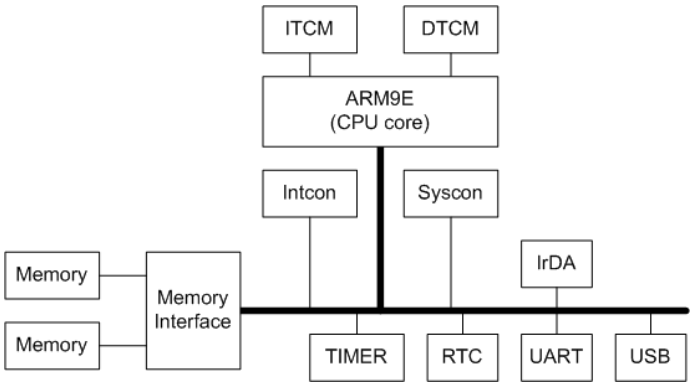
Technical report, IDE0609, December 2005

Baseband simulator

Master's Thesis in Electrical Engineering

Peter De Silva, Roger Pettersson



School of Information Science, Computer and Electrical Engineering
Halmstad University

Baseband simulator

Master's thesis in Electrical Engineering

School of Information Science, Computer and Electrical Engineering
Halmstad University
Box 823, S-301 18 Halmstad, Sweden

December 2006

Description of cover page picture: System to be simulated, a simplified version.

Preface

This report target students who are going to write a similar project and persons associated with Halmstad University, e.g. the thesis tutor, examiner and opponents. Basic knowledge about computer science; preferably ARM CPU's and software development in embedded systems are expected.

"Baseband simulator" is the title of this project which has been initiated by Sony Ericsson Mobile Communications AB in Lund. The work has been carried out at Sony Ericsson Mobile Communications AB in Lund, Sweden. This thesis completes 1.5 years of studies in the field of Electro science at Halmstad University.

To shorten the time to market for a mobile phone some kind of simulator is needed to enable software evaluation and development before the actual hardware is available. Project goal is to be able to simulate a part of the telephone software and being able to see the actual behaviour in conjunction with a software model of for example a UART.

Knowledge is not a one man show; it's based on accumulating what has been learned in the past and by using this accumulated knowledge to come up with new ideas and insights. Therefore we point out some significant persons in our project, without them this would never have been realisable.

Thanks to Anders Åhlander for supervising this project on behalf of Halmstad University.

Special thanks to our supervisors at Sony Ericsson Mobile Communications AB, Carl Christerson and Thomas Fänge who has contributed with their wide knowledge and commitment creating a progressive atmosphere. We would also like to thank the whole MIB department for their moral and technical support.

Peter De Silva & Roger Pettersson
Halmstad University, December 2006

Abstract

Developing software for mobile terminals is a challenging task because the actual hardware is not available at the beginning of the software development phase. Once a hardware prototype is available the software development can continue on that platform. But before that a need for a model of the actual hardware is needed, hence some kind of emulator or simulator needs to be the developed to give the software developers a head start. The aim of this master thesis is to do a market survey of the available simulators for the ARM9E CPU and attached devices in a base-band chip and test their flexibility in terms of adding additional devices (both external and on chip), and also to implement a simulator using the C++ language. The goal is a modular structure for easy addition of certain components such as memory-interfaces, external devices etc. Another important part is the profiling of the executed code to instrument the execution in different ways, and efficiency to allow fast execution. The conclusion of the market study is that due to the high price of these simulators (1.5K€-40k€), we need to design our own simulator. Our simulator consists of different blocks; some of them are merely stubbed while others like the memory and CPU core are modelled more in detail. The performance of the simulator is around 200 KIPS due to the overhead in the debugging functionality. By removing the debugging overhead and optimizing the memory handling we could achieve at least 1 MIPS on the ARM execution and 5 MIPS on the Thumb execution.

Content

1 INTRODUCTION 1

1.1 GOAL 1
1.2 WHY SIMULATION AND WHAT IS IT 2
1.3 APPLICATIONS 3
1.4 TYPES OF SIMULATORS 5
1.4.1 Interpreter 6
1.4.2 Dynamic recompiler 8
1.4.3 Static recompiler 9
1.4.4 Simulator approach 9

2 THIS PROJECT – THE ARMEMU 10

2.1 APPROACH 10
2.2 PROBLEMS TO BE SOLVED 11
2.3 HOW DO WE START 11
2.4 SYSTEM OVERVIEW 12

3 ARM ARCHITECTURE 13

3.1 DEFINITION 13
3.2 ISA FEATURES 13
3.2.1 Programmers Model 14
3.2.2 Coprocessor interface 15
3.2.3 Conditional Execution 15
3.3 COPROCESSOR 15 16
3.3.1 MMU architecture 16
3.4 BUS ARCHITECTURE 17
3.4.1 AMBA bus 17
3.5 THUMB 18
3.5.1 Thumb registers 19
3.5.2 Switching between ARM and Thumb mode 20
3.6 THUMB2 20
3.7 DSP EXTENSION (E) 21
3.8 JAVA EXTENSION – JAZELLE (J) 22

4 RELATED WORK 23

4.1 RELATED WORK 23
4.2 AVREMU 23
4.3 NINTENDO DS EMULATOR 24
4.4 SOFTGUN 24
4.5 VISUAL BOY (GBA) 24

5 IMPLEMENTING THE ARMEMU 25

5.1 TYPE OF IMPLEMENTATION 25
5.2 INTERPRETER 26
5.3 ARM 26
5.3.1 Decoding unit 26
5.3.2 Executing unit 26
5.4 THUMB 26
5.4.1 Macro 26
5.4.2 x86 assembly 27
5.5 OPTIMIZATION 27
5.6 BUILDING BLOCKS 27
5.7 CPU CORE 27
5.8 MEMORIES 27
5.9 IO-DEVICES 27
5.9.1 Real time clock 27
5.9.2 IrDA 28
5.9.3 GPIO 28
5.9.4 Interrupt controller 28
5.9.5 System controller 28
5.9.6 Timer 28
5.9.7 Memory interface 28
5.9.8 UART 28
5.10 MMU 28
5.11 GRAPHICAL USER INTERFACE 28
5.12 DISASSEMBLER 29
5.13 PUTTING IT ALL TOGETHER 29
5.14 WHAT IS SIMULATED AND WHAT IS NOT 30

6 PROFILER 32

7 RESULTS 33

8 DISCUSSION 34

9 CONCLUSION 35

9.1 FURTHER WORK 36
9.2 LESSONS LEARNED 36

10 REFERENCES 37

11 APPENDIX A – ARM ARCHITECTURE VERSIONS 40

12 APPENDIX B – OPEN SOURCE EMULATORS – AN OVERVIEW 42

13 APPENDIX C – COMMERCIAL EMULATORS AND SIMULATORS – AN OVERVIEW 43

Table of figures

FIGURE 1 COMPLETE SYSTEM SIMULATION 3
FIGURE 2 CPU EXECUTION LOOP 6
FIGURE 3 PRIMARY GOAL 11
FIGURE 4 SYSTEM TO BE SIMULATED 12
FIGURE 5 AVAILABLE REGISTERS 14
FIGURE 6 AHB, ASB AND APB IN AN AMBA SYSTEM. 18
FIGURE 7 THUMB REGISTERS IN COMPARISON WITH ARM REGISTERS. 19
FIGURE 8 ARM, THUMB AND THUMB2 PERFORMANCE 20
FIGURE 9 ARMEMU GRAPHICAL USER INTERFACE 31
TABLE 1 PROFILED CODE 32
FIGURE 10 SIMULATED SYSTEM 33

Abbreviations

AHB	Advanced High-speed Bus
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ARM	Advanced RISC Machines
ASB	Advanced System Bus
CISC	Complex Instruction Set Computer
CPSR	Current Program Status Register
CPU	Central Processing Unit
DMAC	Direct Memory Access Controller
DSP	Digital Signal Processor
DTCM	Data Tightly Coupled Memory
FIQ	Fast Interrupt reQuest
GPIO	General Purpose Input Output
GUI	Graphical User Interface
IrDA	Infrared Data Association
IRQ	Interrupt ReQuest
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator
ITCM	Instruction Tightly Coupled Memory
KIPS	Kilo Instructions Per Second
MFC	Microsoft Foundation Classes
MMU	Memory Management Unit
RISC	Reduced Instruction Set Computer
RTC	Real Time Clock
SIMD	Single Instruction Multiple Data
SPSR	Saved Program Status Register
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
VSP	Virtual System Prototyping

1 Introduction

In this section the goals will be explained. A brief introduction to what simulation is and how it differs from emulation. A typical application scenario will be covered. In the end we will go more in to detail of the different types of simulators available and their pros and cons.

1.1 Goal

"The main purpose of this Master Thesis is to implement and evaluate a software simulation of a base band-chip used in coming mobile terminals. The base band-chip is the main core element of the terminal, containing the CPU, several different hardware blocks (ranging from simple interrupt logic to complete DSP's and memory), and control- and data-buses connecting the blocks together. Most of these blocks can be replaced with simple state-machines, but the CPU and control logic must be implemented to allow actual terminal software to be executed in the simulator." [25]

We should investigate if any commercial simulator is good enough to satisfy our modelling needs in terms of flexibility and performance. This simulator should not use any other hardware than the PC it runs on, but if possible should be able to connect external hardware for real life applications.

- Validating the functionality of one or more of these commercial or/and open source simulators.
- Implementing a simulator using the C++ language which will give us enough functionality to run a program compiled for the ARM9E CPU.
- The simulator should be reasonably fast, by that we mean that we should be able to use the mobile terminal graphical user interface once we have come this far.
- Profiling to check the efficiency of the executed code, which will also give us the possibility to instrument the execution in different ways.
- Tracing to enable stepping into and over functions executed by the simulator.

The focus of the goals above are mainly on the baseband chip and some attached devices like memory interfaces, UART's and displays in mobile terminals, but could be extended to include all of the hardware in the mobile terminal.

The result of the evaluation of commercial simulators can be found in Appendix C.

1.2 Why simulation and what is it

Before proceeding further we need to explain the difference between an emulator and a simulator.

The main difference between an emulator and simulator is that emulation is normally done in hardware by “exactly” modelling the behaviour of the actual target. A simulator is a “pure” software implementation that mimics the functions of a given hardware, pure in the sense that it normally carries out all the functions in software but sometimes attaching hardware devices aids the simulator by relieving software of doing certain hardware dedicated tasks and getting a “real” behaviour. But the terms are normally used interchangeably, as can be understood from their formal definitions presented below.

- Emulator; a hardware device, computer program, or system that accepts the same inputs and give the same results on the output as the emulated system.

- Simulator; a device, computer program or system used during software verification, which behaves or operates like a given system when provided with a set of controlled inputs.

Another very important difference between emulators and simulators is the way they gather information about the execution and the limits on the respective approach. Normally when using a hardware emulator, there’s a limit to how many breakpoints that can be handled. In the case of the simulator the number of breakpoints is normally unlimited.

Why do we need a simulator?

Using a simulator to simulate a specific hardware aids the software designers and test engineers in the complex development of software for mobile terminals. Modelling the heart of the mobile terminal namely the BB (baseband) chip using a simulator will also add the functionality of testing future hardware in a much broader sense than using specific hardware which needs to be designed and manufactured, which is a costly process. The simulator can at a very early stage in the development phase help detect certain problems such as code taking to long time to execute or is executed in an inappropriate manner.

An illustrated scenario of a complete system simulation can be seen in Figure 1.

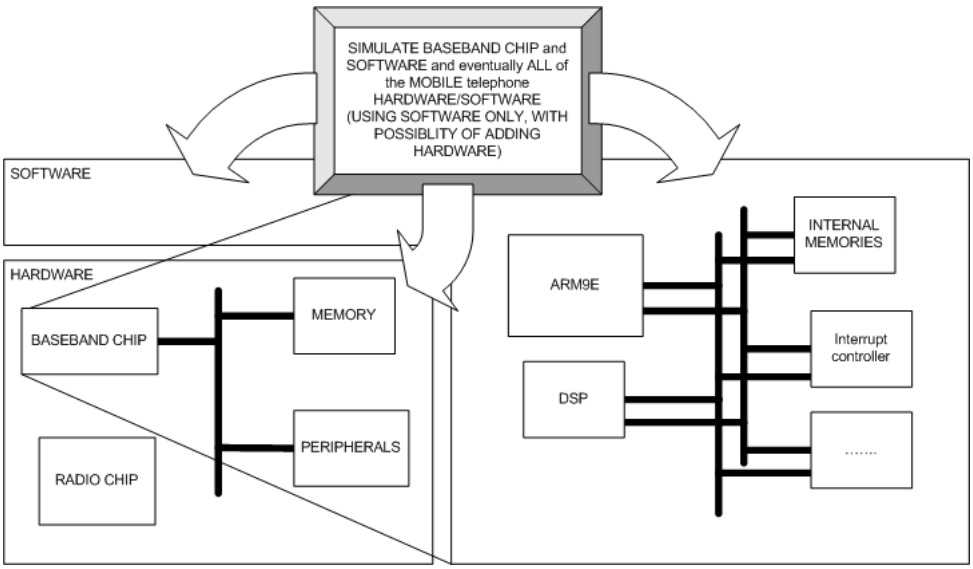


Figure 1 Complete system simulation

1.3 Applications

Today simulators and emulators are used in different areas. In medical care simulators can be used in training, for example simulating different kinds of emergencies. In economics a simulator can be used to simulate the interest over a couple of years based on some decisions. In the development industry simulators can simulate a piece of hardware that is then used to run tests on.

Almost any device that has a processor inside can be emulated; you can emulate a computer system although some systems are very complex which will affect the performance of the emulation.

The most common emulators and simulators are used in areas such as:

- Computer systems:
  UAE Amiga Emulator, it emulates the Amiga 500/1000/2000 and runs on operating systems such as Dos, Windows and MacOS [17].

  The CCS64 V2.0, is a commodore64 emulator that emulates the CPU MOS 6510 cycle exact and runs on operating systems such as Dos and Windows[18].

- Embedded Hardware Simulation:
  SkyEye is open source project for embedded hardware simulation. SkyEye is written in x86 assembly and C. It simulates ARM processors such as ARM7TDMI, ARM920T and StrongARM. A virtual hardware can be created with SkyEye on which operating systems such as ARM Linux, uClinux, uc/OS-II and elastos can be run [19].

- Arcade Videogames and handheld consoles:
  Today there exist numerous emulators for different consoles such as Nintendo and Playstation, almost every console released by Nintendo and Sony has an emulator that can be download for free. Emulators are often used for running software and games, which are called ROMs.

Are Emulators legal?

The case with emulators lies in a grey area; it is legal to emulate proprietary hardware as long as the information which the developer has used to build an emulator on has not been obtained by illegal means. The ROMs on the other hand are illegal to distribute if they are copyrighted which all Nintendo and Playstation games are.

1.4 Types of simulators

The 4 main layers at which a CPU can be simulated are presented below.

1. Hardware simulator – logic level simulator.

2. Host software – using a host software to simulate device drivers
   Fast, but does not simulate hardware, hence no feedback of how the actual hardware reacts in terms of performance (software compiled for the host CPU and not the target CPU).

3. ISS – Instruction Set Simulator – Simulates only the processor.

4. Virtual Processor Model – Using software to simulate ALL hardware. Simulates the hardware from a software point of view. Modification of hardware is easy.

Number 1 is hardware, which is very costly and not really what we're after. Number 2 is in some cases already used but gives a very inaccurate view of the hardware because the software is compiled for the host CPU and not for the target CPU. Hence from the 4 types of simulators, basically 3 and 4 are the ones we are after.

Without the use of a simulator the software designer either has to run the software on a not so accurate model of reality, running the software (compiled for the host its running on) on a CPU using much larger memory and so on. Waiting for the actual hardware is not an option as the project lead times get shorter there's just not enough time.

For the overall performance of the simulator it's important that the CPU core part is efficient. If we have for instance single cycle instructions and a 200 MHz CPU we need to simulate at 200 MIPS to achieve the same performance as the actual hardware.

Different types of CPU simulators are available and which one we are to choose depends on the requirements that we have. Mainly two types of simulators and variants thereof exists today, namely the interpreted and the compiled simulator. The machine code of the simulated CPU core is treated as the source language to the interpreter or compiler. Next we will go through the different types of simulators.

1.4.1 Interpreter

Shown in Figure 2 the normal procedure of the CPU is to fetch, decode and execute each instruction sequentially. An interpreter works in the same way. A setup phase is normally involved in the simulator which consists of setting register values, memory content and timers to their initial values (according to actual system setup). Next the instruction is fetched from the memory model and then the decoder addresses and calls the correct subroutine to execute the instruction as on the actual hardware and updates the registers and memory contents accordingly. This procedure is then repeated until the simulator reaches a steady state (if any).

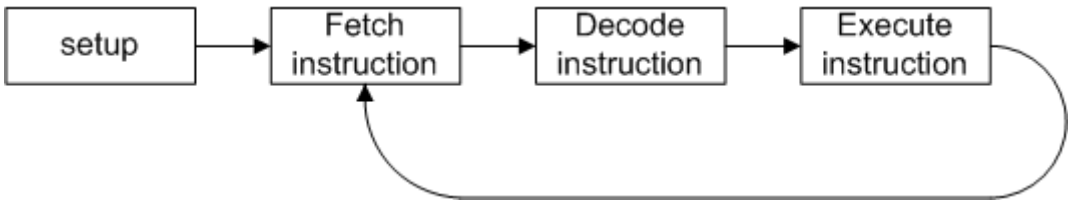


Figure 2 CPU execution loop

From this procedure we can see that an interpreter has several advantages and disadvantages

Advantages:

- Flexible in terms of adaptability (easy to adapt to new instruction sets).
- Simple basic structure of implementation.
- Memory requirements are low due to no redundancy as only the actual elements of the simulated environment are present.
- Handles self modifying code as the decoding is carried out during run-time.

Disadvantages:

- Slow as the decoding is a time-consuming part executed during run-time.
- Static.

A great disadvantage is the fact that an interpreting simulator only cares about the currently executing instruction. As a consequence of this a loop needs to be decoded over and over again which creates an unnecessary overhead in terms of decoding of previously already decoded instructions. The longer the loop is the greater the overhead will be. Locality of reference also known as principle of locality; that is accessing a single resource multiple times is simply not considered in the interpretive approach.

There are three basic types in the principle of locality [26]:

- **Temporal locality**

  Once an instruction has been referenced it's likely to be referenced again sometime in near future.

- **Spatial locality**

  High probability that a instruction is referenced if a instruction near it was just referenced.

- **Sequential locality**

  Memory is normally accessed sequentially which means that the next instruction is likely to come after the current instruction.

The interpretive approach is widely used due to the advantages of being flexible and simple. But it is also the slowest because of the disadvantages mentioned. Simulators that recompile the code to be executed are able to take advantage of the above mentioned principles of locality.

1.4.2 Dynamic recompiler

A dynamic recompiler works with certain parts of the code that are sequentially executed in the simulator. By reusing already decoded parts the dynamic recompiler reduces the overhead in terms of re-decoding already decoded instructions (or at least for that part of the code). Processor caches uses the same technique to store already executed parts to be used in loops and reoccurring functions.

During execution the dynamic recompiler checks to see if this part has been executed before, if not then it will decoded it to the machine code of the executing machine (host). This procedure will create a simulated environment where the fetch, decode and execute sequence overhead is greatly reduced.

Advantages:

- Significantly faster than interpretive simulators.

Disadvantages:

- Large memory overhead due to the need for storing parts of the decoded code for long periods of time.
- Higher complexity.

Dynamic recompilers might be unusable in large embedded applications due to the possible memory overhead.

1.4.3 Static recompiler

The static compiler works in the same way as an ordinary programming language compiler. Instead of decoding the instructions during run-time it actually does all of the decoding during compile-time. This yields a much reduced simulation time due to the fact that the time consuming instruction decoding has been done during compilation. A better optimization is also possible as doing this kind of time-consuming optimization at runtime would probably render a dynamic recompiler too slow.

Advantages:

- The fastest simulation method.

Disadvantages:

- Inflexible due to the demand for static code.

Once the code has been recompiled for the architecture for which it is to be executed on the simulator cannot account for modification during runtime. By adding the functionality from the interpreter or the dynamic recompiling simulator it is achievable to take the dynamicity into account but the static compiler will not take care of self-modifying code like for instance in operating systems where code is frequently changed by loading new programs without the help of this added functionality.

This type of simulator is more suited for the area of trying to execute a single program (static) on architecture.

1.4.4 Simulator approach

The three simulator types mentioned above are to be considered in this project, we will later on describe how and why we have chosen a certain type of simulator. From the above simulators one can say that the interpretive is the most flexible and simple to implement but it also lacks the speed of the other two. The dynamic and static compiled simulator offers more speed at the cost of flexibility and complexity of implementation.

2 This project – the ArmEMU

Designing software for mobile terminals is a tedious task and very difficult as the hardware is normally not available in the beginning of the project, hence no real testing of the software can be conducted at the beginning. By using a model of the actual hardware, one can model the complete system by using a high level language such as C++.

In this project we have written a simulator to comply with the ARMv5T instruction set. More specifically we are using an ARM926-EJS CPU core as reference when considering the hardware structure. We do not intend to model Jazelle (see Chapter 3.8) due to a limited timeframe.

In this chapter the approach that is how we have planned the implementation will be covered as well as what is to be solved initially, how we began our work and a system overview.

2.1 Approach

Depending on how much time each part will take we divided the work into certain areas which are presented in the list below in order of priority. Worth mentioning at this stage is that we focus on functionality rather than accuracy. Certain blocks are only stubbed while others like the CPU core and memories are modelled in more detail.

- CPU core
- Memories
- Buses
- IO devices (mmu, system controller, interrupt controller, RTC, timers etc.)
- IO devices (UART, IrDA, USB etc.)
- Graphical User Interface
- Profiling and optimization

For the CPU core we have chosen to use an interpretive approach, which means that each instruction is fetched, decoded and executed sequentially. We wanted simplicity and adaptability, therefore the choice was an interpretive approach as the static and dynamic recompilers gave us the disadvantage of only running static code not able to simulate self modifying code without having to go through a very complex implementation stage.

The memories are to be implemented as singular arrays attached to an already defined bus structure (written in C++).

Even though the system consists of several bus types we have chosen to model all of these as one interface to just get the functionality of reading and writing data to the IO devices.

The IO devices are memory mapped and so certain memory areas are used as registers and status registers and events are triggered depending on reading or writing to certain areas in memory.

A graphical user interface is to be implemented to simplify the usage of the simulator.

Profiling and optimization will be done if time permits.

Figure 3 below shows the primary goal of this thesis, namely to get the CPU core, memories and a UART up and running. A dissembler is also needed for debugging purposes.

We have chosen to use C++ for simplicity, modularity and flexibility of adding new components as the project progresses.

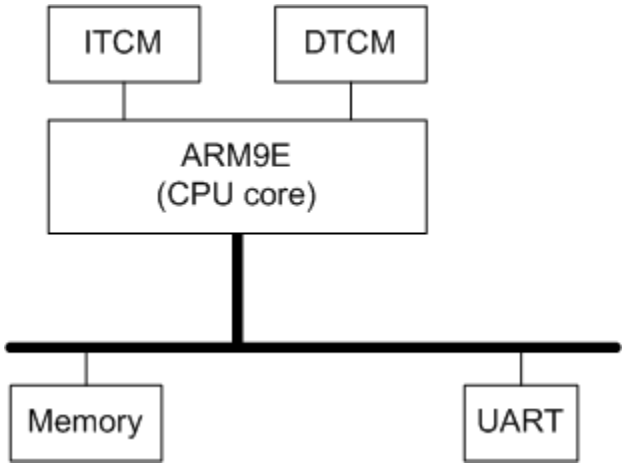


Figure 3 Primary goal

2.2 Problems to be solved

There are initially two main problems to be solved, they are:

- Implementing the CPU core as an interpreter
- Modelling and attaching the different IO blocks to the CPU core

2.3 How do we start

A certain amount of awareness towards supplier's tool chains will of course be a factor to consider when testing the available EDA tools for this particular application of interest. Dependencies of certain modelling languages such as SystemC (which is basically an extension to C++, with the addition of certain components) or pure C, C++ are to be considered. We will as the goal state lean against a wide usage of C++, hence other modelling languages will come in second hand. The platform on which we are to run the simulator is also a factor which needs to be considered, focus here is put on the Windows platform.

2.4 System overview

Below is a rough schematic of what is to be simulated. The main focus is as mentioned previously the CPU core, memory and a UART.

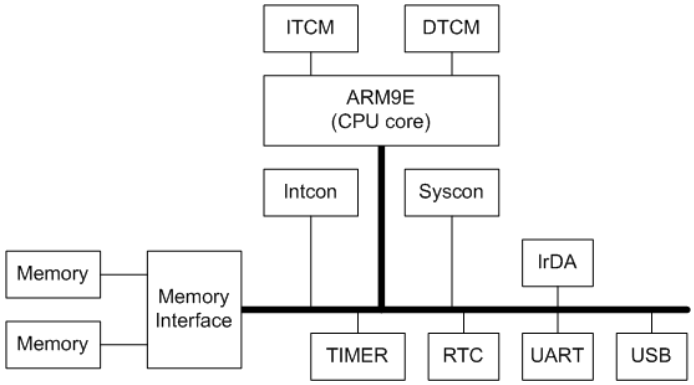


Figure 4 System to be simulated

3 ARM Architecture

ARM which today stands for Advanced RISC Machines originates from a small company named Acorn Computers limited in England from the year 1983. In 1985 Roger Wilson and Steve Furber at Acorn started the development of the ARM1 processor which then stood for Acorn RISC Machine. In 1990 a design team spun off from Acorn. The new design team continued the development of the ARM processor, ARM which now stood for Advanced RISC Machines was also the name of the new company. The first models were released in 1991.

So why does ARM exist today:

Due to High Performance, Low power consumption and low system cost which provides solutions for embedded systems for mass storage, automotive industrial and communication applications. Secure applications and open platforms running complex operating systems are other areas of great success.

Performance improvement while maintaining low power consumption has been the key to success for the ARM processors.

3.1 Definition

The instruction set of a processor is a list of instructions available to the programmer. ARM supports different types of instruction sets for different type of applications and scenarios.

Instructions available during execution are defined through the programmer's model which also defines how the program counter is represented and how context switches affect the register sets. For branching and context switching a stack pointer and link register is available whose contents also depend on which mode the CPU is in.

Without any connection to appropriate devices the CPU won't be very effective. Therefore the architecture specifies how the core interacts with additional devices such as coprocessors. Bus interfaces are also specified through the architecture.

3.2 ISA features

ARM is based on the RISC 1 architecture from Berkeley, apart from some certain parts not being implemented and others being added. ARM can be seen as a mixture of RISC and CISC features. With the combination mentioned above the ARM processor achieves small core size, high power-efficiency and at the same time better code density than a pure RISC processor.

For a list of the available ARM architecture versions, see Appendix A.

3.2.1 Programmers Model

The ARM CPU consists of a 32-bit RISC-processor core which fetches instructions from the instruction memory. 37 32-bit registers are available in total, but a maximum of 17 registers are accessible during execution. Depending on the mode a certain number of registers are common to different modes according to Figure 5. In user mode, 15 general-purpose 32-bit registers from r0 to r14, r15 (program counter, PC) and the current program status register (CPSR) are available.

Reading and writing 8, 16 and 32 bit data types makes the ARM CPU very flexible. System level programming has access to the fiq, svc, abort, irq and undefined modes whilst user level programming only has access to user mode.

Using a pipeline system the ARM can execute certain instructions in parallel. Harvard memory structure is used to increase memory bandwidth and minimize latency. Separate memories are used for instructions (ITCM) and data (DTCM).

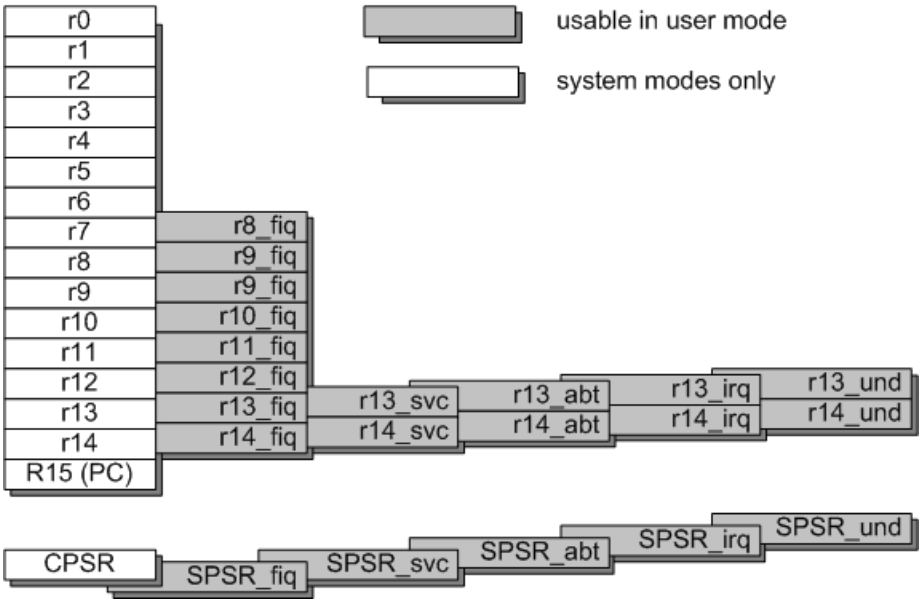


Figure 5 Available registers

3.2.2 Coprocessor interface

It's possible to extend the instruction set by adding hardware coprocessors.
ARM supports the following coprocessor features.

- Up to 16 logical coprocessors
- Each coprocessor can have up to 16 private registers
- A number of instructions are available to access the coprocessor registers and it's specific instructions [12]

Before a specific coprocessor instruction can be performed a handshaking protocol is used to set up the communication between the ARM and the coprocessor.

3.2.3 Conditional Execution

The ARM instructions are all conditional in contrast to the traditional RISC and CISC processors which normally only adds a conditional only to the branch instructions. By using a 4-bit condition code in front of each instruction the following advantages are gained.

- Cuts down significantly on the space available for displacement memory access
- Avoid branch instructions when generating code for small if statements

For a list of the available conditionals and their respective 4 bit code, see [12].

3.3 Coprocessor 15

Coprocessor 15 is an on-chip system control coprocessor; it helps the CPU in performing specific types of operations. The coprocessor controls the on-chip memory management, protection unit, caches, buffers (write, prefetch), branch target cache and system configuration signals. Coprocessor contains 16 registers, each register are 32-bits wide.

Accessing coprocessor 15:

There are two instructions for accessing coprocessor 15. MCR (Move to Coprocessor from ARM Register) is used when a value of an ARM register is passed to the coprocessor register. MRC (Move to Arm Register from Coprocessor) will pass the value from the coprocessor register to an ARM register or to the conditional flag. The only way to access the coprocessor 15 registers for reading or writing is to use the MRC and MCR instructions in a privileged mode. If the processor is in user mode when MRC or MCR are executed an undefined instruction exception will occur [10].

Coprocessor 15 Registers.

- Register 0 is the identification (ID) register and defines the core implementation.
- Register 1. Contains enable/disable bits for caches and MMU. It also has configuration bits for the ARM CPU.
- Register 2-6 8 and 10 handles the memory protection and control.
- Register 7 and 9 handles the control of caches and write buffers.
- Register 13 handles the Fast context switch extension which modifies the behavior of the ARM memory system.
- Register 11, 12 and 14 are reserved of future extensions.
- Register 15 is reserved for implementations defined purposes.

3.3.1 MMU architecture

The memory management unit handles the memory access between the main memory and the processor. The ARM processor generates virtual addresses, which the memory management unit will translate into physical addresses, these physical addresses are then used to identify which memory location is being used.

Coprocessor15 controls the memory management unit by enabling and disabling the memory management unit. The memory management unit will be disabled after a coprocessor reset has occurred.

The Memory Management uses two different blocks of memory these are sections and pages. Sections have 1 MB blocks of memory.

The pages have different sizes, the large pages have 64 KB blocks of memory, and the small pages have 4 KB blocks of memory. Large pages have access control applied to individual 16 Kbytes sub pages and small pages have access control applied to individual 1 Kbyte sub pages [10].

3.4 Bus architecture

Advanced Microcontroller Bus Architecture [10] (AMBA) is an on-chip system bus developed by ARM Limited to set a standard for the interconnection of different blocks in embedded systems. The idea behind the AMBA bus is to be able to interconnect different subsystems with a shared bus; this makes it easier for developers to connect different blocks. AMBA has become a standard for on-chip buses.

3.4.1 AMBA bus

The AMBA bus hierarchy consists of three buses, the Advanced High- performance Bus (AHB), Advanced System Bus (ASB) and Advanced Peripheral Bus (APB). The usage of the AMBA structure is shown in Figure 6.

AHB and ASB are used in high-performance systems, for example microprocessors and Memory controllers. The AHB is used as a high-performance system backbone bus, that supports the connection of processors and both on-chip and external memory.

AHB implements features for high-performance systems such as: high performance pipelined operation, burst transfers, multiple bus masters and split transactions.

ASB is an alternative system bus that is used where the high-performance is not needed, but it also supports the connection of processors and both on-chip and external memory. The ASB has features such as: high performance, pipelined operation, and multiple bus masters.

APB is used to connect peripherals that use low bandwidth, for example UART's, keypads and I/O blocks. The APB acts as a secondary bus from the higher bandwidth pipelined main system bus (AHB, ASB). The APB features are: low power consumption, latched address and control, simple interface and is suitable for many peripherals [16].

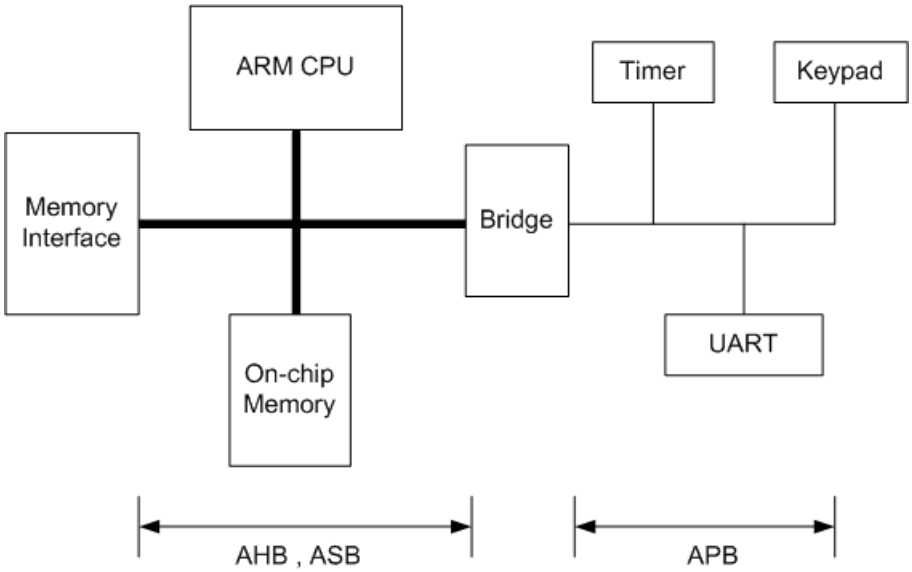


Figure 6 AHB, ASB and APB in an AMBA system.

3.5 Thumb

The Thumb instruction set is a compressed form of the ARM instruction set, and it consists of the most common ARM instructions. The major difference between Thumb mode and ARM mode is the width of the instructions; Thumb mode uses 16-bit instructions instead of 32-bit instructions which is used in ARM mode. That results in better code density when running in Thumb mode instead of the ARM mode. In Thumb mode the instructions are executed unconditionally. Just like ARM, the Thumb mode supports different data types, byte (8-bit), half-word (16-bit), word (32-bit) and 32-bit unsegmented memory.

Both ARM and Thumb are using the same register that makes it easy to pass data between ARM and Thumb mode. Thumb handles the arithmetic and logical operations well, other operations like interrupts and coprocessor 15 has to be managed in ARM mode.

3.5.1 Thumb registers

Thumb registers, R0 to R7 are called Low registers, and they can be accessed directly. The High registers (R8-R15) in Thumb mode can only be accessed by special variants of the MOV, CMP, and ADD instructions. The CPSR and the Low registers in Thumb mode are the same as the CPSR and Low registers in ARM mode see Figure 7. While in Thumb mode the CPSR is not directly accessible (only as status).

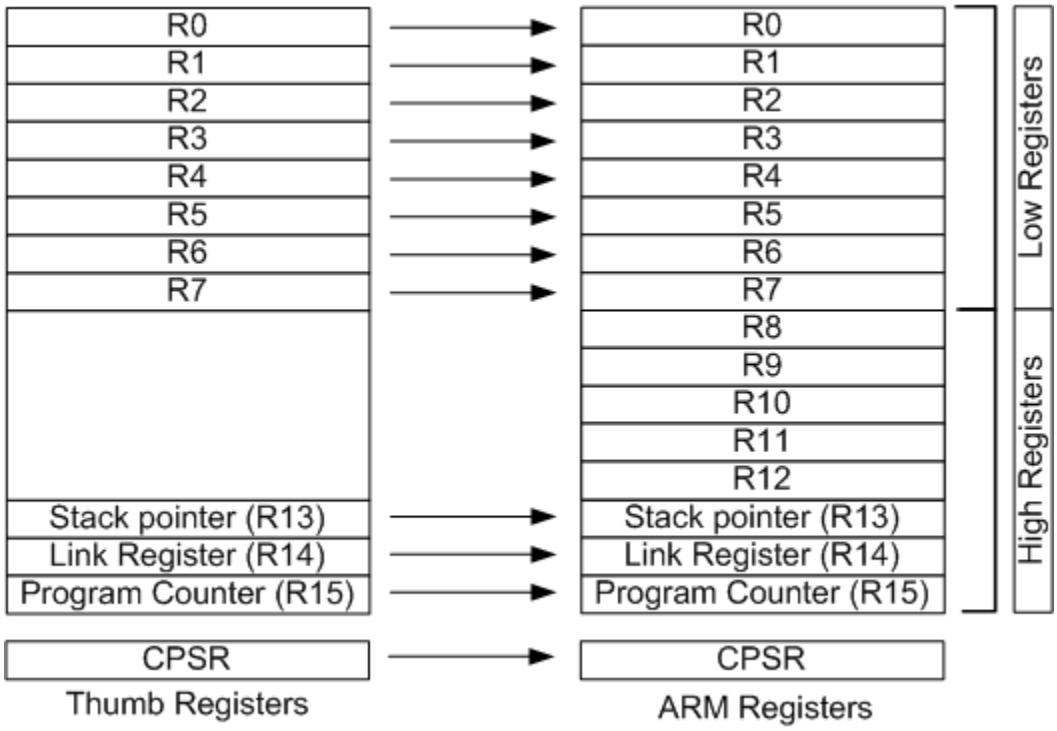


Figure 7 Thumb Registers in comparison with ARM registers.

Thumb mode should be used in systems that have to save power and has a small limited amount of memory. When running a system in Thumb mode will use less memory than running in ARM mode. Thumb mode reduces code size, when running in Thumb state on a system with a narrow (16-bit) data bus and memory port it gives the high performance, since the Thumb instructions are decompressed to their equivalent 32-bit ARM instructions before they are executed.

The best memory and execution performance is obtained by combining the two modes: Thumb for normal program code, and ARM code for timing critical subroutines such as interrupts. ARM and Thumb code cannot be executed simultaneously. The Thumb mode is well suited for mobile phones, handheld consoles like Nintendo DS and Game boy Advance.

3.5.2 Switching between ARM and Thumb mode

As mentioned before Thumb mode does not handle coprocessor 15 or interrupt operations. If running in Thumb mode and an interrupt are encountered then a switch between the modes has to be done, this switch will give an overhead.

To switch between the sets, a branching BX, BLX (branch and exchange, branch link and exchange) or a LDR/LDM (load, load multiple) instruction must be executed; the switch can also occur if the T bit in the SPSR register is set when returning from an exception mode. When entering an exception mode from Thumb mode the switch from Thumb to ARM will automatically occur. When ARM is switched into Thumb mode the T flag is set in the CPSR register [10].

3.6 Thumb2

As of ARM architecture version 5 the Thumb instruction set is version 2.

Thumb 2 is a further development of the Thumb technology, it's a new instruction set for the ARM architecture which uses both 16 bit and 32 bit instructions in the same instruction set. With this new instruction set programmers can mix the16 bit and 32 bit instructions without losing performance due to mode switching.

Thumb 2 does not have the same limitations that Thumb has. For example, when running in Thumb 2 mode the program does not need to switch back to ARM mode when handling interrupts and MMU operations.

Performance is almost the same as the ARM performance and code density is almost as small as Thumb code density [15]. Figure 8 gives a view how the different modes perform relative to each other.

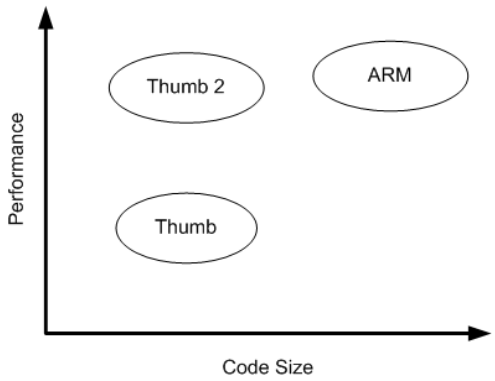


Figure 8 ARM, Thumb and Thumb2 performance

3.7 DSP extension (E)

The DSP extension have made it possible to run applications that require intensive signal processing while retaining the power and efficiency. Applications like speech recognition, speech coders, storage devices, networking, modems, control solutions, and smart phones which require high DSP performance and efficient control implementation are well suited for an ARM processor with the DSP extension. The DSP operating mode adds the E-bit to CPSR and to the architecture version.[10]

The ARM DSP does not perform a well as a pure DSP core; instead the ARM architecture offers an integrated platform which can be used for different implementations.
The benefit with the DSP in the ARM architecture is that that all processing is done on one ARM processor since the ARM has system functions like management of memory, IO's, etc.

A pure DSP implementation requires another microcontroller to manage the rest of the system. The ARM chip area would be smaller than the pure DSP implementation, since the DSP implementation has two processors. A pure DSP implementation with two processors will be more complex than the ARM DSP implementation [23].

ARM DSP features:

- New instructions to load and store:
  The data is loaded and stored more efficiently to get maximum performance of DSP algorithms.

- Zero overhead saturation arithmetic:
  The DSP has saturation instructions (QADD, QSUB, QDADD, and QDSUB) that perform saturation arithmetic more efficient.

- Count leading zeros instruction:
  Count leading zeros gives faster normalization and allows more efficient integer division.

3.8 Java extension – Jazelle (J)

Jazelle extension is an instruction set added to ARM processor to support Java acceleration technology. It gives developers the option to run Java code along side other applications on a single chip. Jazelle allows direct execution of Java byte code. The Java byte codes are 8-bit instructions designed to be architecture independent. The use of Java in hardware will reduce the power consumption, memory access, switching and it will give better performance.
The new java operating mode adds the J-bit to CPSR [10].

4 Related work

In this chapter we will explain what previous work has been done in the field of simulating an ARM based system.

4.1 Related work

Different open source projects have been conducted to see if it's feasible to implement a simulator/emulator in software.

Commercial products such as the ARMulator [22] and MaxSim [5] are fully fledged tools to simulate ARM CPU's and specific blocks connected to the CPU.

Simulators can be further categorized according to the level of simulation, whether at the architectural level or the instruction set, or the techniques used, e.g. dynamic recompilation of parts of the simulated software to natively run on the guest system.

A simulator normally interprets the binary code of software compiled for the target system. The Dynamic Recompilation method involves translating parts of this binary code into native machine code at runtime. Native execution of the recompiled code leads to much faster execution of the simulated software. A lot of simulators are developed using this technique.

As far as implementation is concerned, no single solution for our type of application is available as that would render this project unnecessary. We have looked at a few commercial and open source simulators. Of them we have chosen the open source simulators below for inspiration and also for direct usage of the code provided in the respective projects. In Appendix B and Appendix C a list of all the emulators/simulators that have been considered in this project are presented.

4.2 AvrEMU

AvrEMU is a Sony Ericsson in-house project. From this project we have used the backbone of bus, memory and CPU structure and then attached our models for each of these parts.

4.3 Nintendo DS emulator

DSEmu [9] is an emulator for Nintendo's handheld console Nintendo DS. DSEmu is incomplete at the moment but DSEmu has some devices in place like the disassembler, memory handling and two processors, ARM7TDMI and ARM946. The interesting part to us is the ARM946, which is similar to the ARM926 that we will be simulating. The Thumb instruction set is only 90 % complete, it is written in x86 assembly that will make it fast. The DSEmu does not need a complete Thumb instruction set, only the instructions that are needed have been implemented. That is why the Thumb instruction set is not complete and probably never will be complete. All the functions has not been verified, that is something we have to do. The performance of DSEmu is about 60 MIPS.

4.4 Softgun

Softgun [6] simulates the NS9750 circuit from NetSilicon which uses an ARM926 core, same as the one we are trying to simulate. From the Softgun project we have looked at the instruction decoder and execution unit for the ARM instruction set. Of course the code of each discrete function which executes an ARM instruction can be optimized further, but as far as complexity and implementation method we felt that this part of Softgun would give our project a head start to get the interpreter up and running as fast as possible. According to the author the Softgun performance is around 14 MHz/1 GHz of host CPU.

4.5 Visual boy (GBA)

Visual Boy Advance [7] (VBA) is an emulator for Windows. VBA emulates the Nintendo Game boy Advance that consists of an ARM7TDMI processor and 3,027 Kbits memory. VBA is not complete, but it can still run games at full speed with sound. VBA is able to run Game boy Color games just as the real Game boy Advance. The Visual Boy Advance is written in C++ and x86 assembly.

5 Implementing the ArmEMU

We have chosen to name the simulator ArmEMU, even though it's a simulator, this is to show the heritage from the in-house project named AvrEMU. What kind of principles and how they are implemented is shown in this chapter.

C++ will be used as the preferred language when programming the simulator. Visual studio 6.0 Enterprise edition will be used as development environment. As for the modularity, using an object oriented programming language like C++ will give us the advantage of writing code for easy maintenance and extendibility.

Testing of the software will be done by modelling a small system consisting of the ARM9E CPU and some attached devices like a UART and some memories. Software compiled for the ARM9E will be ran in hardware and compared to results obtained in simulator.

Simulator will be written based on a given backbone, and the output from the simulations will also be based on this backbone which already has the functionality of stepping into, stepping over and showing register contents. Profiling will be implemented if time permits to show how often certain functions are called and to analyse bottlenecks in software. The profiler will also aid us in the optimization of the simulator.

5.1 Type of implementation

We have chosen to implement the ArmEMU with certain aspects in mind, modularity and simplicity. By modularity we mean that it should be easy to add blocks to the IO as well as external and internal memory blocks. When it comes to adapting the interpreter to future ARM architectures assuming the same instruction set is used will probably not be a problem. We do not consider the pipeline and hence do not take the scheduling aspects of running the CPU in to consideration therefore using this simulator for simulating the ARM11 will probably work fine as the main difference is the pipeline. The possibility of executing 1.1 (which is the case in the ARM9E core) instructions per cycle makes it impossible to implement without using some kind of prediction. Extending the parallelism of the simulator is another angle to this project and not considered here.

5.2 Interpreter

In our simulator both the ARM and Thumb interpreter use a lookup table to access a function pointer stored in the respective array for each instruction class. The instruction is fetched and matched against the masking bits in the array and an instruction/instruction class is executed to further decode and execute the instruction with new register and memory contents as a result. Depending on which mode we are in, ARM or Thumb, we use a static lookup table (Thumb) and a dynamic lookup table created during the setup of the simulator (ARM). In the next two subchapters we explain how the interpreter works for both ARM and Thumb.

5.3 ARM

The ARM 32 bit instruction set has a few addressing modes and dependencies that need to be decoded and evaluated upon execution. The decoding and execution units (or functions) are actually tied together in single functions, but we will cover them as two separate units and then we will explain how they fit together. The ARM decoding and execution functions are partly implemented as x86 assembly to enhance the performance.

5.3.1 Decoding unit

The decoder is implemented as a linked list of instructions to be decoded indexed by bit [4:7] and bit [20:27] of the 32 bit ARM instruction and a matching function. This will create a kind of a lookup table where the decoding and execution function classes are addressed using the above mentioned bits.

5.3.2 Executing unit

Execution of each interpreted instruction is carried out by the usage of function pointers to the respective ARM function which executes the decoded instruction.

The decoding and execution units are then tied together in a single function for each instruction/instruction class. The 32-bit instruction is first decoded to extract the operands and registers to use. After decoding the instruction the actual execution is carried out to give the result of the instruction.

5.4 Thumb

In order to get good performance, the Thumb code in our simulator is mainly written with macros that contain x86 assembly code. This makes the execution of Thumb instructions fast. All the Thumb functions are called from an array containing function pointers. To execute a Thumb instruction the opcode is first masked, then a function pointer in the array is called and the instruction is executed.

5.4.1 Macro

There are some disadvantages when using macros. For example when we want to debug the code, we can not step into a macro and because of that it is difficult to debug the Thumb instructions. When arguments are passed through a macro the data types will not be checked, that means that if the input variable is not long enough or if the input variable is a character when it

should be an unsigned integer, the result will be wrong but we will not get any errors when we compile the simulator.

The advantage when using macros instead of functions are that macros do not have the overhead that a function call has, which increases the execution speed.

5.4.2 x86 assembly

The Thumb instruction set is partly written in x86 assembly which gives good performance and is often used in time critical parts of a system, which is why it is used in our simulator.

5.5 Optimization

Optimization by caching certain instructions or/and functions will of course speed up the simulator depending on how often and how effective a certain instruction or function can be cached and used.

Optimizing the simulator by usage of compile time interpretation an not run time as now will of course speed up the simulator, how much will depend on how often a certain instruction/function is executed and how well that particular instruction/function is implemented in the interpreter/executor.

Due to lack of time, optimization has been left out, but has still been considered when writing the code.

5.6 Building blocks

All the blocks presented below are in a usable state though some functionality has been left out as a result of debugging with live hardware.

5.7 CPU core

The CPU core, disassembler and debugger functionality is implemented as a single C++ class.

5.8 Memories

The memories are singular arrays with normalized address mapping; different read and write methods are used depending on whether it’s flash or other type memory being accessed.

At the moment the simulator is writing and reading to the memory byte wise, which has a negative impact on the performance; this will later have to be changed to comply with the byte, half word and word access.

5.9 IO-devices

5.9.1 Real time clock

Real time clock is provided to enable alarms and other time dependent functions, the RTC simply uses the host clock to provide year, hour, minutes, seconds and date to the simulated system.

5.9.2 IrDA

Infrared communication, simply a stubbed IO unit as the functionality is not needed but it might be the case that some registers are updated and read from and therefore we need to model this accordingly.

5.9.3 GPIO

To dynamically map different IO pins to different units a GPIO is available, this needs to be modelled to route the signals to the correct blocks.

5.9.4 Interrupt controller

The interrupt controller block handles incoming interrupts at low and medium latencies. The interrupt controller has two output interrupts to the CPU, one interrupt request (IRQ) and one fast interrupt request (FIQ). Only the basic registers are implemented and the actual function of the interrupt controller is left to be implemented later.

5.9.5 System controller

To enable and disable different blocks and to distribute the clock signals the systems controller also needs to be modelled in a functional matter so that when certain bits are set all the blocks affected needs be taken into consideration.

5.9.6 Timer

The timer is responsible for generating time driven interrupts and time dependent events. Due to dependency on the interrupt controller only the registers have been implemented.

5.9.7 Memory interface

To interface the different types of external memories a memory interface with its own set of registers has been implemented to achieve the correct settings for each memory.

5.9.8 UART

To connect different types of equipment such as RS232 devices, we have implemented a simple model of the UART, mainly to show the system log which is communicated out on one of the UART's.

5.10 MMU

MMU or Coprocessor 15 as it is more commonly known as, is implemented to the extent that we can turn the TCM's on and off. We can also retrieve the different registers contents which is necessary for identification of the CPU. No configuration of memory areas is implemented.
The coprocessor is part of the main class.

5.11 Graphical User Interface

The graphical user interface has been implemented using MFC. It has all the basic functionality of single stepping, setting breakpoint and listing memory contents. The GUI was implemented for ease of use and configuration.

5.12 Disassembler

This part though not contributing to the actual simulation is needed for the purpose of debugging and ease of use. Here we have been inspired by the Visual Boy, Gameboy Advance simulator disassembler which uses a simple masking pattern to decode each instruction. This block was added to the main class and will be adapted to the ARM v5TE architecture used in this project.

It works in very much the same way as the actual interpreter, but with exception that no execution is carried out.

Disassembler works in the following way:

1. Decode the operation using bit 24-27 or 20-27 or the upper 8 bits if it's Thumb code
2. Match the opcode to the operation to be disassembled
3. Extract the operands using masking
4. Print the disassembled code

5.13 Putting it all together

Here is a short summary of how we put our simulator together:

1. In order to get a good understanding of what a simulator is and what it can do, we started reading about simulators and emulators. Since we were going to simulate an ARM CPU we read a lot about the ARM architecture.

2. By looking at different parts in open source projects, we have learned how the parts work and how they interact with each other.

3. We started implementing different parts like the disassembler and interpreter (Thumb and ARM decoding and execution) in separate projects for testing purposes. Then we put the parts together into to one project, this together with the bus, memory and CPU-interface from the AvrEMU project became the foundation of our simulator.

4. After the foundation was in place we added the memory to the project. Now the simulator had basic functionality, it could execute code and read and write from and to the memory.

5. Certain IO devices were implemented; some of the blocks were more complicated to implement than others, for example the interrupt controller (which was not finished). The UART was the most important IO device which was put in place to enable for instance logging.

6. At last we implemented a graphical user interface, which made the simulator easier to use.

5.14 What is simulated and what is not

The ARM9 CPU core and some memories are fully implemented giving a solid backbone to build upon.

A few important blocks have not been completely implemented but skeletons and main part of their functionality is there (mainly registers and a few trigger functions), these are; UART, Interrupt Controller, DMAC, IrDA, USB, TIMER, RTC, and MMU.

Certain functions and aspects of the baseband chip are not modelled due to unnecessary overhead and little gain in knowledge. The ARM9E 5-stage pipeline and scheduling of the fetch/decode/execute chain is not taken into account in ArmEMU, neither is there any cycle accuracy. We simply want to simulate as fast a possible with as little overhead as possible without lacking any function that prevents the phone to start.

All of the blocks have been implemented upon usage in telephone code, which means that we executed the code on the actual hardware and compared with our simulator. After that we have implemented the different blocks as they function in the actual hardware.

The graphical user interface can be seen in Figure 9 which also shows how the registers and memory contents are updated. The LOG window shows the executed instructions and also messages from the different IO-blocks. In the Instant Memory Viewer window it's possible to show the contents of a certain memory address range. STACK shows the contents of the current stack pointer and some word (32-bit) aligned addresses.

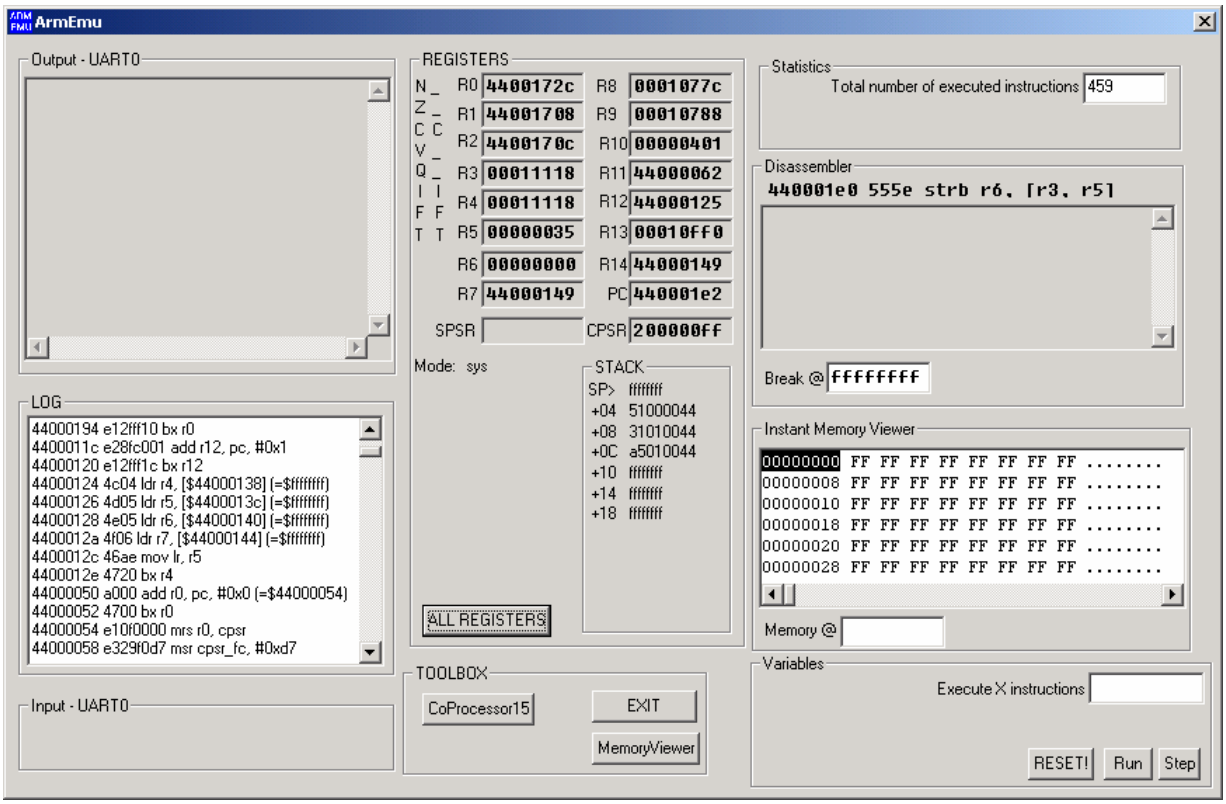


Figure 9 ArmEMU graphical user interface

6 Profiler

The purpose of the profiler is to find bottlenecks in the executed code. A view like the one in Table 1 can easily be generated by using an open-source profiler. This will however in the end result in performance loss, so therefore the profiler should be a compile time included feature.

Function name	Time consumed (% of total)	Time consumed (s)
Function1	30%	30s
Function2	70%	70s

Table 1 Profiled code

It would be a good idea to make the profiler a compile time dependent feature to optimize the performance of the simulator. Otherwise we would need to use conditional statements in the code to turn the profiler on and off which would cost performance.

Due to lack of time the profiler has not been implemented. The estimated time for implementing the profiler is about a week using an already existing open source profiler. Extending the profiler to handle not just functions but on the discrete instructions is also interesting.

7 Results

Simulating a few seconds of the phone software and printing data out on the UART is considered to be goal reached but further work is needed to get the telephone to “standby mode”.

By complete simulator in this context we mean running the phone software to the standby state which means that the phone has booted completely.

The simulator mainly consists of the blocks presented in Figure 10. Some of them are merely stubbed while others like the memory and CPU core are modelled more in detail.

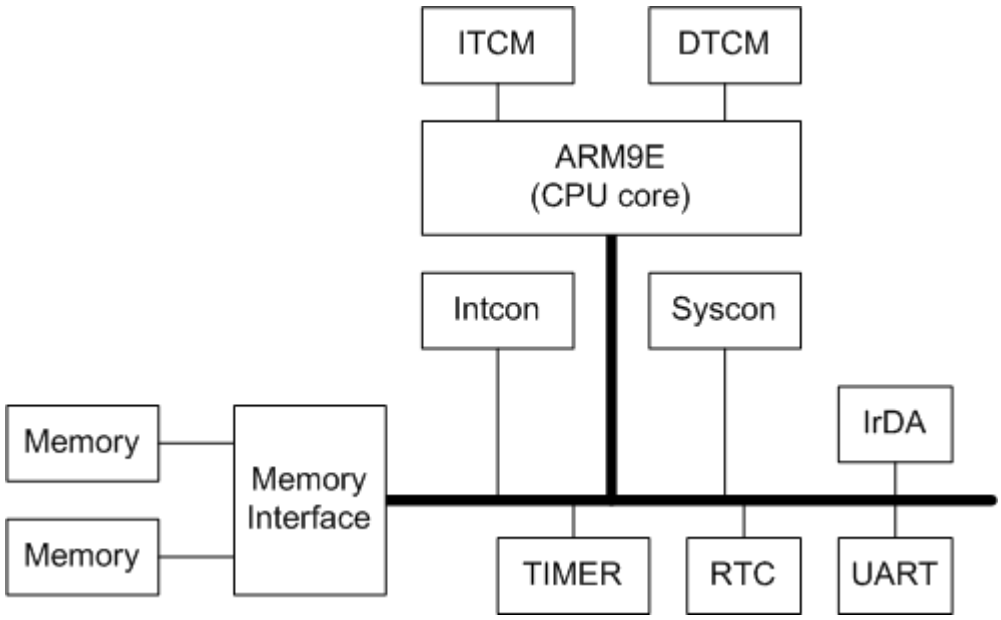


Figure 10 Simulated system

The performance of the simulator is around 200 KIPS which can be considered to be quite low, but we can however greatly increase this figure by removing overhead in the debugging functionality, and we also see a number of optimizations to be done in the way the memory is written to and read from. By removing the debugging overhead and optimizing the memory handling we could achieve at least 1 MIPS on the ARM execution and 5 MIPS on the Thumb execution.

8 Discussion

Attempting to write a simulator that simulates a complete mobile terminal is a daunting task, nevertheless one shouldn't be scared by this as there is nothing as satisfying when you get to put a whole system simulator together.

When approaching a project similar to the one presented in this thesis one should keep in mind that a good overview of the complete system is necessary to write a qualified simulator. Not only the CPU core part needs to be taken into consideration but the blocks that affect the simulator performance like the frequently accessed blocks such as the graphical display and memories. A block that is frequently used will of course create bottlenecks if written poorly.

Extending the interpretive approach to a compile time simulator is another aspect of improving the performance of the simulator, but be careful, self changing code will prohibit the correct behaviour unless some sort of memory matching function is applied to check if the memory contents are consistent (or other function which consider the changing of memory contents).

Getting early results should be avoided to prevent early mistakes from taking up a lot of correction time in the later stages of the development. Think things over both one and two times before implementing.

Implementing a 32 bit instruction set as complex as the one in the ARM9E should not be taken as a simple task, try to find other simulators (preferably open source) and test the performance of those before writing your own.

9 Conclusion

As stated in the results the primary goal has been met and the simulator is running actual telephone code up to a certain state where we lack functionality due to a limited timeframe.

A lot of open-source simulators are available and they are certainly becoming more and more useful as the PC gets more sophisticated in terms of performance. Certainly the performance on the simulator very much depends on the evolution of in this case the ARM processor. If the development of the ARM progresses much faster than the processor for the PC it will probably become less likely that these processors are simulated in real time but we still can do a lot of tests by running at speeds of say 10% percent of the actual running speed.

Although a very extensive work, we have been greatly rewarded when the stage of actually executing the telephone code was reached. We have also learned how a telephone starts up and what is initiated at what time and also how the IO blocks are handled during start-up.

When using x86 assembly, we loose the simplicity of porting the code to other architectures but in some cases we gain a lot of performance. As the focus was put a Windows based system, this was not a major concern, but it still needs some consideration.

This work is far from solved in terms of simulating a complete mobile terminal, but it has definitely created a foundation to build upon. Due to the modularity of the simulator it’s possible to easily add new IO blocks and to extend the interpreter with new instructions/instruction classes.

9.1 Further work

To-do list, in order of priority, according to specification of master thesis.

- Implement the blocks left to enable the boot of a “complete” phone
- Profiler to verify the function effectiveness and execution statistics
- More advanced debugger (memory triggers etc)
- Optimization of the interpretation of the ARM and Thumb instructions

9.2 Lessons learned

When we started we had basically no knowledge about the ARM processor and architecture. We had some basic programming skills. This project was very instructive and we have learned a lot in different areas like:

- CPU and architecture

  ARM architecture and especially the ARM9E CPU core

- Simulators

  During this project we have learned how different simulators works and how to build a CPU core simulator from scratch. Different ways of implementing the simulator has been studied.

- Programming

  Wider knowledge about C and C++ has been gained, and x86 assembly has to some extent been used which was a new experience.

- Working in a large project

  Large project was something that we are not used to; most of the projects in which we have been involved have been smaller school projects. In large projects as this, it is important to have a good plan which you follow because there is a limited amount of time and certain tasks has to be prioritized. Before you start you have to be well prepared.

10 References

[1] Synopsys, http://www.synopsys.com, November 2005

[2] Mentor, http://www.mentor.com, November 2005

[3] Virtio, http://www.virtio.com, November 2005

[4] VaST, http://www.vastsystems.com, November 2005

[5] ARM, http://www.arm.com, November 2005

[6] Jochen Karrer, “Softgun – The Software ARM”, http://softgun.sourceforge.net, November 2005

[7] Forgotten, “Visual Boy Advance”, http://vba.ngemu.com/index.shtml, November 2005

[8] “Nintendo DS Emulators”, http://www.ndsemulator.com, November 2005

[9] Chris Double, “Homebrew Nintendo DS Development”, http://www.double.co.nz/nintendo_ds/index.html, November 2005

[10] Steve Furber, “ARM System-on-chip Architecture” Addison Wesley, ISBN 0-201-67519-6

[11] Alan Clements, “PRINCIPLES OF COMPUTER HARDWARE” Oxford University Press, ISBN: 0198564538

[12] David Seal, “ARM Architectural Reference Manual” Addison-Wesley, second edition, 2000, ISBN 0-201-73719-1

[13] Application Note 25 Exception handling on the ARM (including Thumb-aware processors)

[14] Christerson, Carl, “Master thesis proposition”, Sony Ericsson Mobile Communication AB, 2005

[15] Richard Phelan, “Improving ARM Code Density and Performance”, Thumb-2 Core Technology Whitepaper, June 2003

[16] ARM Limited, “AMBA™ Specification (Rev 2.0)”, http://ww.gaisler.com/doc/amba.pdf (051204)

[17] Bernd Schmidt, ” UAE Amiga Emulator”, http://www.freiburg.linux.de/~uae , November 2005

[18] Per Håkan Sundell, ”Documentation for CCS64 V2.0 BETA”, http://www.computerbrains.com/ccs64.html, November 2005

[19] “SkyEye”, http://www.skyeye.org/index.shtml, November 2005

[20] Electronic design,“Advanced Microprocessor Bus Architecture (AMBA) Bus System”, http://www.elecdesign.com/Articles/Index.cfm?ArticleID=4165&pg=1 , November 2005

[21] Martin Korth, ”GBATEK”, http://neimod.com/dstek, November 2005

[22] ARM Limited, “RealView ARMulator ISS User Guide Version 1.4”, http://www.arm.com/pdfs/DUI0207C_realview_armulator_iss_ug.pdf , November 2005

[23] Hedley Francis, “ARM DSP-Enhanced Extensions”, ARM White Paper, May 2000

[24] David Sharp, “TARMAC”, http://davidsharp.com/tarmac, November 2005

[25] Carl Christerson, “Master thesis proposition”, Sony Ericsson Mobile Communication AB, 2005

[26] “Locality of reference” http://en.wikipedia.org/wiki/Locality_of_reference , November 2005

[27] Marat Fayzullin, “How To Write a Computer Emulator”, http://fms.komkon.org/EMUL8/HOWTO.html, November 2005

[28] Reshadi, Mehrdad et al. “Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation”, DAC 2003, 2003

[29] Jianwen Zhu, Daniel D. Gajski, “A Retargetable, Ultra-fast Instruction Set Simulator”, University of California, Irvine, CA 92717.3425, USA, 1999

[30] Simon Segars, “The ARM9 Family – High Performance Microprocessors for Embedded Applications”, ARM Ltd

[31] “SWARM – Software ARM”, http://www.cl.cam.ac.uk/users/mwd24/phd/swarm.html, November 2005

[32] “ARMphetamine”, http://armphetamine.sourceforge.net, November 2005

11 Appendix A – ARM Architecture versions

Presented below are the different versions of the ARM architecture and what is new in each of them.

Version 1

- First processor developed by Acorn Limited.
- 26 bit addressing, no support for coprocessor interface or multiplication instructions

Version 2

- First commercial chip developed by Acorn Limited.
- 16 registers, 32-bit data bus and a 26-bit address bus.
- Using only 30k transistors made it one of the simplest useful processors at that time.
- Multiply functionality and coprocessor interface was now included.

Version 2a

- On chip cache.
- SWP instruction for atomic load and store.
- Coprocessor 15 was now added as system controller to manage the cache.

Version 3

Developed and marketed by the team spun off from Acorn in 1990, now known as ARM Limited. Version 3 was sold in a number of variants, as a macro cell, stand-alone processor and as an integrated CPU with an on-chip cache, MMU and write buffer.

Version 3 had 32-bit addressing, separate CPSR and SPSR's and added undefined and abort modes to enable coprocessor emulation and virtual memory support in supervisor mode.

Version 4

- The oldest version of the architecture supported today.
- Signed and unsigned half-word, signed byte load and store instructions added.
- System mode is introduced and the instruction space which is not defined is now categorized for future usage.
- The most revolutionizing addition in Version 4 was the Thumb 16-bit instruction set to get higher code density.

Version 5

- ARM and Thumb interaction is refined.
- Count leading zeros, CLZ, is added to the ARM instruction set. Addition of E and J extensions, E for enhanced DSP instructions with saturated arithmetic and 16-bit multiply operation and Jazelle (J) for Java byte code execution.
- Updated Thumb instruction set, Thumb 2 was introduced in this version.

Version 6

- Improved exception handling, better support for multiprocessing environments.
- Media instructions to support SIMD (Single Instruction Multiple Data) software execution.

Version 7

This is the latest ARM architecture version.

- Includes NEON technology extensions to increase DSP and media processing throughput by 400%.
- Improved floating point support.

12 Appendix B – Open source emulators – an overview

ARMulator [22]

Instruction set simulator from ARM that enables the development of ARM targeted software on non-ARM-base systems. Depending on the ARM core to be simulated it is more or less cycle accurate. It allows for benchmarking of the ARM-targeted software, even though the performance is slow compared to real hardware.

SWARM [31]

Initially designed as an ARM module to plug into the SimOS system developed at Stanford University, it is now an independent ARM simulator. It implements the core and a small amount of internal co-processors at a basic level, and provides full support for registers, memory caches, and the external memory hierarchy. Mainly used for modelling the internal data path to enable research of possible modifications.

ARMPhetamine [32]

This simulator is based on the direct compilation technique. Direct Execution: Target machine (the machine that are simulated) binaries can be executed natively on the simulator host processor by encasing the program in an environment that makes it execute as though it were on the simulated system.

tARMac [24]

Uses the same simulation method as the ARMPhetamine the tARMac mainly focuses on the much earlier ARM3. The author has written the emulator with full computer system emulating real world problems such as synchronization with subsystems, interrupts and exceptions all taken into account.

13 Appendix C – Commercial emulators and simulators – an overview

ARM/Axys - MaxSim [5]

Two advantages with MaxSim, the first one is that there are many components (both off-the-shelf and extendable building blocks) available such as peripheral, memory and bus models from ARM, and other vendors. The second is that MaxSim have C++ based component API.
With MaxSim we can do fast modelling, simulation and debugging of complex System-on-Chip (SoC) designs. MaxSim has a cycle-based modelling paradigm which uses SystemC and allows high accuracy. MaxSim have quick debugging and profiling capabilities which is good for testing the project before any hardware is involved. The built-in SystemC component wizard allows users to add new models easily in a simple plug-and-play fashion.

VaST - CoMET and METeor [4]

CoMET is System-level Engineering Environment for concurrent design, development and verification of hardware and software. With CoMET it is easy to do a modification of device parameters such as cache size, bus width and component latencies.
METeor is a Software Development Environment which executes virtual system prototypes created by CoMET. With METeor you can use the virtual system prototype (VSP) created with CoMET to develop, simulate, and debug embedded system software months before hardware prototypes, are available which saves a lot of time.

Virtio - Virtual platform [3]

With Virtual Platforms you can create and integrate embedded software with full debug visibility into the system that is not possible in hardware kits. A virtual platform is a software model of the complete system. It is very fast; you can boot operating systems and run application binaries. With a virtual platform software teams can start software developing, integrating and testing before any hardware is constructed. This saves time and is very cost-effective. Virtual platforms consist of high speed processor, instructions set simulator and functional C/C++ models of the hardware building blocks.

The conclusion of the market study is that due to the high price of these simulators (1.5K€-40k€), we need to design our own simulator. By designing our own we will also gain a higher understanding of how this type of simulator works.