on several platforms, which is why the LDT is utilized. As described in 2.5.3, the LDT functionality moves hardware specification and driver selection from the kernel to a separate LDT which is independent from the Linux kernel.

The LDT is written in Device Tree Syntax, specifically developed for this purpose. The Device Tree Compiler is used to compile the LDT definitions to a binary format. All LDT bindings are documented in the Linux kernel documentation.

Since the Linux kernel did not include any hardware description of the SHMAC architecture, this needed to be added. The entire SHMAC LDT description can be seen in Figure 4.2.

This description starts by defining a top level compatible string. This is used by the kernel to identify the correct configuration. The bootargs contained in the `chosen` section is sent to the Linux kernel as bootloader arguments. This way, configurations that should be provided by the bootloader can be reconfigured as seen fit by the developers without having to actually modify the bootloader.

The hardware specific code starts in the `soc` section. Here, each of the components which requires their own driver is described. Which details are provided in the devices description depends on what type of device is being described.

Common for all SHMAC device descriptions are the compatible flags and register definition. The compatible flag tells the kernel which driver should be loaded for this device. The register definition specifies where in memory the control for the device can be found.

The `interrupt-controller` description contains four extra fields:

interrupt-controller A flag stating that this device description is an interrupt controller.

#interrupt-cells=1 The width of the interrupt line is 1 word (32 bits)

valid-mask Bitmask telling which of the 32 interrupt lines that are valid.

clear-mask Bitmask telling which of the 32 interrupt lines should be cleared upon initialization.

Both the `timer` and `uart` descriptions contain `interrupts` values that indicate which interrupt lines belongs to the device. This value is used by the interrupt

controller when handing over control to the correct interrupt handler upon receiving an interrupt. The `uart` description also contains a `status` flag, telling the kernel that the device being described is not disabled. The `timer` description also contains a `frequency` flag telling the Linux kernel the frequency of the clock source.

```
/include/ "skeleton.dtsi"
/{
        compatible = "shmac,shmac";
        model = "SHMAC RAV";
        interrupt-parent = <&intc>;

        aliases {
                serial0 = &uart0;
        };

        chosen {
                bootargs = "earlyprintk console=ttyshmc";

        };

        soc {
                compatible = "simple-bus";
                #address-cells = <1>;
                #size-cells = <1>;
                ranges;

                intc: interrupt-controller@fffe2000 {
                        compatible = "shmac,shmac-intc";
                        reg = <0xfffe2000 0x20>;
                        interrupt-controller;
                        #interrupt-cells = <1>;
                        clear-mask = <0xffffffff>;
                        valid-mask = <0x0000000f>;
                };

                uart0: uart@ffff0000 {
                        compatible = "shmac,shmac-uart";
                        reg = <0xffff0000 0x24>;
                        interrupts = <1>;
                        status = "ok";
                };

                timer0: timer@fffe1000 {
                        compatible = "shmac,shmac-timer";
                        reg = <0xfffe1000 0x20>;
                        clock-frequency = <60000000>;
                        interrupts = <2>;
                };

                timer1: timer@fffe1100 {
                        compatible = "shmac,shmac-timer";
                        reg = <0xfffe1100 0x20>;
                        clock-frequency = <60000000>;
                        interrupts = <3>;
                };
        };
};
```

Figure 4.2: SHMAC LDT definition.

4.3 Basic Setup

The first steps in porting Linux for SHMAC is creating the machine specific folder for SHMAC. This contains basic SoC initialization and SoC constants. The next step

is to provide Linux with simple macros for putting a character through the UART. This enables the kernel to print messages in the early phases of the initialization without a serial driver initialized.

4.3.1 SoC Initialization

A minimal SoC initialization is a file that contains a LDT machine definition for either a SoC or a family of SoCs. This definition describes the initialization function to be called and the LDT compatibility list. The initialization function calls the `of_platform_populate` function which walks the LDT and populate the platform devices from the nodes, as seen in Figure 4.3

```
1  static void __init shmac_init(void){
2    of_platform_populate(NULL, of_default_bus_match_table, NULL, NULL);
3  }
4
5  static char const *shmac_dt_compat[] = {
6    "shmac,shmac",
7    NULL
8  };
9
10 DT_MACHINE_START(SHMAC_DT, "SHMAC")
11   .init_machine = shmac_init,
12   .dt_compat = shmac_dt_compat,
13 MACHINE_END
```

Figure 4.3: Linux SoC kernel configuration.

4.3.2 Early Kernel Messages

The kernel uses `printk` to print messages to the user. The `printk` function logs the messages in a buffer and writes the buffer to every console which has been registered with the function. In the early stages of the kernel initialization, `printk` has no consoles registered so the messages are logged in a buffer. In order to print these messages before the consoles have been registered, Linux provides the earlyprintk mechanism where a SoC provides four simple low level functions for pushing a single character through the UART. These low level functions are implemented as assembly macros. Their implementation can be seen in Figure 4.4.

```
#define SYS_BASE 0xffff0000
#define SYS_OUT_DATA 0x00
#define SYS_INT_STATUS 0x20
#define INT_HOST_IRQ 0x2

/* Loads the base memory address of the system registers. */
.macro addruart,rp,tmp
  ldr \rp, =SYS_BASE
.endm

/* Send a character to the host.*/
.macro senduart,rd,rx
  strb \rd, [\rx, #SYS_OUT_DATA]
.endm

/* Wait until the host is ready to recieve a new character. */
.macro waituart,rd,rx
1001: ldr \rd, [\rx, #SYS_INT_STATUS]
  tst \rd, #INT_HOST_IRQ
  bne 1001b
.endm

/* No implementation needed for SHMAC. */
.macro busyuart,rd,rx
.endm
```

Figure 4.4: SHMAC early printk implementation.

4.3.3 Starting the Kernel

With early kernel messages supported, it is possible to debug the initialization process. To start the kernel initialization process, a bootloader is needed.

The job of a bootloader is to initialize the hardware to a state known by the kernel, pass system information, and tranfer control to the kernel. The LDT is capable of providing the kernel with the same information as the bootloader. For this project, all system information is provided to the kernel by the LDT.

The bootloader created for this project provides address of the LDT to the kernel before handing over control. It is implemented in assembly, and the memory location of the kernel is hard coded, so the Linux kernel needs to be placed in this memory location for it to be booted.

A simplified version of the bootloader is shown in Figure 4.5.

```
/* Set all registers not used to '0' */
  mov r0, #0
  mov r3, #0
  mov r4, #0
  mov r5, #0
  mov r6, #0
/* Load the Linux Device Tree memory address */
  ldr r2, =(dtb_entry)
/* Now hand control over to the Linux kernel */
  ldr r7, =(linux_entry)
  mov   pc, r7
```

Figure 4.5: Simplified version of the bootloader.

4.4 Interrupt Controller Driver

The interrupt controller used on the SHMAC processor tiles are from the Amber project. In order for Linux to be able to use the interrupts generated by this interrupt controller, a driver needs to be defined. The Amber Interrupt Controller (AIC) is relatively simple, and it does not support prioritizing or pending interrupts.

A Linux interrupt controller driver needs to implement the following functions:

ack Clear an interrupt line.

mask Disable an interrupt line.

unmask Enable an interrupt line.

init Initialize the interrupt controller, and the driver.

handle Deal with an interrupt.

map Translates a virtual interrupt request number used by the kernel to the physical interrupt request number used by the interrupt controller.

The initialization code for the AIC driver loads the base address of the interrupt controller from the device tree, configures the IRQ-domain (described below), and sets the handler function. The `ack`/`mask`/`unmask` functions writes **0** or **1** to their respective control bits in the AIC. The `handle` function identifies which interrupt line generated the interrupt and hands the virtual interrupt request number over to the Linux kernel so that the correct device specific handler can be called.

An IRQ controller needs to associate each interrupt device with an interrupt number. The Linux kernel assigns a virtual interrupt identifier to each device which registers an interrupt. This is done to avoid that several devices require the same interrupt

identifier, since it is important that the identifiers are unique. These virtual interrupt identifiers are translated to hardware interrupt numbers in the interrupt controller driver.

4.5 Timer Driver

The Linux kernel uses two devices for keeping track of time: A *clocksource* device and a *clockevent* device.

The SHMAC timer driver exports its compatibility flag, as seen in Figure 4.6. This is done so that the LDT can identify this driver as the correct one for the SHMAC architecture. The SHMAC timer driver initializes both a clocksource and a clockevent device.

```
1 CLOCKSOURCE_OF_DECLARE(shmac, "shmac,shmac-timer",
2       shmac_timer_init);
```

Figure 4.6: SHMAC timer driver exporting compatible flags to the LDT.

4.5.1 Clockevent Device

The standard Linux clockevent device interface contains the functions required for the kernel to be able to generate periodic ticks. The SHMAC clockevent device is implemented using a periodic timer found in the SHMAC processor tile. Its functions can be seen below.

start start the timer.

stop stop the timer.

clear clear the timer interval value.

set_value set the timer interval value.

set_mode set the timer mode.

set_next_event set up the timer to generate the next tick event.

time_interrupt manage the timer interrupts, call the registered handler.

timer_init initialize the timer.

4.5.2 Clocksource Device

In addition to the events generated by the clockevent device, Linux requires a free running timer to keep track of wall time. In the case of SHMAC, this is implemented using a SHMAC system register that keeps track of the amount of clock ticks elapsed since the system was started.

The only information required by the kernel is the frequency of these clock ticks, and how this value can be read. Figure 4.7 shows how the system register is read, compared to the eight functions implemented for the clockevent device, it is evident that the clocksource device interface is simpler.

```
1 static u32 notrace shmac_sched_clock_read(void)
2 {
3         return readl(SHMAC_SYSTEM_TICK_COUNTER);
4 }
```

Figure 4.7: Function for reading the value of the SHMAC system clock register.

4.6 Serial Driver

SHMAC is able to perform serial communication with the host system. This communication is done by reading and writing to a set of three registers that are visible to both the host and the SHMAC system. By implementing a serial interface on top of this protocol, it is possible to make Linux think it has a standard serial interface available for use.

For a serial port to be integrated in a Linux system, it needs to be visible for userland applications as a TTY device.

To provide a serial interface, it was necessary to implement a complete driver for serial communication in the TTY layer. This driver is a platform driver with an LDT binding, and it is integrated with both the UART and the console subsystems in the Linux kernel.

4.6.1 Driver Registration

The driver initialization function is responsible for registering the driver information with the kernel. To do this, the initialization code calls the `uart_register_driver` and `platform_register_driver` functions.

The `uart_register_driver` function registers the `uart_driver` and the `console` structures with the serial core layer. The `uart_driver` structure has information about driver name, minor and major numbers and the number of serial ports supported by this driver. The `platform_register_driver` function registers a

`platform_driver` with two functions, `probe` and `remove`. The `probe` function is called once for each node in the LDT that is compatible with the driver.

4.6.2 Console

If the `console` structure is registered with the flag `CON_PRINTBUFFER` set, the console is added to `printk`s list of consoles. The callback function `shmac_uart_console_write` will then be called for each message from the kernel. This uses the `uart_write_console` helper function, which ensures that each newline character is handled correctly. This function, in turn, takes a function for printing a single character as input. The `shmac_console_putchar` implements the SHMAC protocol for printing characters via the SHMAC system registers, see Figure 4.8.

```
1  /* print a single character through the SHMAC system register */
2  static void shmac_console_putchar(struct uart_port *port, int ch)
3  {
4          while(readl_relaxed(port->membase + UARTn_STATUS) &
               UARTn_STATUS_TXBUSY);
5          writel_relaxed(ch, port->membase + UARTn_TXDATA);
6  }
7
8  /* print a console message */
9  static void shmac_uart_console_write(struct console *co, const char *s
       , unsigned int count)
10 {
11   struct shmac_uart_port *shmac_port = shmac_uart_ports[co->index];
12         uart_console_write(&shmac_port->port, s, count,
               shmac_console_putchar);
13 }
```

Figure 4.8: Console write function in SHMAC serial driver.

4.6.3 Serial Port

The `probe` function in the `platform_driver` structure is called each time a platform device is added to the system. This function will parse the device node in the LDT and extract information about the memory map and interrupt lines. Memory is then allocated for a `uart_port` structure which is registered with the serial core by calling the function `uart_add_one_port`. The `uart_port` structure has information about the memory map of the port, the interrupt line it should use and an `uart_ops` structure containing callback functions to the serial driver.

4.6.4 UART Callback Functions

The `uart_ops` structure has callback functions which implements the device specific functions and is the interface between the serial core and hardware specific driver. Whenever a SHMAC UART callback function has an empty definition, the return

value is set according to what the Linux documentation states should be the default value.

tx_empty Test whether the transmitter queue for the port is empty. Since the SHMAC serial protocol has no transmitter queue this function always returns "empty".

set_mctrl Set the modem control lines for the port. In the SHMAC serial implementation, this function is empty.

get_mctrl Return the status of the modem control inputs. In the SHMAC serial implementation, this function always returns "active".

start_tx Start the transmission of characters. In the SHMAC serial implementation, this function calls the `shmac_tx_chars` function which handles the transmission.

stop_tx Stop the transmission of characters. In the SHMAC serial implementation, this function is empty since the transmission is handled without interrupts

stop_rx Stop receiving character. In the SHMAC serial implementation, this function is empty.

enable_ms Enable the modem status interrupts. In the SHMAC serial implementation, this function is empty.

break_ctl Control the transmission of a break signal. In the SHMAC serial implementation, this function is empty.

startup Initialize interrupts and setup all low level driver state. In the SHMAC serial implementation, this function sets up the receiver interrupt handler.

shutdown Disable the port and free resources. In the SHMAC serial implementation, this function removes the interrupt handler used for receiving character.

set_termios Alter the UART parameters of the port. In the SHMAC serial implementation, this function is empty since there are no parameters to change.

type Return the UART port type.

request_port Request the resources required by the port. In the SHMAC serial implementation, this function requests the memory map of the SHMAC system registers used to communicate with the host.

release_port Release the resources acquired by the port. In the SHMAC serial implementation, this function release the memory map of the SHMAC system registers.

config_port Perform autoconfiguration of the port. In the SHMAC serial implementation, this function sets the port type to "SHMAC port".

verify_port Verify the port configuration. In the SHMAC serial implementation, this function returns "false" if the port is not a SHMAC port.

4.6.5 UART Transmission

When the `write` system call is invoked on a file descriptor that is connected to a serial character device, the `write` call is invoked on the TTY subsystem. The TTY subsystem will perform its own buffering and line editing before calling the device driver. When the TTY subsystem calls the `start_tx` function, the characters to be written are stored in a circular buffer named `xmit` and the callback function `uart_tx_start` is called. It is then up to the serial driver to transmit these characters over the serial interface. The function `shmac_tx_char` shown in Figure 4.9 handles the transmission of these characters. First the function checks if it should send the high priority character, this is used to implement flow control. Next it loops over the circular buffer and pushes all the character over the serial interface. The driver is expected to call the `uart_write_wakeup` function when the number of characters in the transmit buffer drops below a certain threshold. This will wake up any processes which is waiting for the TTY subsystem to flush its buffer.

```
static void shmac_tx_chars(struct uart_port *port)
{
    struct circ_buf *xmit = &port->state->xmit;

        if (port->x_char) {
                /* send port->x_char out the port here */
                shmac_console_putchar(port, port->x_char);
                port->icount.tx++;
                port->x_char = 0;
                return;
        }

        while (1)
                if(!uart_circ_empty(xmit) && !uart_tx_stopped(port)) {
                        port->icount.tx++;
                        shmac_console_putchar(port, xmit->buf[xmit->
                            tail]);
                        xmit->tail = (xmit->tail + 1) & (
                            UART_XMIT_SIZE - 1);
                } else
                        break;

        if (uart_circ_chars_pending(xmit) < WAKEUP_CHARS)
                uart_write_wakeup(port);

        if (uart_circ_empty(xmit))
                shmac_uart_stop_tx(port);
}

static void shmac_uart_start_tx(struct uart_port *port)
{
        /* Start transmitting characters */
        shmac_tx_chars(port);
}
```

Figure 4.9: SHMAC serial driver UART transmit function.

4.6.6 UART Reception

Receiving characters from the user is handled using interrupts. When the `uart_startup` function is called, it registers an interrupt handler for the specified IRQ number. The IRQ handler will then read the character from the UART and insert it into the TTY layer by calling `tty_insert_flip_char`, as seen in Figure 4.10. The SHMAC serial driver will then notify the TTY layer that characters (in this case, never more than one) have been inserted by calling `tty_flip_buffer_push`.

```
static irqreturn_t shmac_uart_rxirq(int irq, void *data){
        struct shmac_uart_port *shmac_port = data;
        struct uart_port *port = &shmac_port->port;
        struct tty_port *tport = &port->state->port;
        char rxchar;
        spin_lock(&port->lock);

        rxchar = readl_relaxed(port->membase + UARTn_RXDATA);
        port->icount.rx++;
        tty_insert_flip_char(tport, rxchar, 0);

        spin_unlock(&port->lock);
        tty_flip_buffer_push(tport);
        return IRQ_HANDLED;
}
```

Figure 4.10: SHMAC serial driver UART receive function.

4.7 Adding Multicore Support to Linux

The problem description for this thesis states that multicore support should be added to Linux if times permits. This section presents the results from a detailed investigation into what modifications would be required for the ported version of Linux to support SMP.

ARMv6 is the first ARM architecture with SMP support in the Linux kernel. This can be seen when inspecting the Linux configuration dependencies and by looking at the kernel code, as shown in Figure 4.11. The only reason for this requirement is that the atomic instructions required for SMP execution are implemented using instructions not found in earlier ARM ISA versions than ARMv6.

```
#if __LINUX_ARM_ARCH__ < 6
/* min ARCH < ARMv6 */
#ifdef CONFIG_SMP
#error "SMP is not supported on this platform"
#endif
```

Figure 4.11: Excerpt from Linux SMP code for ARM.

A second requirement for SMP support in Linux is a Memory Protection Unit (MPU) or MMU, see Figure 4.12. One reason for this requirement is that either an MMU or an MPU is need for the kernel to be able to enforce memory protection. In a system without an MPU or MMU, such as SHMAC, there is no way to protect memory addresses from being modified by unauthorized applications. Any application can

modify any memory address, including those containing the kernel code. It is possible to have SMP without MPU or MMU, but this would imply security issues. Also, an MMU or MPU has the ability to mark memory regions as "shared". This is required by the load exclusive (`ldrex`) and store exclusive (`strex`) mechanism added in ARMv6.

```
config SMP
  bool "Symmetric Multi-Processing"
  depends on CPU_V6K || CPU_V7
  depends on GENERIC_CLOCKEVENTS
  depends on HAVE_SMP
  depends on  MMU || ARM_MPU
```

Figure 4.12: Excerpt from Linux kernel configuration for ARM.

These requirements imply that either the kernel code or the hardware would need to be modified in order to add SMP support for Linux on SHMAC. A possibly incomplete set of changes required for adding SMP support is now presented.

4.7.1 Adding Kernel Support

For Linux to be able to provide mutual exclusion, it depends on a set of locking primitives. Since the existing locking function for ARM are implemented using instructions not supported by ARMv4T the functions would have to be re-implemented using the `swp` atomic instruction.

Atomic

In order for the ported Linux kernel to support SMP, the atomic interface would need to be updated.

The functions which need to be defined in the atomic interface are shown in Table 4.1:

Compare and Exchange

A second synchronization mechanism required by the Linux kernel is the `cmpxchg` (compare and exchange) function. This function performs a conditional atomic exchange between a register and a memory address. The condition is that the value residing in that memory address must match that of a given register. This function could be used as the base function for implementing a mutex, which again could be used to simplify the implementation of the functions in the atomic interface.

Linux atomic interface

Function	Functionality
`atomic_add_return`	Add a value to the atomic value and return the new value.
`atomic_sub_return`	Subtract a value from the atomic value and return the new value.
`atomic_cmpxchg_return`	Compare two values, and change the value of the atomic only if they are equal.
`atomic_clear_mask`	Clear bits in the atomic.

Table 4.1: Linux atomic interface.

Spinlock

The spinlock interface provides functions for interacting with a spinlock. Compared to the `cmpxchg` and `atomic` interfaces, these functions includes mechanisms not only for handling the lock, but also for the power management to be done while waiting for the lock. Since the Rav processor does not support any power management, these functions could be implemented using `cmpxchg`.

Configuration

The configuration process of the Linux kernel contains a large set of dependencies between the various properties. For instance, if the value of the `CONFIG_ARCH_` variable indicates an ARM architecture that precedes ARMv6, it would not be possible select `CONFIG_SMP`. In order for the ported Linux kernel to support SMP, several of these dependencies would need to be removed, including dependencies between SMP, the ISA version and whether the processor contains an MPU or MMU.

4.7.2 Adding Hardware Support

Another possible solution is to update Rav to support the ARMv6 ISA. This would require supporting the load exclusive and store exclusive instructions. These instructions might require either MMU or MPU support in order to implement the exclusive protocol.

4.8 Testing Linux

Testing of the ported Linux kernel has been done in two areas: the SHMAC device drivers and the Linux system call interface.

4.8.1 Interrupts

The interrupt controller driver is arguably one of the most central pieces of the ported Linux kernel, as it works as a wrapper for all interrupt handling. The functionality that needed to be tested was that the virtual IRQ-number mapping was handled correctly, that the appropriate interrupt handler was called, that it was possible to mask and unmask interrupts, and that the configuration from the LDT was correctly invoked.

This testing was performed by using the already implemented timer driver and serial driver. A test application was executed that echoed the users input, and that generated a timer interrupt each second. By adding print statements in various places of the interrupt controller driver, and executing this test application, it was possible to verify that both timer and serial devices were being handled correctly.

4.8.2 Timer

The timer driver provides timed interrupts and kernel time progression. It had to be verified that the timer was able to generate the interrupts, and that kernel time progressed correctly. To verify that the timer driver correctly provided timed interrupts, the system call `sleep` and `setitimer` were used. To verify that the kernel time progression was correct, long running benchmarks were used with the time utility provided by BusyBox. The time reported was compared with the wall clock.

4.8.3 Serial Communication

The serial driver was tested by doing user interactions with the BusyBox shell. The driver was tested by typing commands to the shell, performing line editing by deleting characters with the backspace and delete key and navigating in the line with the arrow keys. Also, the ability to deliver control codes such as `Ctrl+c` or `Ctrl+z` was tested.

4.8.4 System Calls

The Linux kernel provide its services through the system call interface. It is therefore important to verify that the system calls behave as expected. There exists 380 Linux system calls, which are too many to verify individually given the time frame of this project. A set of 38 system calls, considered to be of significant importance by [29], was verified.

By modifying the kernel code to print the name of each system call being invoked, it was possible to do a crude system call tracing. BusyBox contains a large set of applications that interact heavily with the kernel. These applications were built for the ported Linux kernel, and used to trigger the system calls. By calling BusyBox applications with system call tracing, it was possible to trace which system calls were being invoked by the applications. By verifying the behavior of the application

performing the system calls, it was possible to indirectly test whether or not the various system calls invoked were behaving correctly. Figure 4.13 shows the system call trace from calling the BusyBox application `ls`. In cases where it was difficult to verify a system call using a BusyBox application, a test application was written.

```
write, open, lseek,
write, close,ioctl,
ioctl, sigaction, setpgid,
execve, ioctl, ioctl,
gettimeofday, ioctl, ioctl,
ioctl, stat64, open,
fstat64, fcntl, mmap_pgoff,
getdents, lstat64, lstat64,
getdents, close
```

Figure 4.13: System call trace for `ls`.

Process System Calls

The Linux process management subsystem handles how processes are created and destroyed. Linux uses signals as a mechanism for handling interprocess communication. When a signal is delivered to the process, the process' signal handler is run. All process related system calls were verified using test applications.

Simple Shell Sits in a loop and reads commands from the user. For every command, it uses the `vfork` call to create a new process. The new process then tests if the command is a file, if the file exists it replaces the process image using the `execve` system call with this file. If the file does not exist the process exits using the `_exit` system call. The old process then wait for the new process to exit before it reads a new command.

Test Alarm Installs an alarm signal function handler and then performs the `alarm` system call in order to generate an alarm signal after one second. When the alarm signal handler is run, it decrements a counter. If the counter is not zero, it performs a new alarm system call in the same way as before.

Test Sigpending Blocks signal by using `sigprocmask`, the process is then uninterruptable for 10 seconds. After 10 seconds, the program checks if the user has tried to interrupt the program using `sigpending`.

Test Sigsuspend Creates two processes, the first process spawns the second process and then suspends until it receives a signal using `sigsuspend`. The second process will then deliver a signal using `kill` to the first process and wake it up.

Test Clone The clone test creates two threads using `clone` and hands them two different arguments. The test then waits for both of the threads to terminate before it exits.

A summary of the process and signal related system calls that were tested, and which test is used to verify them, is shown in Table 4.2.

Process System Calls

System Call	Functionality	Verification
`vfork`	create a child process and block parent	simple_shell
`wait`	await process completion	simple_shell
`execve`	execute program	simple_shell
`_exit`	terminate the process	simple_shell
`sigaction`	examine and change a signal action	test_timer
`sigprocmask`	examine and change blocked signals	test_sigpending
`sigpending`	examine pending signals	test_sigpending
`sigsuspend`	wait for a signal	test_sigsuspend
`kill`	send a signal to a process	test_sigsuspend
`alarm`	set an alarm clock for delivery of signal	test_signals
`clone`	start a new thread	test_clone

Table 4.2: Verifying process system calls.

Security System Calls

Linux handles security with the notion that a process runs with the same permissions as the user who started the process. This is implemented through the use of a user ID and a group ID. It is possible for a privileged process to drop to a lower security level by assuming the identity of another user. This is implemented through the use of effective user ID and effective group ID.

Changing the owner and access permission bits of files in the system has been tested using the BusyBox utilities `chmod`, `chown`.

Test IDs Uses the get/set user/group ID system calls and tries to open a test file owned by the root user. It starts out as the root user and creates the test file. First it changes the effective user and tries to open the file, this access fails. It then switches the effective user back to the root user and tries again to open the file, this access succeeds. It then switches the real user ID and tries to open the file, this access fails. Then it tries to switch back to the root user and open the file, this switch and the access to the file fails.

Test permissions Uses `access` to check file permission on a file. It then switches to another user and performs the same test.

A summary of the security related system calls that were tested and the test that verifies them is shown in Table 4.3

Security		
System Call	**Functionality**	**Verification**
`chmod`	change file mode bits	`chmod`
`chown`	change file owner and group	`chown`
`access`	determine accessibility of a file relative to directory file descriptor	test_permissions
`getuid`	get a real user ID	test_ids
`geteuid`	get the effective user ID	test_ids
`getgid`	get the real group ID	test_ids
`getegid`	get the effective group ID	test_ids
`setuid`	set user ID	test_ids
`setgid`	set group ID	test_ids
`seteuid`	set effective user ID	test_ids
`setegid`	set effective group ID	test_ids

Table 4.3: Verifying security system calls.

Memory Management System Calls

Memory in a Linux system is typically managed by the library procedure `malloc`. This procedure relies on the system call `brk` and `sbrk`, which both change the size of the data segment for the process. The `mmap` and `munmap` system calls are used to map and unmap a file in memory. As the Linux kernel on SHMAC runs entirely in memory, these system calls are used every time a file is created or destroyed. They were verified by the BusyBox utilities `touch` and `rm` which were used to create and destroy a file.

Sbrk test A simple version of `malloc` was written to test `sbrk`. It implements `malloc` by expanding the data segment with the amount of bytes that is requested and returning the address of the original end of the data segment. If the `sbrk` call fails it returns zero.

A summary of the memory management system calls that were tested and the test that verifies them is shown in Table 4.4

File and Folder System Calls

Files and folders play an important role in Linux, and there are many system calls for manipulating them. BusyBox contains many utilities used for manipulating files and folders from the shell.

Memory System Calls

System Call	Functionality	Verification
`sbrk`	change data segment size	test_sbrk
`mmap`	map pages of memory	`touch`
`munmap`	unmap pages of memory	`rm`

Table 4.4: Verifying memory system calls.

A summary of the file and folder system calls that were tested and the test that verifies them is shown in Table 4.5.

File and Folder System Calls

System Call	Functionality	Verification
`open`	open file relative to directory file descriptor	`ls`
`close`	close a file descriptor	`ls`
`read`	read from file	`cat`
`write`	write to file	`echo`
`lseek`	move the read/write file offset	`tail`
`stat`	display file or file system status	`find`
`pipe`	create an interprocess channel	`|`, as in "ls | sort"
`fcntl`	file control	`find`
`mkdir`	make directories	`mkdir`
`rmdir`	remove empty directories	`rmdir`
`link`	create a link to a file	`ln`
`unlink`	remove the file	`mv`
`chdir`	change working directory	`cd`

Table 4.5: Verifying files and folders system calls.

Chapter 5
Userland Toolchain

Two different toolchains were needed for this project: one to build the Linux kernel and one to build userland applications. The userland toolchain needed to contain a standard C library implementation compatible with the Linux system call interface. The kernel toolchain needed to be capable of producing ARMv4T code.

The toolchain used to compile the Linux kernel was a prebuilt bare metal ARM toolchain [9]. The userland toolchain had to be built from source, a process that proved to be more difficult than expected. This chapter describes the steps taken in order to produce a working userland toolchain.

5.1 Userland Toolchain Requirements

The job of the userland toolchain is to produce applications that can be executed by the kernel. They should also be able to communicate with the kernel through the Linux system call interface.

As the target for this toolchain was a Linux kernel on an MMU-less architecture, the *uClibc* implementation of the standard C library had to be used. At the time of writing, this is the only standard C library implementation compatible with an MMU-less version of Linux. Also, in order for the target Linux kernel to be able to execute the executable files, they would need to be converted from the ELF format to the BFLT format. For this operation a tool called *elf2flt* was required. In addition to these specialized MMU-less tools, the GNU C compiler, GNU linker and GNU assembler needed to be contained in the toolchain. All of these components needed to be configured and compiled in such a way that they were able to work with each other and produce a correct executable file for the target system.

Crosstool-NG was used to build the userland toolchain.

5.2 Challenges Encountered

Several challenges had to be overcome during the toolchain building process. A majority of these challenges emerged from the lack of an MMU. Below is a short

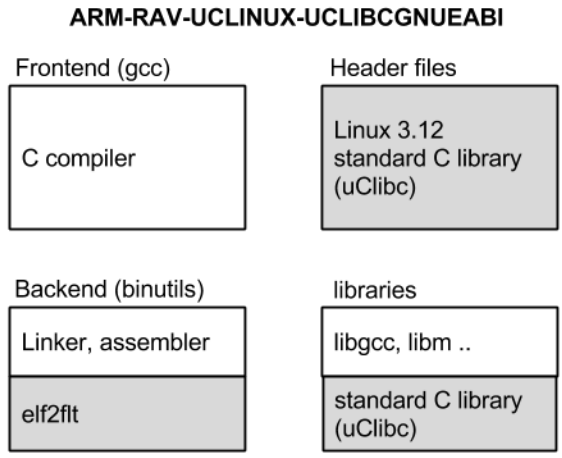

Figure 5.1: Components included in the userland toolchain.

description of the challenges encountered.

5.2.1 libgcc

The libgcc library and ARMv4T instruction set had some compatibility issues with the version of GCC used. With GCC 4.8.1, the linker was unable to link with libgcc. With GCC 4.7.2, the compilation of libgcc terminated with a segmentation fault. Therefore, version 4.6.4 of GCC was used in the final toolchain.

5.2.2 uClibc

It should be possible to utilize shared libraries in uClibc. However, it was not possible to compile the toolchain with this setting, as the linking with libgcc failed. Therefore, uClibc was built using only static libraries.

5.2.3 Crosstool-NG

Crosstool-NG assumed that all ARM-architectures had an MMU and terminated with a message saying that ARM was an invalid target for an MMU-less toolchain. Therefore, the Crosstool-NG source code needed to be modified to allow ARM-architectures without an MMU as a target.

5.2.4 elf2flt

The development of elf2flt appeared to be in a state of abandonment, with their official repository being offline since February 2014. The last version obtained from the official repository had no support for the ARM.extab sections. The ARM.extab

sections are a part of the exception handling functionality of the ARM ABI, and are used to provide stack trace information. The elf2flt version included in the PTXdist project came from an unofficial repository and contained fixes not found in the original version. This version included patches which added support for the ARM.extab sections.

When compiling for ARMv4T architectures, the compiler inserts ARM_V4BX relocations whenever it uses a `bx` instruction. Relocations are needed to make the program able to run independent of its memory location. This information is only needed on ARMv4T to allow an ARMv4 application to link with an ARMv4T object. Then every "`bx rm`" instruction is transformed into a matching "`mov pc, rm`" instruction. The elf2flt version included in PTXdist had to be modified to add support for this relocation.

When linking with the POSIX threading library (pthreads) implementation in uClibc, the relocation R_ARM_TARGET1 is used. This relocation was not supported by elf2flt and no fix was found.

5.3 The Final Toolchain

The toolchain is not able to link with the POSIX threading library. The configuration steps and modifications to produce this toolchain are described in Appendix B. The toolchain is named *arm-rav-uclinux-uclibcgnueabi*. Its name is of the format: [architecture]-[vendor]-[os]-[systemtype].

5.4 Testing the Toolchain

As the generated toolchain is a product of bundling a set of already verified tools, only the interaction between these tools is verified for this project. This toolchain needed to be compatible with the ABI of the ported Linux kernel. This could be verified by compiling a set of userland applications and verifying their behavior.

As a part of extending the ported Linux kernel with a large set of UNIX userland tools, the BusyBox project was planned to be built. Since BusyBox contains a large set of Linux applications with heavy use of both Linux system calls and standard C library functions, it was decided that building and running BusyBox would give a satisfactory level of confidence in the correctness of the toolchain. As described in the next chapter, the toolchain proved to be capable of building BusyBox.

Chapter 6
Building a Linux Distribution

In order to provide a Linux distribution, it would be necessary to build a large set of userland applications. To build these userland applications, the userland toolchain generated would need to be utilized. Building and executing a large set of userland applications would increase the confidence in the correctness of both the ported Linux kernel and the toolchain.

The resulting Linux distribution is based on the Linux kernel and BusyBox and is called *SHMAC Linux*.

6.1 Creating a Userspace Environment

In order for any non-kernel code to be executed by the Linux kernel, it needs to be placed in the root file system. The root file system can be realized in several ways. The simplest way is to use the kernels initramfs functionality as the root file system. This filesystem is contained within the kernel binary file, and is unpacked during the Linux kernel initialization phase.

In the case of SHMAC Linux, all userland applications needs to be located within this filesystem. When the kernel is compiled, the files which should be included in the filesystem are located, and packed together with the kernel image. Which files should be included is defined in a special list file as seen in Figure 6.1. This list specifies not only which files should be included, but also their type, permissions and user and group identification numbers.

6.2 Integrating BusyBox with Linux

As seen in Figure 6.1 BusyBox consists of only one file in the SHMAC Linux userspace. Also, a symbolic link is created that makes the BusyBox the standard shell interpreter.

Once the kernel is initialized, it executes the first userland application. In SHMAC Linux, this file is the *init* shell script seen in Figure 6.2. This shell script is interpreted by the BusyBox shell.

```
/* type name path permission userid groupid */
dir /dev 0755 0 0
dir /usr 0755 0 0
dir /usr/bin 0755 0 0
dir /usr/sbin 0755 0 0
dir /sbin 0755 0 0
dir /bin 0755 0 0
dir /proc 0755 0 0
dir /sys 0755 0 0
nod /dev/console 0755 0 0 c 5 1
nod /dev/loop0 0644 0 0 b 7 0
file /bin/busybox ../userland/busybox-1.22.1/busybox 0755 0 0
file /init ../initramfs/init 0755 0 0
slink /bin/sh /bin/busybox 0755 0 0
```

Figure 6.1: Example of initramfs configure list.

The first command installs the BusyBox applications. Once this command returns, all BusyBox applications are available for use. The next command mounts pseudo file systems, exposing information about processes and devices. For the BusyBox applications to be callable directly from the shell without giving their absolute path, the `PATH` variable is set to contain the locations of the BusyBox applications. The last command of the shell script executes BusyBox' initialization program, which again starts the shell. Once the shell has been started, the system is ready for user interaction.

6.3 Using SHMAC Linux

A set of tools has been developed to simplify the process of running SHMAC Linux. The SHMAC system is contained on a development system accessible via serial communication from the host system. These scripts automate the process of compiling the bootloader, LDT, Linux kernel and userland applications; uploading them to the SHMAC host; placing them in the correct place in memory and start execution. In addition to these scripts, a guide has been written to aid further development of this project. This guide is found in Appendix A.

User tests have been performed. In these, SHMAC developers without knowledge about SHMAC Linux have followed this guide and executed the scripts. Results indicate that SHMAC Linux is easy to install and use.

```
#!/bin/busybox sh

/bin/busybox --install -s

clear
echo "SHMAC Linux"

#Mount pseudo file systems
mount -t proc proc /proc
mount -t sysfs sysfs /sys

export PATH="/bin:/sbin:/usr/bin:/usr/sbin"
exec /sbin/init
```

Figure 6.2: SHMAC Linux *init* script.

Chapter 7
Benchmarking

A set of benchmarks provided by [31] was built to run on both SHMAC Linux and on bare metal SHMAC. These benchmarks were executed on Rav and an improved version of Rav named Turbo-Amber [16].

7.1 Benchmark Set

bitcount performs a set of bit counting algorithms. Seven different algorithms are each performed over 75 000 iterations.

qsort sorts strings by using the `qsort` function provided by the standard C library. An input of 10000 lines was used.

basicmath solves a set of equations and converts between radians and degrees. This benchmark contains floating point operations, which is not supported in hardware by SHMAC.

dhrystone is a synthetic integer benchmark. It calculates a value called "Dhrystones per second" which is the number of main loop iterations per second.

dijkstra finds the shortest path in a statically defined two dimensional map.

sha1 performs the SHA1 hashing algorithm on a static set of data.

7.2 Comparing Benchmark Results on Linux and Bare Bones

To investigate the overhead introduced by Linux, the set of benchmarks was executed on both SHMAC Linux and bare metal SHMAC. The results from this can be seen in Table 7.1 The results show the performance decrease of Linux execution compared to bare metal execution.

The performance decrease is calculated by:

$$\text{Performance decrease} = \frac{\text{Linux execution time}}{\text{bare metal execution time}}$$

Benchmark	Linux	Bare Bones	Performance Decrease
bitcount	24.05 s	21.92 s	1.10
qsort	85.30 s	14.49 s	5.89
basicmath	121.90 s	181.18 s	0.67
dhrystone	1.04 DMIPS	1.81 DMIPS	1.74
dijkstra	212.04 s	146.12 s	1.45
sha1	189 iterations/sec	203 iterations/sec	1.07

Table 7.1: Comparing benchmark results from Linux and bare metal.

The difference between the results from Linux and bare metal varies from a performance loss of 0.67 in basic math to a speedup of 5.89 in qsort. A large difference between executing the benchmark on Linux and bare metal is that they are built with two different toolchains.

As described in chapter 5, different standard C library implementations can be used in different toolchains. The bare metal benchmarks have been compiled with a toolchain that uses the newlib standard C library implementation, while the Linux benchmarks have been compiled with a toolchain that uses the uClibc standard C library implementation. Since the implementation of the various standard C library calls vary, it is not meaningful to compare the results from benchmarks compiled with different standard C library implementations.

7.3 Comparing Performance Increase Achieved on Linux and Bare Bones

It is clear that comparing the results from benchmarks compiled with different standard C library implementations does not produce comparable results. However, for SHMAC it is interesting to see the performance increase achieved when executing the same benchmark on different hardware. It would therefore be interesting to see if executing benchmarks on two different hardware setups would produce the same performance increase when running the benchmarks on bare metal SHMAC and SHMAC Linux. That is, if the introduction of a hardware accelerator yields a 2x performance increase when performing the benchmark on bare metal, is the same performance increase found when executing the benchmark on SHMAC Linux.

Turbo-Amber is an improved version of Rav [16]. It extends Rav with an instruction buffer between the fetch and decode stage, and branch predictors.

Table 7.2 and Table 7.3 shows the comparison of the benchmark results from the Rav and the Turbo-Amber processor when executed on top of Linux and bare metal

respectively. The tables show the results achieved in the different benchmarks, and the performance increase achieved by Turbo-Amber. It is worth noting that the benchmarks have all been run with no cache due to a bug in the cache. This means that the performance increase achieved here does not give an accurate speedup for Turbo-Amber compared to Rav.

The performance increase is calculated by:

$$\text{Performance Increase} = \frac{\text{Rav execution time}}{\text{Turbo-Amber execution time}}$$

Linux

Benchmark	**Rav**	**Turbo-Amber**	**Performance Increase**
bitcount	24.05 s	4.70 s	5.11
basicmath	121.90 s	25.98 s	4.69
qsort	86.81 s	21.39 s	4.05
dhrystone	1.04 DMIPS	3.42 DMIPS	3.29
dijkstra	152.00 s	38.40 s	3.95
sha1	189 iterations/sec	766 iterations/sec	4.05

Table 7.2: Performance increase measured when executing benchmark on top of Linux.

Bare Bones

Benchmark	**Rav**	**Turbo-Amber**	**Performance Increase**
bitcount	21.92 s	4.50 s	4.87
basicmath	181.00 s	40.81 s	4.43
qsort	14.49 s	4.16 s	3.48
dhrystone	1.81 DMIPS	6.51 DMIPS	3.59
dijkstra	146.12 s	36.7 s	3.98
sha1	203 iterations/sec	841.00 iterations/sec	4.14

Table 7.3: Performance increase measured when executing benchmark on top of bare metal.

The benchmark results in Table 7.2 and Table 7.3 shows that the performance increase is not equal when the benchmarks were executed on bare metal SHMAC and SHMAC Linux. In order to investigate the difference between the performance increase values, the percent difference was calculated. The percent difference has been calculated by dividing the difference of the performance increase (PI) with the average of the performance increase.

$$\text{Percent Difference} = \frac{\text{difference}}{\text{average}} \cdot 100\ \% = \frac{\mid \text{PI Bare Bones} - \text{PI Linux} \mid}{\frac{\text{PI Bare Bones} + \text{PI Linux}}{2}} \cdot 100\ \%$$

Benchmark	**Percent Difference**
bitcount	4.13 %
basicmath	3.72 %
qsort	3.12 %
dhrystone	2.25 %
dijkstra	2.96 %
sha1	3.04 %
Average	3.20 %
Variance	0.43 %

Table 7.4: Percentage difference between executing benchmark on Linux and bare metal.

From Table 7.4 it is clear that the percent difference between the two performance increase values is small, with an average of 3.20% and a variance of 0.43%. This shows that when the benchmarks have been built using the same toolchain and executed on two different hardware platforms, the performance increase of the hardware can be correctly captured.

Chapter 8
Discussion

This project has been executed in four stages: hardware modification, porting, toolchain creation and operating system building. The results from these stages are Rav, SHMAC support in Linux, a userland toolchain and SHMAC Linux, respectively. These four results have each placed a set of requirements to the later stages of the project. Also, the order in which the tasks were executed has had a big effect on the final result of the project. This chapter discusses the process of this project, the design choices made, their consequences and the results achieved.

8.1 Performing a Bottom Up Project

The nature of this project makes a bottom up approach the only logical choice; only by starting at the bottom would it be possible to verify the correctness of the next task. Once the new processor tile, Rav, was in place, the porting process of the Linux kernel could be verified by executing the kernel code on a SHMAC system containing the Rav processor tile. If the final execution platform for the Linux kernel had not been finished, it would not have been possible to verify that the port was correct. Again, once the Linux kernel had been ported, it was possible to verify that the toolchain built was correctly configured.

8.2 Top Down Verification

In addition to this bottom up verification, a top down verification takes place once the later tasks are finished. Figure 8.1 shows how tasks identified in the assignment interpretation is based on, and hence verifies, the results from the previous tasks.

8.3 Building a Toolchain in Parallel

By experience, the authors were aware that building a toolchain for an MMU-less architecture was difficult, but also required. The process of building a toolchain was started as soon as the work on porting Linux started. There existed several tools for building a toolchain, most of which also support building an entire Linux distribution. Testing these tools to see if they are able to produce a toolchain

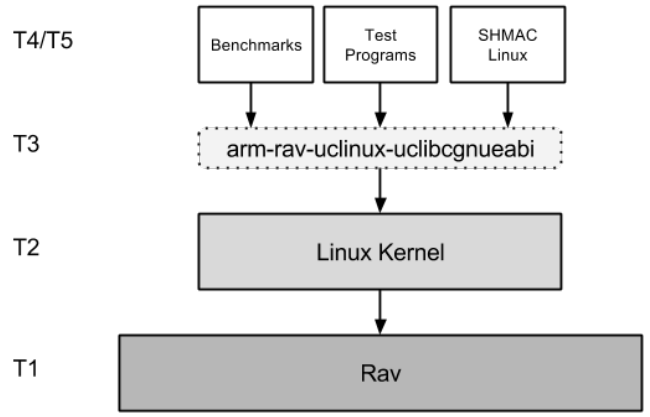

Figure 8.1: Top down verification.

compatible with the results from this project proved to be a tedious and time consuming process. It took several months from the first tools were tested to a working toolchain had been created. Since this was done in parallel with porting the Linux kernel, once the first version of a compatible toolchain had been built, it was possible to verify its correctness.

8.4 Design Choices

This section will present the biggest design choices taken during this project, and emphasize on their motivation and their consequences.

8.4.1 Supporting ARMv4T

As the assignment text for this project states, ARMv3 was the target ISA. During the initial research, it was found that this ISA had not been used in any of the processors supported in the latest version of the Linux kernel. The last version to claim ARMv3 support was 3.9, which was considered to be new enough to fulfill the requirement from the assignment text that a "recent" version of the Linux kernel should be ported. As explained in section 8.3, the work on building a toolchain started almost as soon as ARMv3 support had been added to the SHMAC processor tile. During the search for a framework for building a toolchain, it was found that none of the available tools had support for ARMv3. This was a result of the `bx` instruction used in the standardized AAPCS which was missing in ARMv3. An operating system kernel without a compatible toolchain is not of any use, since its would not be possible to utilize its system calls, so after looking at the documentation for ARMv4T it was decided that this should be the target architecture.

8.4.2 Including an FPGA Specific Multiplier

During the effort to increase the performance of the Amber core by Akre and Bøe [16], an FPGA specific multiplication unit was added to the SHMAC processor tile. When the choice was made to support ARMv4T in Rav, it was found that one of the biggest changes from ARMv3 was support for long multiplication and long multiply accumulate. Instead of extending the Amber multiplication unit to 64-bit, which would require 65 cycles to complete a long multiplication instruction, it was decided that an FPGA multiplier should be used instead. This did not only increase the performance but also simplified the development of the ARMv4T support. The multiplication unit was realized through a cooperative effort with Akre and Bøe [16]. Akre and Bøe supplied the multiplication unit according to an agreed upon interface. The multiplication unit was then integrated in the Rav processor.

8.4.3 Building BusyBox for SHMAC

The main goal of this project was to port the Linux kernel to SHMAC to simplify the development process for SHMAC researchers. Since the mandatory tasks were completed within the time frame it was decided that an attempt should be made to build the entire BusyBox suite for the ported Linux kernel. Since a toolchain capable of building user applications for the ported Linux kernel had been generated, it was possible to build BusyBox for SHMAC. BusyBox improved the usability of the final result of this project.

8.5 Results

This section discusses the result of each of the tasks interpreted from the assignment text.

T1 Modify the Amber processor tile so that it supports the ARMv3 ISA. The Amber processor tile was upgraded to *Rav*, a new processor tile that first supported ARMv3. It was found that ARMv3 did not support the ARM architecture procedure call standard, hence it proved to be problematic to generate the required toolchain for this architecture. Therefore, *Rav* was updated to support ARMv4T. The Rav processor tile was tested using both instruction tests and system tests. As a complete Linux distribution has been executed on Rav without issues, the confidence in its correctness is great.

The work of updating the Amber processor tile proved to be simpler than expected. There are several reasons for this. ARM provides well written documentation for almost all of its ISAs, which makes it simple to produce an accurate set of modifications required for upgrading the ISA support. The Amber project provided a powerful framework for verifying its processor core that was easy to extend. Had this framework not been provided, a similar framework would have to be built from scratch for the upgrade of the Amber processor tile to be possible. Lastly, the

authors had previous experience with constructing and verifying an implementation of the classic five stage RISC pipeline. This provided the authors with knowledge essential for this task.

T2 Port a recent version of the Linux kernel to this new processor tile Once Rav had been successfully implemented and tested, it was possible to use it as a test platform for this task. The version of Linux ported was the 3.12.13 version, which at the time was the newest long term version of the Linux kernel available. The porting was done by following the guidelines, practices and examples for the newest ARM SoC platform support in the Linux kernel.

First, basic platform initialization and debug support was added for the SHMAC platform. Then device drivers was added for the SHMAC hardware. The ported version of Linux was tested by running application that uses the core subset of the system calls. A set of user applications were executed with system call tracing to verify a set of 38 high importance system calls.

Compared to the work done by the authors in [17], where the Linux 2.4.28 kernel was ported to the SHMAC platform, this task was simpler to execute and required less work. One of the reasons for this is that the Linux kernel has undergone major changes since then to make it portable. Another reason is that the target architecture for the port was fully supported by the Linux kernel version ported, with the exception of the SHMAC specific drivers. The decision to provide ARMv4T support in Rav was vital for this task, since support for ARMv3 architectures was poor.

T3 Provide a toolchain which is compliant with the ported version of Linux. For the work done in **T2** to be of any use, a toolchain compatible with the ported Linux kernel needed to be provided. Without such a toolchain, it would not be possible for SHMAC developers to utilize the system call interface provided by the kernel. By using the cross-compiler generation tool Crosstool-NG alongside a set of tools from the uClinux project, uClibc and elf2flt, it was possible to build a toolchain that was fully compatible with the ported Linux kernel. The toolchain was verified by using it for building a complete Linux distribution, SHMAC Linux, hence the confidence in its correctness is great.

As with the previous task, this task was simplified by the ARMv4T support provided by Rav. Still, this proved to be the most time consuming task of this project. A reason for this could be the time consuming nature of the process of building and verifying toolchains. Finding a combination of configurations which did not produce a compilation error during the build process was difficult. Once such an error free configuration had been found, it would take 30 minutes to complete the build process. Now that it had been built it could be tested by building an application which utilized system calls.

T4 Test and Benchmark the new processor tile and the ported version

of the Linux kernel A set of benchmarks provided by Wikene [31] was ported to both bare metal execution on Rav and to run on the ported Linux kernel. These benchmarks were executed on both the standard Rav processor tile and a faster version called Turbo-Amber, provided by Akre and Bøe [16]. The motivation behind executing the benchmarks on these different platforms was to investigate the overhead introduced by Linux, and to argue that the tools provided by this project did not perturb the results produced by the benchmarks.

The percent difference was found when measuring the performance increase of two different processor tiles on bare metal and Linux. This percent difference was low, which shows that Linux works well as a benchmarking platform for hardware.

T5 Build a minimal Linux distribution capable of user interaction. Once the kernel had been ported and the toolchain had been verified, it was desirable to build a large set of user applications. This was desirable not only to verify the correctness of the toolchain, it would also allow for a low control high coverage verification of the Linux system calls. BusyBox was perfect for building such a distribution as it targets embedded architectures with limited memory resources and has support for MMU-less targets.

Using the toolchain produced in **T3**, building BusyBox for the ported Linux kernel was a straight forward job. Only minor modifications to the default configuration were required to build an executable which provided a set of userland applications large enough to make out a complete Linux distribution. Once this was done, only minor modifications to the kernel configuration were required to generate a userspace in which BusyBox could reside.

The result from this task, SHMAC Linux, is a complete Linux distribution.

T6 Enable multicore support in Linux. As SHMAC is a multicore architecture, it was desirable to provide multicore support for SHMAC Linux. Due to time limitations for this project, no implementation was done for this task, only research into the effort required for adding multicore support. Several interesting results were found.

Since SHMAC is a *single ISA* heterogeneous multicore architecture with shared memory, the SMP model of Linux would work with only few modifications. Two possible approaches to provide multicore support were found: modifying the kernel or modifying the SHMAC processor tile.

Since ARMv6 was the first ARM ISA to have SMP support in the kernel, the kernel configuration system does not allow for pre ARMv6 architectures to be configured with SMP functionality. This is easily circumvented by modifying the configuration dependency files. Also, since the locking functionality, upon which SMP support depends, has been implemented using instructions not supported by ARMv4T, all required locking functionality would have to be re-implemented using instructions supported by ARMv4T.

Alternatively, SMP support could be added by using the approach used for this project: modify the hardware to increase software support. By upgrading the SHMAC processor tile once again, and adding support for the ARMv6 ISA, it would possible to add SMP support without modifying the kernel.

T7 Investigate the possibility of mass storage through the SHMAC host controller. Due to time limitations no progress was made on this task.

Chapter 9
Conclusion

For a hardware research project such as SHMAC, it is important that developing tools for testing and verifying suggested designs can be done efficiently. The results from this project provide tools which allows for increased efficiency when developing software for the SHMAC prototype.

The aim of this project was to upgrade the SHMAC processor tile, port the Linux kernel to SHMAC and provide a toolchain for the ported Linux kernel. The new processor tile, Rav, is based on the Amber Core and supports the ARMv4T ISA. SHMAC Linux and the userland toolchain enables SHMAC developers to write applications which utilize the standard C library and Linux system calls.

The lack of tools for the SHMAC prototype prior to this project made it difficult to be efficient when developing software for SHMAC. Also, the ISA supported by the old processor tile imposed restrictions to the set of software tools compatible with SHMAC. By upgrading the SHMAC processor tile to support a newer ISA and porting Linux this project has raised the abstraction level of SHMAC software from bare metal software to Linux software. All components produced have been tested to argue for their usability and correctness. To ensure that SHMAC Linux is an attractive solution for SHMAC developers, effort has been made to simplify the process of installing and using it. This has been done by providing a set of automated tools and user documentation. User tests have been executed to verify that these tools and documents are of satisfactory quality.

During the project, work has been done in all layers of the computer model, from the hardware layer to the application layer. This made it possible to approach challenges from both the top and bottom of the computer model, something which proved to be a necessity for building SHMAC Linux.

Chapter 10
Further Work

This chapter presents possible tasks for increasing the usefulness of the products from this thesis.

10.1 Memory Management Unit

Most operating systems and C libraries depend on the virtual memory abstraction provided by an MMU. This project has shown that it is possible to port Linux to an MMU-less architecture, and also to provide a large set of UNIX-tools. For further development, however, the authors *strongly* recommends that an MMU should be implemented for use with the SHMAC processor tiles.

The OpenRisc project [11] contains an open source implementation of a Harvard architecture based MMU which would be a good fit for the SHMAC processor tile architecture.

10.2 Upgrade Serial Port Driver for new SHMAC Implementation

During this project, the SHMAC architecture has been extended with more TTY channels. The SHMAC serial driver would have to be updated in order to work with this modified hardware interface. By supporting this new SHMAC architecture, it would be possible to integrate the debugger proposed by Seime [28] with SHMAC Linux.

10.3 Enable Mass Storage Support

As the SHMAC architecture only contains 32 MB of memory, the amount of data available for benchmarks is limited. Providing mass storage support for SHMAC Linux would make it possible to execute benchmarks with large data sets.

10.4 SMP Support in SHMAC Linux

Since the SHMAC project targets multicore architectures, it is desirable that SHMAC Linux supports SMP. Section 4.7 presents a detailed description of the tasks required for adding SMP support to SHMAC Linux. This section presents two approaches for adding SMP support: modifying the kernel or modifying the hardware. As this project has shown, modifying hardware could be an efficient alternative to modifying software tools. Also, as explained in the next section, supporting a more recent ISA would still contribute in simplifying development of SHMAC software.

10.5 Upgrading Rav

Further upgrading the SHMAC processor tile to support more recent ISAs would still yield an increase in supported software tools. The authors recommend ARMv6 as the next target ISA, as this would simplify the process of adding SMP support for SHMAC Linux. This project provides valuable information into how such a process should be completed.

Bibliography

[1] Acorn A300 Series Service Manual. http://acorn.chriswhy.co.uk/docs/Acorn/Manuals/Acorn_A300_SMCLSup.pdf. Accessed 2014-04-16.

[2] Acorn A3000 Service Manual. http://acorn.chriswhy.co.uk/docs/Acorn/Manuals/Acorn_A3000SM.pdf. Accessed 2014-04-16.

[3] Acorn A3010, A3020 and A4000 Module Level Service Manual. http://acorn.chriswhy.co.uk/docs/Acorn/Manuals/Acorn_A3010A3020A4000SM. Accessed 2014-04-16.

[4] Acorn A5000 Hardware Guide. http://acorn.chriswhy.co.uk/docs/Acorn/Manuals/Acorn_A500HwGuide.pdf. Accessed 2014-04-16.

[5] ARM Company Milestones. http://arm.com/about/company-profile/milestones.php. Accessed 2014-05-27.

[6] ARM SoC Checklist Presentation. http://elinux.org/images/a/ad/Arm-soc-checklist.pdf. Accessed 2014-05-21.

[7] Buildroot Mailing List, "Buildroot and NOMMU". http://lists.busybox.net/pipermail/buildroot/2013-August/076236.html. Accessed: 2014-02-25.

[8] Buildroot Project Home Page. http://buildroot.uclibc.org. Accessed: 2014-03-09.

[9] Mentor Graphics Sourcery CodeBench for ARM EABI. http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/. Accessed 2014-04-16.

[10] Newlib Home Page. https://sourceware.org/newlib/. Accessed: 2014-03-25.

[11] OpenCores Project Home Page. http://opencores.org. Accessed: 2014-05-19.

[12] SHMAC Project Home Page. http://www.ntnu.edu/ime/eecs/shmac. Accessed: 2014-03-03.

[13] Top 500 Project Statistics. http://www.top500.org/statistics/overtime/. Accessed 2014-04-28.

[14] uClibc Homepage. http://www.uclibc.org/. Accessed: 2014-03-25.

[15] W3Techs Web Technology Surveys. http://w3techs.com/technologies/overview/operating_system/all. Accessed 2014-04-28.

[16] Anders T. Akre and Sebastian Bøe. Turbo Amber, A high performance processor core for SHMAC. Master's thesis, Norwegian University of Science and Technology, 2014.

[17] Håkon Ø. Amundsen and Joakim E.C. Andersson. SHMAC Operating System. Technical report, Department of Computer and Information Science, Norwegian University of Science and Technology, 2013.

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

[19] ARM. *Procedure Call Standard for the ARM Architecture*, November 2012.

[20] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, Oct 1974.

[21] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376, June 2011.

[22] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[23] VLSI Technology Inc. *ACORN RISC MACHINE (ARM) FAMILY DATA MANUAL*, 1990.

[24] Kernighan, Brian W and Ritchie, Dennis M. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[25] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufmann, 2005.

[26] Leif Tore Rusten and Gunnar Inge Sortland. Implementing a Heterogeneous Multi-Core Prototype in an FPGA, 2012.

[27] Conor Santiford. *Amber 2 Core Specification*, May 2013.

[28] Bjørn C. Seime. Debugger for SHMAC. Master's thesis, Norwegian University of Science and Technology, 2014.

[29] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[30] Linus Thorvalds. Linux: a portable operating system, 1997.

[31] Håkon O. Wikene. Benchmarking SHMAC. Technical report, Norwegian University of Science and Technology, Department of Computer and Information Science, 2013.

Chapter A
Compile and Run Linux

A.1 Setup Enviroment Variables

In order for the scripts to run, the following environment variables needs to be set:

```
export SB=<your shmacbox ip>
export SBUNAME=<your shmacbox login name>
export RL=<path/to/repo/software/linux> (no slash at end)
export RAV_CROSSTOOL=arm-rav-uclinux-uclibcgnueabi-
export ARCH=arm
```

(Put these in your bashrc)

A.2 Toolchain

Follow the guide at Appendix B for toolchain setup. (you will need both kernel and userland toolchain).

A.3 BusyBox

Download and compile BusyBox.

```
cd $RL/userland/
wget http://www.busybox.net/downloads/busybox-1.22.1.tar.bz2
tar xfj busybox-1.22.1.tar.bz2
cd busybox-1.22.1
make CROSS_COMPILE=$RAV_CROSSTOOL defconfig
make CROSS_COMPILE=$RAV_CROSSTOOL menuconfig
```

- BusyBox Settings —>
  - Build Options —>

        * CHECK Build BusyBox as a static binary
        * CHECK Force NOMMU build
        * UNCHECK Build with Large File Support
        * Additional CFLAGS (-elf2flt)
      - BusyBox Library Tuning —>
        * UNCHECK Tab Completion
  - Networking Utilities —>
    - UNCHECK *everything*

Exit and save.

```
make CROSS_COMPILE=$RAV_CROSSTOOL SKIP_STRIP=y
```

A.4 Linux Kernel

Download and compile Linux. The patch file 0100-ravlinux.patch contains all the changes made by this project.

```
cd $RL
wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.12.13.tar.xz
tar xJf linux-3.12.13.tar.xz
cd $RL/linux-3.12.13
patch -p1 < ../patches/0002-ARM-deprecate-mach-timex.h-for-
    ARCH_MULTIPLATFORM.patch
patch -p1 < ../patches/0003-ARM-make-mach-xyz-Makefile.boot-optional-
    for-ARCH_MU.patch
patch -p1 < ../patches/0020-ARM-efm32gg-dk3750-add-simple-framebuffer.
    patch
patch -p1 < ../patches/0100-ravlinux.patch
make shmac_defconfig
make menuconfig # (if you wish to customize anything)
make Image -j4
```

A.5 Run

A script named ravrun has been created to make it easy to run Linux on SHMAC. Add it to the path:

```
export PATH=$RL/bin:$PATH
```

For a first time run, use

```
ravrun ldrsbu

Usage: ravrun {[rldbush] || m [SIZE_IN_BYTES]}
   r - Run
   --------------------
   l - Compile and upload LINUX
   d - Compile and upload DEVICE TREE BLOB
   b - Compile and upload BOOTLOADER
   --------------------
   u - Compile and upload all userland applications
   s - Copy run script to host
   m [bytes] - Dump bytes from memory
   h - display this help message
```

A.6 Building Userland Applications

Add a folder in $RL/userland for your application

```
mkdir $RL/userland/hello-world
```

Create the source file hello-world.c

```
#include <stdio.h>
int main(int argc, char* argv[]){
  printf("hello world\n");
  return 0;
}
```

Create Makefile

```
SRC=hello-world.c              # source files
TGT=hello-world                # the target file
include ../include/common.mk   # include common make rules
```

```
make
```

Add entry in $RL/initramfs/initramfs.list

```
<type> <file name> <path to executable> <permission bits> <userid> <
    groupid>
file /hello-world ../userland/helloworld/hello-world 0755 0 0
```

Compile and Run Linux.

```
ravrun lur
```

Execute program in shell

```
# ./hello-world
hello world
#
```

A.7 Miscellaneous

Compiling with -lpthreads does not work due to R_ARM_TARGET1 relocation is not supported by elf2flt.

Chapter B
Toolchain Guide

This guide explains how to setup the kernel and userland toolchain.

B.1 Kernel Toolchain

Download and install the arm-eabi-none toolchain at sourcery.mentor.com. Make sure to add it to your $PATH so that its callable from anywhere. Alternatively, a pre built toolchain can be downloaded using "scp <username>@login.idi.ntnu.no:/home/felles/card/toolch rav-uclinux-uclibcgnueabi.tar.gz".

B.2 Userland Toolchain

A prebuilt version of the userland toolchain an be downloaded.

```
1 ``scp <username>@login.idi.ntnu.no:/home/felles/card/toolchain/arm-rav
      -uclinux-uclibcgnueabi.tar.gz''
```

If this does not work, or you wish to customize the toolchain, follow the next steps.

B.2.1 Setup

Create, and enter a workspace folder.

```
1 mkdir $HOME/toolchain-rav
2 export TC=$HOME/toolchain-rav
3 cd $TC
```

Create a source folder.

```
1 mkdir src
```

Download and< extract all required sources (uClibc 0.9.33.2, Linux 3.12.13, crosstool-ng 1.19.0)

```
cd $TC/src
wget http://www.uclibc.org/downloads/uClibc-0.9.33.2.tar.xz
tar xf uClibc-0.9.33.2.tar.xz -C ../
wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.12.13.tar.gz
tar xzf linux-3.12.13.tar.gz -C ../
wget http://crosstool-ng.org/download/crosstool-ng/crosstool-ng
    -1.19.0.tar.bz2
tar xjf crosstool-ng-1.19.0.tar.bz2 -C ../
```

B.2.2 Setup uClibc

In order to generate a valid uClibc configuration, uClibc needs to be matched with the Linux header files, we will set these up now.

```
cd $TC/linux-3.12.13
make mrproper
make ARCH=arm headers_check
make ARCH=arm INSTALL_HDR_PATH=dest headers_install
```

B.2.3 Configuration uClibc

```
cd $TC/uClibc-0.9.33.2
make menuconfig
```

- Target Architecture -> arm
- Target Architecture Feature And Options
  - CHECK 'Build For glseabi'
  - CHECK 'Use BX in function return'
  - Target Processor Endianess -> Little Endian
  - UNCHECK Target CPU has Memory Management Unit (MMU)
  - UNCHECK Target CPU has a floating point unit
  - CHECK Enable full C99 math support
  - Linux Kernel Header Location -> 'pwd'/../linux-3.12.13/dest/include
- General Library Setting
  - UNCHECK PIC

- Thread Support -> older (stable) version of linuxthreads
- CHECK Enable SuSv3 LEGACY functions
- CHECK Enable SuSv4 LEGACY ...
- CHECK Provide libutil
- UNCHECK Enable etc/TZ file support

– Library Installation Options

- UNCHECK Hardwire absolute paths into linker scripts

– Development/debugging Options

- Cross-compiling toolchain prefix -> 'arm-none-eabi-'
- Extra cflags -> ' -mabi=aapcs-linux '

B.2.4 Setup Crosstool-NG

Create a folder in which to install crosstool-ng

```
mkdir $TC/ct-ng
```

Go to the crosstool source folder and install it.

```
cd $TC/crosstool-ng-1.19.0
./configure --prefix=`pwd`/../ct-ng
make
make install
export PATH=$PATH:`pwd`/../ct-ng/bin
```

Create a folder in which to save all tarballs used by crosstool. Preload this directory with elf2flt, and load custom patches to ct-ng.

```
mkdir $TC/ct-ng-src && cd $TC/ct-ng-src
wget http://redmine.idi.ntnu.no/attachments/download/6/elf2flt_shmac.
    tar.gz
tar xzf elf2flt_shmac.tar.gz
mv elf2flt/elf2flt-cvs.tar.gz $TC/ct-ng-src
mkdir $TC/ct-ng/lib/ct-ng.1.19.0/patches/elf2flt
mkdir $TC/ct-ng/lib/ct-ng.1.19.0/patches/elf2flt/cvs
mv elf2flt/*.patch $TC/ct-ng/lib/ct-ng.1.19.0/patches/elf2flt/cvs
rm -rf elf2flt
rm elf2flt_shmac.tar.gz
```

B.2.5 Configuration Crosstool-NG

```
cd $TC/ct-ng
ct-ng arm-unknown-linux-gnueabi
ct-ng menuconfig
```

- Path and misc options
  - Local Tarballs Directory -> " $TC/ct-ng-src "
- Target Options
  - UNCHECK Use the MMU
  - Architecture Level -> 'armv4t'
  - Floating Point: -> softfp (FPU)
  - CHECK Use Thumb Interworking
- Toolchain Options
  - Tuple's Vendor String -> 'rav'
- Operating System Options
  - Linux Kernel Version -> 'custom tarball or directory'
  - Path to custom source, tarball or directory -> ''pwd'/../linux-3.12.13'
  - UNCHECK Build shared libraries
- Binary
  - binutils version -> 2.22
  - UNCHECK binutils libraries for the target
- C Compiler
  - GCC version -> 4.6.4
  - UNCHECK C++, FORTRAN and Java
  - UNCHECK link libstdc++ ...
  - UNCHECK Enable 128 bit long doubles
- C Library
  - Configuration file ->' 'pwd'/../uClibc-0.9.33.2/.config'
  - Threading implementation to use -> linuxthreads
- Debug facilities

    - UNCHECK everything!
  - Companion Libraries
    - MPFR version -> 3.1.2

Since Crosstool-NG assumes that no ARM architectures are MMU-less, we need to tell it otherwise. Run the command:

```
1 sed -i 's/m68k)/arm)/' $TC/ct-ng/lib/ct-ng.1.19.0/scripts/build/kernel
    /linux.sh
```

Which changes line 14 to say 'arm' instead of 'm68k', voila!

B.2.6 Building the Toolchain

```
1 cd $TC/ct-ng
2 ct-ng build
```

Now add the new toolchain to $PATH:

```
1 export PATH=$PATH:$HOME/x-tools/arm-shmac-linux-uclibcgnueabi/bin
```

Chapter C

Instruction Test Cycle Comparison

Test Name	Rav no. Cycles	Amber no. Cycles	Rav Speedup
cacheable_area	941	1074	1.141
ldm3	191	200	1.047
uart_reg	186	187	1.005
cache3	23373	23374	1.000
sbc	342	343	1.003
cache_swap	15198	15197	1.000
flow3	482	458	0.950
add	172	173	1.006
ldm2	251	249	0.992
sub	117	118	1.009
stm_stream	9941	11928	1.200
barrel_shift_rs	93	94	1.011
cache_swp_bug	5458	5459	1.000
bic_bug	126	127	1.008
bl	133	134	1.008
adc	85	86	1.012
ddr33	4712	4713	1.000
irq	23413	23402	1.000
movs_bug	127	128	1.008
conflict_rd	301	301	1.000
strb	265	266	1.004
swp_lock_bug	77	78	1.013
stm2	314	309	0.984
cache2	69	68	0.986
flow_bug	101	102	1.010
ldm1	371	372	1.003

Test Name	Rav no. Cycles	Amber no. Cycles	Rav Speedup
uart_tx	32746	32776	1.001
cache_flush	3673	3962	1.079
cache1	2999	3000	1.000
swp	240	241	1.004
firq	6609	6772	1.025
hiboot_mem	88	89	1.011
ddr32	47123	47124	1.000
undefined_ins	336	390	1.161
irq_stm	1779	2155	1.211
flow2	774	777	1.004
swi	143	146	1.021
ethmac_tx	4648	4653	1.001
stml	1455	1456	1.001
ddr31	22821	22822	1.000
ethmac_mem	17618	17619	1.000
ldm4	221	198	0.896
ldm_stm_onetwo	747	746	0.999
barrel_shift	761	760	0.999
inflate_bug	83	84	1.012
flow1	404	404	1.000
ldr	706	707	1.001
uart_rx	32779	32760	0.999
bcc	45	46	1.022