NTNU

Norwegian University of
Science and Technology

Hardware Implementation of a Time Management Unit (TMU)

Stian Juul Søvik

Master of Science in Engineering Cybernetics
Submission date: December 2010
Supervisor: Amund Skavhaug, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem Description

The purpose of this master thesis is to develop a hardware mechanism for execution time control hereby named a Time Management Unit (TMU). The TMU is to have high resolution execution time measurement, running at the same frequency as the CPU clock. The TMU shall allow execution time clocks to be swapped with low overhead. The TMU is to be implemented as a slave module connected to the APB of the Atmel AVR32 UC3.

A functional specification for the TMU is given in the paper "Functional specification for a Time Management Unit" by Gregertsen and Skavhaug. Proposing improvements to this specification and implementing a subset of these, is also a part of this thesis. A software framework consisting of device drivers will be provided. Finally, various functional and performance tests will be created and performed. The resulting product of the master thesis is intended to be used for further research within dependable real-time systems at NTNU.

To achieve these goals, the student is to:

- Familiarize himself with earlier work and relevant theory
- Implement a TMU as specified
- Suggest and preferably implement improvements to the original specification
- Evaluate the final product with regards to correct functional operation and performance
- Develop necessary drivers and example software
- Create documentation for the final product

Assignment given: 23. August 2010
Supervisor: Amund Skavhaug, ITK

Preface

This master thesis is the final part of my master's degree at the department of Engineering Cybernetics, Norwegian University of Science and Technology (NTNU). A large portion of the work was practical, as a digital hardware module was implemented. This work has been performed at Atmel Norway AS.

I am very grateful for the support from my supervisors, hereby Associate Professor Amund Skavhaug who has given advice both about the project as a whole and about the report; PhD Student Kristoffer N. Gregertsen who participated in valuable discussions regarding the specification and implementation of the Time Management Unit; and Dr. Chong-Fatt Law who has been very helpful in answering technical questions about Verilog, giving advice about best practices for digital hardware design, and reviewing the module's code to ensure it meets Atmel's quality standards.

I also appreciate the support from Atmel Corporation, as I have been allowed to use their proprietary source code of the AVR32 UC3 microcontroller to simulate my module, and they have provided tools and resources necessary to complete the project. Also, several of their skilled and kind employees have provided help with setup of tools and given advice.

Finally, I would like to thank my parents, who have spent some of their Christmas vacation proofreading my report.

At the time of completion, a paper based on this thesis is being prepared for submission to the 14th Euromicro Conference On Digital System Design (DSD), alternatively the 23rd Euromicro Conference on Real-Time Systems (ECRTS) if time does not permit preparation before DSD's deadline. (http://www.euromicro.org/Events.php).

Trondheim, December 23, 2010

Stian Juul Søvik

Summary

This thesis describes the implementation of a Time Management Unit (TMU) in hardware as specified by Gregertsen and Skavhaug (1), the specification and implementation of several improvements to the proposed specification, and the creation of a software framework to enable use of the module in a convenient way. A set of thorough automatic functional tests are also described and provided. The performance of the module is assessed and discussed. A user description similar to the AVR32 UC3 datasheets is also created.

The TMU has been implemented as a hardware module on the peripheral bus (APB) on the AVR32 UC3 microcontroller, which makes it easy to develop and test stand-alone, and simple to integrate into future UC3 microcontrollers. Also, as the APB interface of the AMBA standard is an open standard used by several System-on-a-chip (SoC) designs (2), the module can be implemented on other microcontrollers with very low effort.

The final product makes it possible to measure and control the execution time of tasks with high precision and low overhead. It supports atomic swapping of registers in a manner closely related to a context switch.

Gregertsen and Skavhaugs's research in implementing support for the Ada language and run-time environment on the UC3 microcontroller will benefit directly from this project, as the system relies on the hardware support provided by the TMU. Also, as the project can be used in proving that hardware support of execution time monitoring may allow for new ways of ensuring schedulability in real-time systems, it can possibly be a part of a new direction in real-time research.

Contents

List of Figures

The source of each figure is referenced to in the figure text. Figures without references are created by the author of this thesis.

List of Code Examples

List of Tables

Nomenclature

Abbreviations

AHB	Advanced High-performance Bus
AMBA	Advanced Microcontroller Bus Architecture (ARM)
APB	Advanced Peripheral Bus
ARM	Advanced RISC Machines Ltd
ASB	Advanced System Bus
AVR	Atmel's microcontroller architecture and product line
CLK	Clock
CMP	Compare Match
CPU	Central Processing Unit
DUT	Device Under Test
DVE	Discovery Visualization Environment
EDF	Earliest Deadline First
FPGA	Field-Programmable Gate Array
FPS	Fixed Priority Scheduling
HDL	Hardware Description Language
HI	High
HSB	High Speed Bus, Atmel's name for AHB
IDR	Interrupt Disable Register
IER	Interrupt Enable Register
IMR	Interrupt Mask Register
INT	Interrupt
LO	Low
negedge	Negative edge
OVF	Overflow
PB	Peripheral Bus, Atmel's name for APB
PC	Personal Computer
posedge	Positive edge
PREVCMP	Previous Compare Match
PREVOVF	Previous Overflow
RISC	Reduced complexity Instruction Set Computer (as defined by Atmel)
RM	Rate Monotonic
RST	Reset
RTL	Register Transfer Level
SCR	Status Clear Register
SR	Status Register
TMU	Time Management Unit
UC3	Atmel microcontroller series
VCS	Synopsys Verilog Compiler
VHDL	VHSIC HDL
VHSIC	Very High Speed Integrated Circuits
WCET	Worst-Case Execution Time
XML	Extensible Markup Language

Formatting

Code examples are either given inline, such as "`result = doComputation();`", or in separate blocks, as shown below. Comments to the reader that are not a part of the original code are written in italics, and where implementation details have been removed, a short description of what is done is given in parenthesis where relevant.

```
for (i = 0; i < ITERATIONS; i++)
  (Compute new result)
```

Signal, register or variable names are written in bold, such as "**COUNT**". Uppercase variable names used in underlying articles or reports have been retained, although the same case is not necessarily used in the code implementing this project. When a register or variable is referenced to by its general usage, no special formatting is applied, such as "the counter".

Larger quotes are indented and italicized.

Chapters without a separate introduction section give a shorter introduction to the reader which describes the intention of the chapter, or other notes to make the text easier to read and understand. These notes are italicized.

Empty pages are inserted so that each chapter starts on a left page, to make it easier to keep illustrations and their references visible at the same time when reading the printed booklet.

Code Shortening

Most code example blocks have been shortened in several ways. Many identifiers require several prefixes due to code standards, such as `apb_tmu_pwrite`, but in the code examples only `pwrite` is displayed. Some comments from the original code are removed, and explained in the text instead. Also, the implementation uses wires called `apb_write` and `apb_read`, which hides the implementation of APB control signal logic. This makes the code simpler to read.

All blocks writing to registers are synchronous with the clock, which is not necessarily displayed in the code. Other context of many statements is also removed where it is not directly relevant to the example. For instance, if an action happens when a register is written using a bus, only the action itself might be displayed, and the control flow constructs checking that the correct address and control signals are set might be omitted. In these cases, the conditions for the statement to be executed are implied from the text.

Language

Chapter 5 gives a description of the final module in a format that is intended to be similar to a chapter describing a module in the AVR32 UC3 datasheets, that is, it follows the Atmel Document Standards (3). This implies that the text is written in a more standardized and repetitive manner than what would otherwise be expected in a report.

The rest of the report only follows these standards where it is beneficial for consistence without decreasing readability. When there is a conflict between these goals, readability is intended to be given most weight.

In cases where threads, processes, tasks, or other entities that can be executed are referred to without the need of being more specific, "tasks" have been used.

1 Introduction

1.1 Problem Description and Motivation

> *The correctness of a real-time system depends not only on the logical result of the computation, but also on the time at which the results are produced.*

Quote 1 – Definition of a real-time system (4)

Tasks running on hard real-time systems are regarded as incorrect if they do not complete before their deadline, and on hard real-time systems, failure can have catastrophic results (4). Tasks need to be scheduled to ensure that they will be able to complete in correct time, hence the timing capabilities of a real-time system is essential.

In real-time systems, the worst-case execution time (WCET) of a process is a widely used measure in the process of ensuring that a system will be schedulable. However, due to modern processors' performance enhancing techniques such as pipelining, caching and branch prediction, finding WCET may be very hard (4; 5), and it will often be much greater than the average execution time (6). This leads to the dilemma between choosing to have poor CPU utilization to decrease probability of deadline misses, or use optimistic budgets and risk deadlines being missed. (1)

An alternative approach is to base the scheduling on giving each task a budget of at least its average or excepted execution time. Overruns of these budgets can be handled dynamically with an alternative task with a constant or limited short execution time. The alternative task can for instance return a result based on a simpler algorithm, or perform necessary handling of the error to prevent the system from failing. Because of the alternative task's simplicity, its WCET can be computed.

This will let the programmer allow higher utilization of the processor and still ensure schedulability (1; 6). On the other hand, this approach depends on the availability of a precise mechanism for execution time monitoring. To handle overruns, a mechanism for interrupting the processor when the execution time budget is depleted is necessary.

1.2 Previous Work

Bjørn Forsman has previously implemented a Time Management Unit (7) based on a specification given by Håvard Skinnemoen and Amund Skavhaug (8) to improve predictability of real-time systems. It will limit the execution times of tasks as well as the occurrences of interrupts as perceived by the CPU. The module is implemented as a peripheral device on the APB bus (9) of the LEON3 processor. It requires modifying the CPU's internal workings, that is, it interferes with the interrupt lines to stop the signals from propagating when the interrupt line in question has reached its limit of interrupts per time. It contains a dedicated timer for each interrupt line.

A simpler approach is to use a hardware unit with only the capabilities of counting CPU cycles and generating an interrupt when this count reaches a given number. This allows for more flexible scheduling policies as they are implemented in software, as well as improved portability because of the unit's simplicity. Gregertsen and Skavhaug took this approach when *implementing*

the new Ada 2005 timing event and execution time control features on the AVR32 architecture (10). Here, the built-in **COUNT** and **COMPARE** registers of the AVR32 UC3 processor were used (11). *A real-time framework for Ada 2005 and the Ravenscar profile* has also been created (12), which is based on the work in (10).

The hardware implementation used in their works has some drawbacks, that is, relative time has to be used as the counter is only 32-bit which results in computational overhead, and special care is needed to prevent the counter from overflowing. Also, atomic swapping of the **COUNT** and **COMPARE** registers was desired. Hence, there is a need for a dedicated timer unit to support this work (1).

The TMU implemented in this project will be compared to these implementations in section 8.3.

1.3 Main Contributions from This Project

This master's thesis supports the work of Gregertsen and Skavhaug by implementing the proposed Time Management Unit (TMU) in hardware as specified in (1). This work also contributes to the research by making improvements to the specification of the TMU and implementing a subset of these in the hardware unit.

Successful completion of this project has led to a module that extends the AVR32 UC3 microcontroller. Also, a testbench and tests both for the TMU as a stand-alone module, written in Verilog; and for the TMU integrated with UC3, written in C; were developed as a part of this project. In addition to proving the correctness of the implemented module, the tests can facilitate further work with the project. Furthermore, an introduction to Verilog for programmers familiar with C was created, which will benefit students extending the implementation.

To make the module as ready as possible for integrating into a commercial UC3 model, the relevant parts of a datasheet were also created. A software framework consisting of drivers and example code is also provided.

1.4 Scope of This Project

Exploring or choosing target processor is not relevant to this thesis, as the AVR32 UC3 is already chosen in the work this project is based on (1; 10; 12). Several soft-core processors are explored in (7); hereby LEON, OpenRISC and AEMB. Also, as Atmel Corporation requires Verilog to be the hardware description language used for their modules, no research and choice of language is performed.

To implement a digital unit in hardware as is required in this project, knowledge of digital hardware design is mandatory. This includes learning the discipline of designing hardware, as well as learning a hardware description language. Furthermore, one needs to be familiar with real-time related principles such as scheduling theory, timing concepts and real-time requirements. Good knowledge to microcontrollers in general, the microcontroller to host the module in particular, and the buses needed to interface the module, is also important.

The use of various tools such as hardware simulators, waveform viewers, compilers, verification frameworks and proprietary scripts is also a part of the required work, although this will not be a main focus in this report.

To gain the necessary understanding of the application for the module, literature about real-time systems and related concepts, as well as the articles relevant to Gregertsen and Skavhaug's research and earlier related work, need to be read and understood.

The proposed specification is very simple, and suggesting additional improvements, making choices where details are missing, correcting any errors detected, and implementing some of the improvements, are natural parts of this project as well.

To ensure the correct function of the module, tests need to be created and executed, and possible errors must be corrected. Furthermore, tests for assessing performance are desired. Creating software drivers that make the module easy to use is also a part of the project.

1.5 Outline

After this first introductory chapter, the second chapter presents background theory necessary to understand the rest of the report. Both real-time concepts, the AVR32 architecture and digital hardware design will be touched. The third chapter describes the TMU as it was originally proposed, and how this version was implemented. Then, the fourth chapter proposes and discusses several improvements, and gives details about implementation where it is relevant.

Chapter five is a description of the final product, and can also be used as a user guide which resembles the description of a module in an AVR32 UC3 datasheet. Chapter six presents the software framework developed for the hardware module. The seventh chapter treats functional and performance tests of the final product, and the eighth chapter discusses the module as a whole. Finally, the ninth chapter concludes the report and project.

Each chapter contains either an introductory section, or a shorter explanation of the chapter's purpose and other useful information to the reader.

Because this report covers several details that need to be discussed, the discussions are often written at the same place as the detail is presented. The final discussion chapter covers the TMU at a higher level.

2 Theory and Background

As the reader is expected to be familiar with common real-time concepts, this chapter presents a brief introduction of the subject with references to relevant literature. Then, the AVR32 architecture with a main focus on the UC3 family is introduced. Some concepts of special interest to this thesis are emphasized. Then, some background information of digital hardware design is given. The chapter is concluded with an introduction of Verilog to C programmers.

2.1 Real-Time Systems

As mentioned in the introduction, the correctness of real-time systems is determined both by their functional correctness, as well as the time the results are delivered. For instance, the result of a computation must be available early enough to output a new set point in a control system. Real-time computing does not necessarily imply high performance; qualities such as predictability and safety better describe its most important properties.

Real-time systems are divided into three categories; soft, firm and hard, which are defined in (4). Soft real-time systems have deadlines but can tolerate some being missed, possibly leading to degraded quality. For instance, a video communication system relies on data being delivered with low latency for the video to appear without any significant delay, but can allow some frames to be received late or not at all and still perform satisfactory. Firm real-time systems are similar, but have no use of the data being delivered too late.

In hard real-time systems, it is absolutely crucial that the system responds within the deadline. For instance, the airbags in a car need to be released within milliseconds. Missing deadlines in hard real-time systems might cause large negative consequences; for instance financial loss or even risk human lives.

2.1.1 Scheduling

Scheduling for multitasking in general purpose operating systems

In regular computers, such as PCs or servers, fast response time as perceived by the user and fair allocation of system resources are common objectives for the scheduler. By rapidly switching between tasks, they seem to run concurrently, thus enabling multi-tasking.

Linux achieves this by dynamically allocating priorities to tasks according to how much they have been using the CPU earlier. If a task with a higher priority than the currently running task becomes ready to run, the current task is interrupted and the high priority task might be allowed to execute (13). In this way, historical data is used to schedule tasks.

This approach is not satisfactory for real-time systems, however. Because the Linux kernel as described in (13) is non-preemptive, real-time processes might be blocked for several milliseconds while the system is in Kernel Mode.

Real-time scheduling

Scheduling schemes for real-time systems can be divided in two categories; static and dynamic. Both might use priorities to determine which task is allowed to run next, and the main goal is to ensure that all tasks will complete within their deadlines if possible. Static scheduling algorithms do calculations before execution, while dynamic algorithms do the computations at run-time. Several scheduling algorithms are described in (4), including the dynamic earliest deadline first

(EDF) algorithm, and the static fixed-priority scheduling (FPS) with rate monotonic (RM) priority assignment.

A disadvantage of EDF is that if the system gets overloaded, the set of tasks that are allowed to run is unpredictable, and not necessarily the most important tasks. As explained in (14), this is a considerable disadvantage to real-time systems. Also, the algorithm is difficult to implement in hardware, and have issues about representing deadlines in different ranges. Because of these disadvantages, this algorithm is not commonly found in industrial real-time systems.

FPS with RM is more common in real-time systems. As the priorities are fixed, one can guarantee which tasks will miss their deadlines if the system is overloaded. A disadvantage with both this approach and EDF is that the worst case execution time (WCET) is expected to be known (4). This can be hard or intractable to compute; and even if the execution time is measured, it is difficult to know when the worst case has been observed. Also, although the real issue is to ensure that tasks complete within their deadlines, FPS' mapping of deadlines to priorities can be regarded as an indirect means to solve the problem.

Primary and alternative tasks

An interesting alternative to the scheduling schemes mentioned above is presented in (6). Here, instead of having to prepare for the worst case by allocating enough time for all processes to run even if they reach their WCETs, two versions of each task is used, called primary and alternative. These must be specified so that completing either the primary or alternative can be regarded as successful completion of the task.

The primary task will provide the best service, for instance by returning a set point value calculated with high accuracy and precision in control systems, or simply by completing the job the task was intended to do. The alternative task will, however, merely offer a service that is just acceptable. Examples include returning the previous value of the calculated set point, or perform cleanup of the primary task that did not finish.

The primary task is allowed to run for a pre-determined time, for instance close above its average or expected execution time. The alternative task should have a low WCET, which is easier to achieve because of its simplicity. For all tasks in a system to be schedulable, it suffices to guarantee that the sum of all primary tasks' allowed execution time plus the WCETs of all alternative task do not result in the system exceeding a utilization of 100%.

However, this approach depends on support for measuring execution time with high precision, and the ability to interrupt a task as soon as it has reached its execution time limit.

2.2 AVR32 Architecture

AVR32 is a new high-performance 32-bit RISC microprocessor core, designed for cost-sensitive embedded applications, with particular emphasis on low power consumption and high code density. In addition, the instruction set architecture has been tuned to allow for a variety of microarchitectures, enabling the AVR32 to be implemented as low-, mid- or high-performance processors. AVR32 extends the AVR family into the world of 32- and 64-bit applications.

Quote 2 – AVR32UC Introduction (15)

The AVR32 series consists of several processors, which are divided in two product families according to CPU core: AP7 and UC3. The AVR32 AP7 family is optimized for embedded Linux applications, and is not relevant to this project.

AVR32 is a microprocessor architecture with focus on achieving high code density, which lowers memory requirements and contributes to the core's low power characteristics (11). Load and store operations for up to double words (64 bits) are provided. To accommodate for several applications, different micro architectures are defined. AVR32B is implemented in the AP7 family, while UC3 implements AVR32A.

AVR32B is suited for applications where interrupt latency is important, by for instance providing registers to hold the status register and return address for interrupts, exceptions and supervisor calls, and allowing hardware shadowing of registers. AVR32A is targeted at cost-sensitive lower-end applications like smaller microcontrollers, and does not provide these features.

2.2.1 UC3 Core

AVR32UC is the first implementation of the AVR32A architecture. There exist three revisions of this implementation, where UC3 is the most current one. All revisions are backward compatible.

AVR32 UC3 (may be referenced to as "UC3" from now on) is the microcontroller family of interest to this thesis, as previous related work is based on it. UC3 microcontrollers are optimized for highly integrated embedded applications requiring integrated flash memory (16). The UC3 family is focused on high CPU performance relative to power consumption. The UC3 core and the AMBA buses used for internal communication are of special interest for this project, and will be detailed in the following sections.

The UC3 microcontrollers use AMBA buses for internal communication between many of its modules. The devices on the buses are memory mapped, easily accessible for a user. The AVR32 framework includes C header files that provide structs the user can utilize to communicate with the different modules in a convenient manner that enables writing maintainable and portable code.

Models such as the most recent AT32UC3L run at clock frequencies up to 50 MHz (17). The Power Manager (PM) is responsible for generating clock and reset signals for digital logic, and distributes a possibly scaled down signal from a main clock source to the CPU, High Speed Bus, and Peripheral Buses (referenced to as APB in this report). This is illustrated in Figure 1. Clocks may be disabled individually.

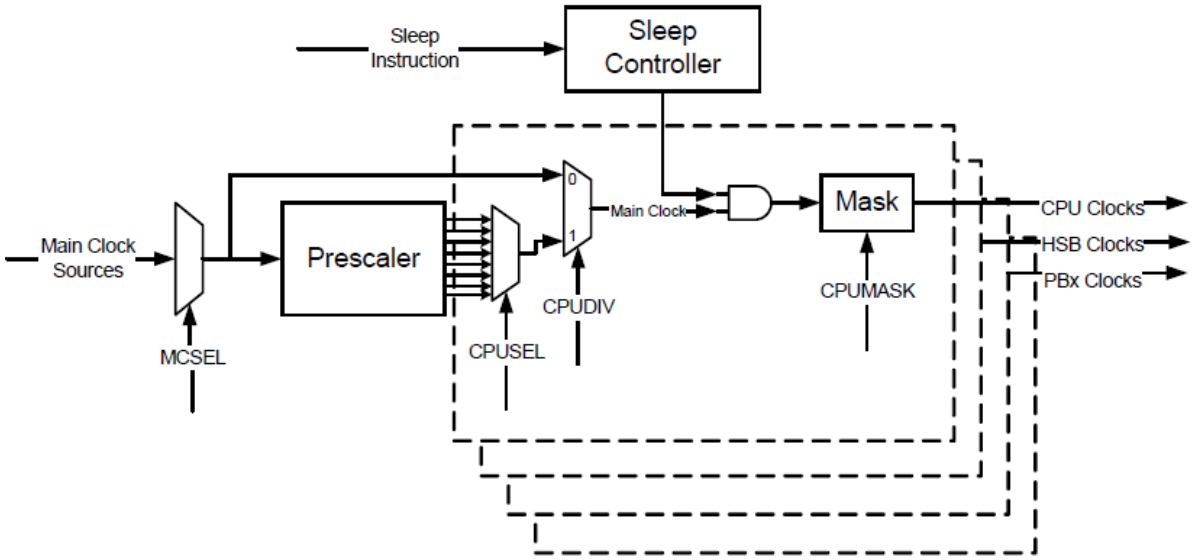

Figure 1 – UC3 clock distribution (17)

Instruction set and issue latency

AVR32 is a RISC architecture, and (11) contains a description of all instructions. Note that Atmel's definition of RISC is "Reduced Complexity Instruction Set Computer" (18), in contrast to the more popular "Reduced Instruction Set Computer". Atmel's view is that it is not the number of instructions that is reduced, but the complexity of the digital circuitry required to decode the instructions. The AVR32 instruction set is fully orthogonal, which means that any instruction can use data of any type via any addressing mode (19). There are a rich number of instructions available, and many allow formats of operands such as post-increment or pre-decrement of pointer registers which reduces code size and speeds up execution.

The pipelined architecture allows one instruction per clock cycle (11), but some instructions, like those that change the program counter, will cause a pipeline flush and hereby require more clock cycles. Also, performance of memory accesses and instruction fetching are affected by the performance of system memories and system bus (20). That is, accesses to external modules might be delayed because of wait states, slow responding modules, restrictions given by the bus protocol, and occupied buses.

An instruction is *issued* when it leaves the instruction decode stage and enters the execute stage. The *issue latency* represents the number of clock cycles required between the issue of the instruction and the issue of the following instruction, that is, how many cycles the instruction "delays" the total system. These definitions and the issue latency of each instruction are given in the AVR32UC3 Technical Reference Manual (20). Factors that can be taken into consideration when computing number of clock cycles include issue latency, memory delay and occupied buses.

2.2.2 AMBA Buses

For internal communication between modules, UC3 uses an ARM Advanced Microcontroller Bus Architecture (AMBA) compatible bus. This is not specified in public available manuals or datasheets as it is first and foremost of interest to those who develop the microcontrollers. The knowledge of the use of AMBA is given from Atmel internally, while the technical specification of the bus (9) is publicly available. AMBA was introduced by ARM Ltd in 1996, and three versions have been created since then (2). UC3 uses the second version.

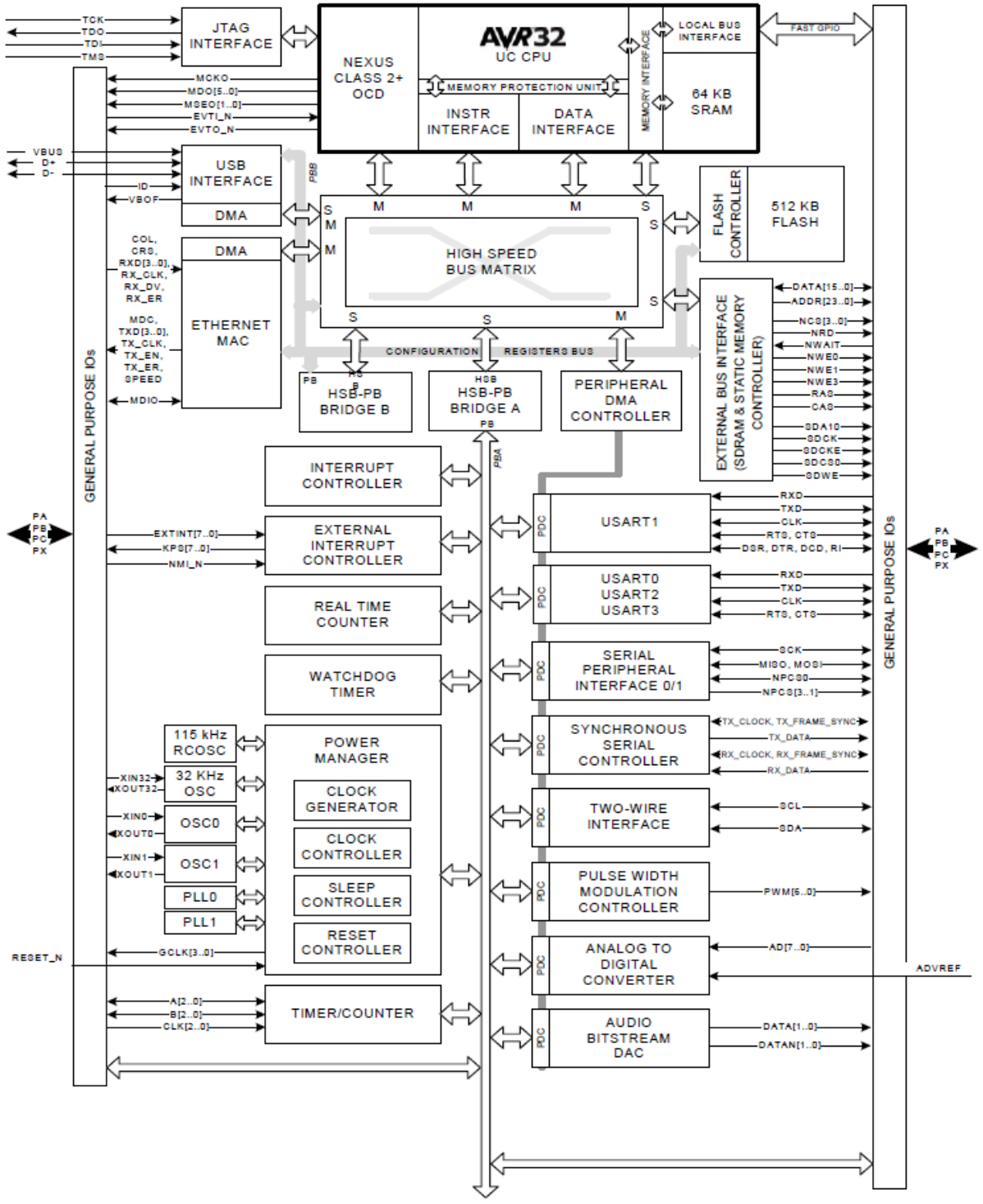

Figure 2 – AT32UC3A block diagram (15)

The AMBA specification (9) describes three buses, namely Advanced High-performance Bus (AHB), Advanced System Bus (ASB) and Advanced Peripheral Bus (APB). This thesis will use the ABP to connect the TMU, and that particular bus will be given the most detailed explanation here.

Figure 2 illustrates the AVR32UC3A microcontroller including its CPU core, peripheral devices and their interconnections. Note that Atmel references to the AHB and APB as HSB and PB, respectively. Figure 3 illustrates how ARM perceives a typical system using the AMBA buses.

The AHB is intended for high-performance, high clock frequency system modules, and is used as the system backbone bus in UC3. The ASB is an alternative system bus, suitable for applications which do not require the high-performance features of the AHB. UC3 uses the APB for its peripherals.

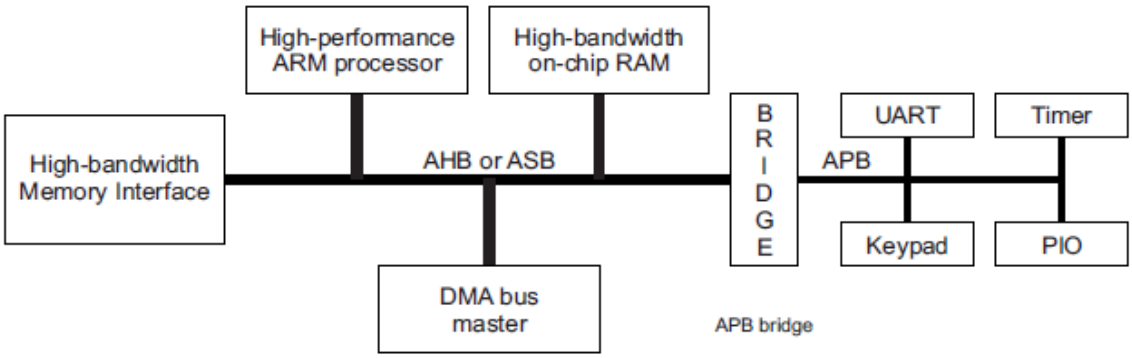

Figure 3 – AMBA buses (9)

AHB implements features such as burst transfers, split transactions and wide bus configurations (64/128 bit), and is thus suited for high-performance, high clock frequency systems. The bus can be bridged together with the APB to allow connection of low-bandwidth peripherals. The APB bridge appears as an AHB slave module, and is responsible for handling the bus handshake and control signal retiming on behalf of the local peripheral bus. It provides latching of all address, data and control signals, and generates slave select signals for the APB peripherals by providing a second level of decoding.

The APB appears as a local secondary bus, and is intended to interface any low bandwidth peripherals that do not require a pipelined bus interface. The bus is optimized for minimal power consumption and reduced interface complexity.

In the following, **PSEL** will refer to the select signal of the APB slave in question, or a general slave where it is relevant. In reality, the APB bridge will assert at most one of several select signals based on the current value on the address bus.

The APB can be in one of three states; **IDLE**, **SETUP** or **ENABLE**. **IDLE** is the default state, and indicates that no transfer is ongoing. When a read or write transfer is initiated, the bus enters the **SETUP** state where a device connected to the APB is selected using the **PSEL** signal, and the **ENABLE** state is then entered in the beginning of the next clock cycle. During this last transition, the address, write, and select signals all remain valid and stable, and the **PENABLE** signal is asserted. The **ENABLE** state also lasts for only one clock cycle. If another transfer is pending, the bus will return to the **SETUP** state; otherwise, it will move to **IDLE** again.

In a write transfer, the address, data, and control signals change after a positive clock edge. This occurs in the **SETUP** state, and is illustrated in the period starting with **T2** in Figure 4. The next clock period starting with **T3** corresponds with the **ENABLE** state, where the **PENABLE** signal is asserted, and the other signals remain valid. In the end of this period, the transfer completes. **PENABLE** will become low after this period, and if there is no transfer immediately after the current one, the **PSEL** signal will also go low. The address and write signal will remain the same to reduce power consumption.

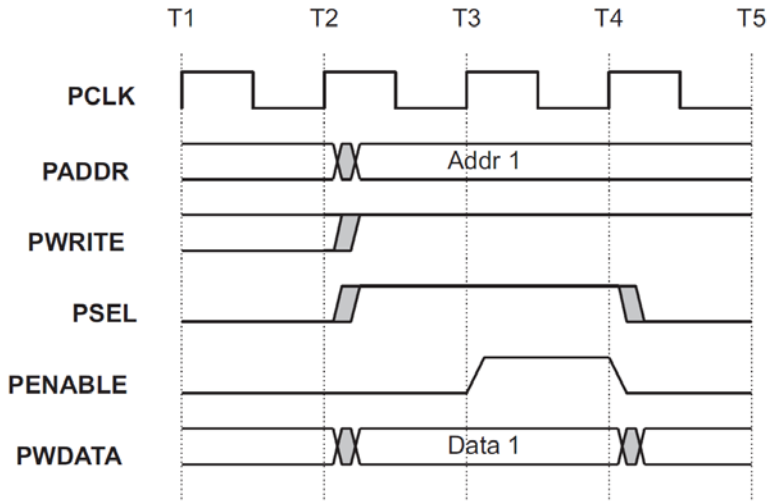

Figure 4 – APB write transfer (9)

A read transfer is specified to have the same timing of the address, write, select and strobe signals as for the write transfer. However, during the **ENABLE** state, the slave will provide the data which is sampled on the rising clock edge at the end of that cycle. An illustration of the read transfer is given in Figure 5.

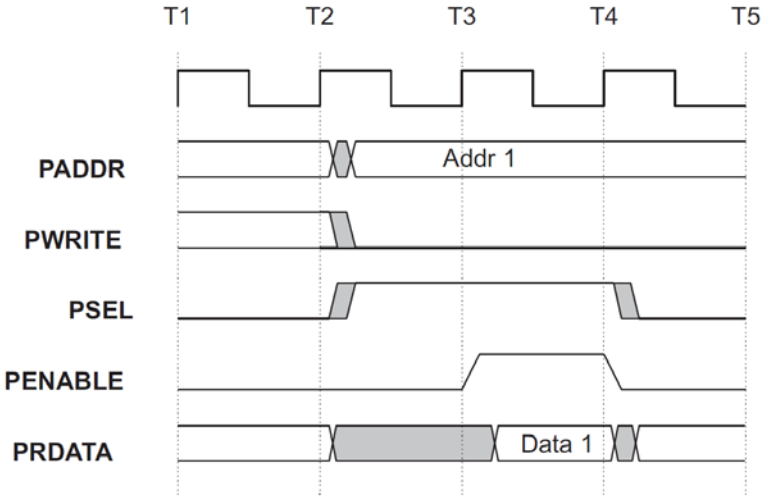

Figure 5 – APB read transfer (9)

An APB slave module, as illustrated in Figure 6, must adhere to the interface given in (9). For a write transfer, data can either be latched on the rising edge of **PCLK** or **PENABLE**, when **PSEL** is high. For a read transfer, data should be driven when **PWRITE** is low and both **PSEL** and **PENABLE** are high. In both cases, the **PADDR** determines which slave device to be activated by

the APB bridge, and it can also decide which internal register or function to be activated in the slave.

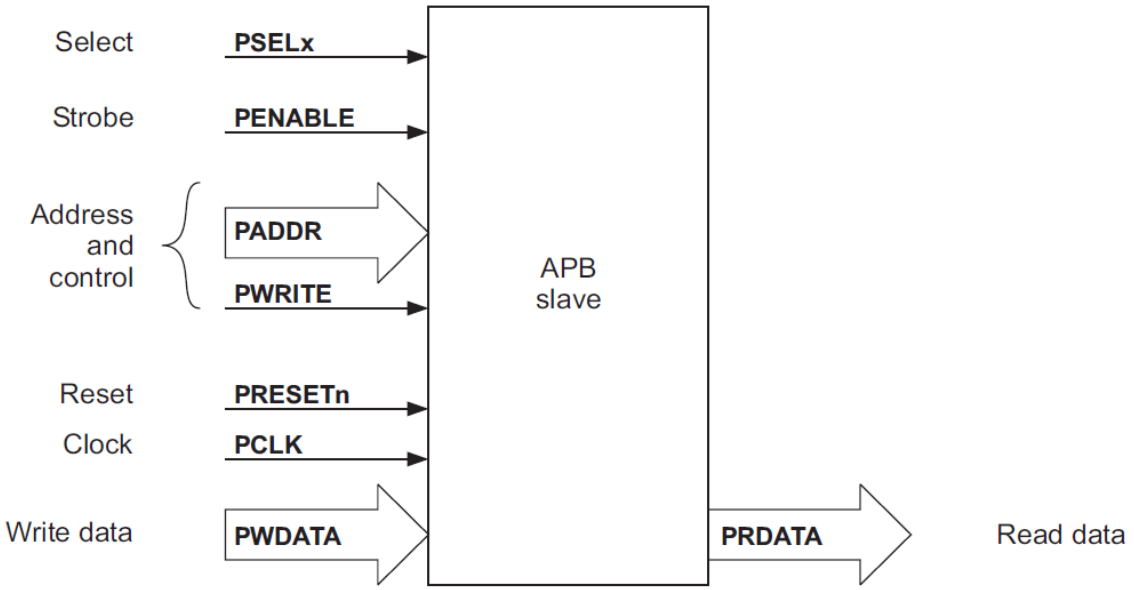

Figure 6 – APB slave module (9)

2.3 Digital Hardware Design

This section first presents an overall workflow, which can apply to a general project. Then, a brief overview of the steps taken to create, simulate, and finally integrate the TMU on the UC3 with Atmel's tools is provided. Also, some of the most important tools used in this project are mentioned.

2.3.1 Workflow for Designing Digital Hardware

When designing digital electronics, following a well-planned workflow will help the designer in having an effective process, and ensure that documentation is created and testing/verification is performed properly.

An example of a workflow with examples of tools is given in the following quote:

1. *Specification: Word processor like Word, Kwriter, AbiWord, Open Office.*
2. *High Level Design: Word processor like Word, Kwriter, AbiWord, for drawing waveform use tools like waveformer or testbencher or Word, Open Office.*
3. *Micro Design/Low level design: Word processor like Word, Kwriter, AbiWord, for drawing waveform use tools like waveformer or testbencher or Word.*
4. *RTL Coding: Vim, Emacs, conTEXT, HDL TurboWriter*
5. *Simulation: Modelsim, VCS, Verilog-XL, Veriwell, Finsim, Icarus.*
6. *Synthesis: Design Compiler, FPGA Compiler, Synplify, Leonardo Spectrum. You can download this from FPGA vendors like Altera and Xilinx for free.*
7. *Place & Route: For FPGA use FPGA' vendors P&R tool. ASIC tools require expensive P&R tools like Apollo. Students can use LASI, Magic.*
8. *Post Si Validation: For ASIC and FPGA, the chip needs to be tested in real environment. Board design, device drivers needs to be in place.*

Quote 3 – design and tool flow (21)

As the specification and high-level design was already given in (1), this project concentrated on low-level design, RTL coding and testing. The author of this thesis was unfamiliar with hardware development, hence it was not easy to estimate how much time was needed to set up a development environment, learn Verilog and implement features. Thus, it was decided that the module should be implemented according to the original specification at first, and then improvements and new features should preferably be introduced at a later stage according to available time.

Sample code for a tutorial module was used as a framework for syntax, tab spacing etc, and it also demonstrated communications with the APB and a testbench. Although this code was not directly usable for the TMU, it was utilized as a starting point when a new language had to be learned in short time.

Automatic tests have been used extensively in this project. There are several benefits of developing thorough test programs, as described in 7.1. Figure 7 illustrates the workflow used in this project.

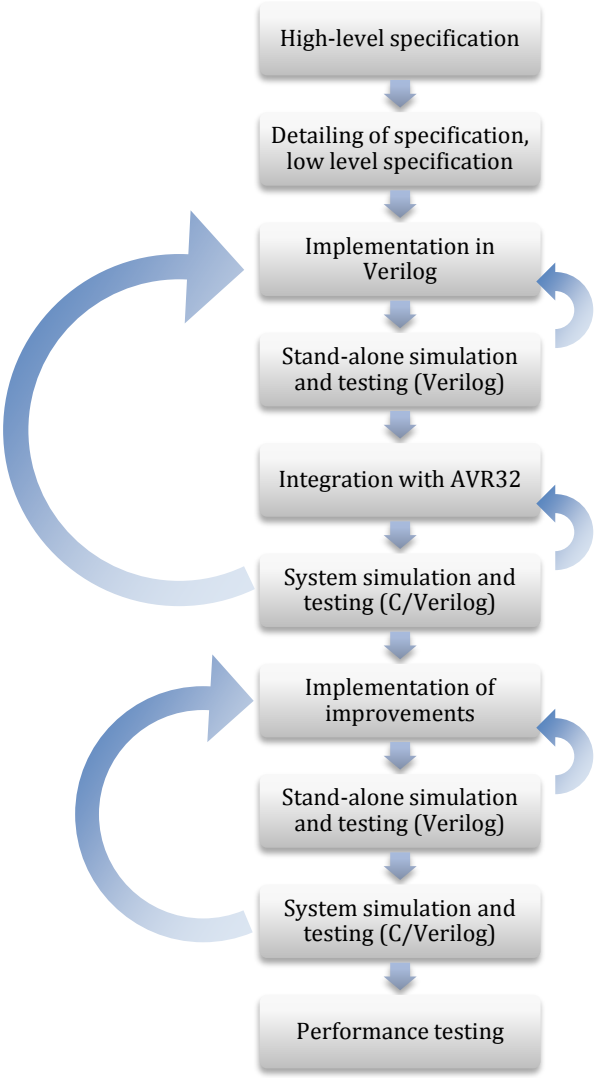

Figure 7 – development process used in this project

2.3.2 Workflow with Atmel's Tools

Substantial configuration and setup is needed to create a hardware module and integrate it with an existing microcontroller, and possibly synthesize it to run on an FPGA. Unfortunately, most of the procedures and tools will differ between projects and hardware configurations. The tools and procedures used in this project are provided by Atmel Corporation, and the information about the procedures, scripts, tutorials and some tools is confidential and cannot be published in this report. Also, one would need assistance from several people to be given necessary access, so cooperation with Atmel is needed in order to continue the work of this project in any case.

However, a brief overview of the tools and procedures used in this project is provided here. Hopefully, this can be used as a starting point for students continuing to work on this project.

Atmel has an internal tutorial for creating a module connected to the APB on UC3. Although the guide does not go into any details, it provides some key information about where some of the necessary tools are located, and hints about where to look for more information. As the tutorial is far from complete, assistance from the experts at Atmel is needed to complete implementation of the module. Also, the tutorial does not provide anything related to describing the hardware using Verilog – only an overview of steps related to workflow.

Setup of tools and environment

The following presents a more concrete list of required actions to perform the steps of the workflow given in 2.3.1 when using Atmel's tools and equipment.

1. User account needs to be set up with correct permissions, SVN access, and internal Atmel modules need to be activated.
2. Check out stand-alone tree
3. Write code for module
4. Compile the module
5. Create testbench and tests in Verilog
6. Run tests and verify that module works properly stand-alone
7. Check out a project containing complete module
8. Extend "modules file" to include your own module
9. Run `avr32-checkout` to get necessary files
10. Use PartTool to create XML file for module
11. Run PartTool for complete microcontroller to configure it and include XML file for module
12. Run generate scripts to generate various header files, Verilog files and so on
13. Modify necessary files describing interconnections in device, instantiation of modules, clock signal distribution etc.
14. Simulate the complete microcontroller with integrated module
15. Create tests in C for testing module integrated with UC3
16. Run tests and verify that the module works properly as an integrated UC3 module

Some of the most used tools are described in the following. This is provided as a starting point for exploring the tools involved in the workflow. For a full description, consult the respective user manuals.

Synopsys Verilog Compiler (VCS)

When the module itself has been implemented using Verilog, it is possible to compile it to an executable that will simulate the hardware by using VCS. In order to get output from the module, input data needs to be provided. A testbench is normally used to generate input data and verify output data. The output data can be displayed on the terminal, and all signals are stored in a database so that they can be displayed in a waveform displayer (described next). The same program can be used to simulate a complete microcontroller with the module integrated.

Discovery Visualization Environment (DVE)

To study all internal signals and values, DVE provides a graphical user interface that can read the database output of the VCS compiler and display waveforms of signals of the user's choice. Various forms of debugging are available, such as tracing of signal drivers and loads.

Spyglass

Spyglass can generate a graphical view of the physical units and interconnections that would be generated from the Verilog files, and also give useful warnings and hints of possible problems in the code. This tool is useful to inspect the result of the code, and for instance discover unnecessary latches resulting from mistakes.

PartTool

PartTool is an internal Atmel tool which can generate and modify XML files containing metadata about modules and complete microcontrollers. For instance, all registers and bit fields of a module can be described using this tool, and various parameters can be set. The XML files output from this tool are then used to generate several support files for the complete microcontroller, see the next section for a more detailed description.

"Generate" scripts

A microcontroller contains a vast amount of registers belonging to various modules and a lot of interconnections between them. To make the process maintainable, much of the Verilog code describing interrupt lines and memory maps are automatically generated from the XML files describing the module. Also, C header files including register definitions are automatically generated. These support tools reduces both development time and the occurrence of errors from manually maintaining such files.

2.4 Verilog for C Programmers

This section is not intended to be a full guide for learning Verilog, but rather an introduction for programmers familiar with C or similar languages, which will likely be the case for students continuing the work of this project. Some barriers and new concepts a C programmer is likely to meet are described here, as the main focus is on what the author of this thesis found challenging when learning to develop digital hardware. (21) provides a tutorial, and for a reference manual which describes the exact syntax and semantics of the language see (22).

Verilog is a hardware description language (HDL) used to model electronic circuits by describing their behavior. The language can be used in many of the development stages of an electronic system. An important part of the design process is implementing the system in Verilog code, which can later be simulated and tested in the verification stage. Test programs and a testbench can also be written in Verilog. By synthesizing the code, the system can eventually be implemented in e.g. an FPGA or ASIC.

A C programmer will find some of Verilog's syntax familiar. The language is case-sensitive, includes a basic preprocessor with e.g. `` `include `` and `` `define `` statements, equivalent control flow keywords (`if/else`, `for`, `while`, `case` etc.) and compatible operator precedence (23). As an example of the differences between the languages, blocks of code are demarked with `begin` … `end` in Verilog instead of `{ … }` as in C.

To support encapsulation and modularization of a system, Verilog provides the possibility to organize the code into a hierarchy of modules. Modules communicate with other modules through ports, which can be declared as inputs, outputs or both, and constitute the module's interface. The internal behavior of a module is defined using internal net or register definitions and concurrent or sequential statements. A module can also instantiate other modules. Verilog supports tasks and functions which are analog to procedures and functions, respectively, in programming languages.

What is written in Verilog is often intended to eventually translate to a set of basic electronic circuits (flip-flops, latches, registers, gates etc) and their interconnections. Thus, one should keep in mind that the synthesizer will try to recognize patterns in the code that corresponds to the various components, and follow conventions and best practices when writing code.

As for the semantics, or the meaning of statements in the code, there are some important cases where HDLs differ from programming languages, which are described in the following sections.

2.4.1 Data Types and Storage Elements

Most data types in Verilog can store four different basic values (22):

- 0: logic zero, or a false condition
- 1: logic one, or a true condition
- x: unknown logic value
- z: high-impedance state

To store useful information or sets of logical values, it is possible to declare registers which store a given number of such values, or arrays of the different data types. A net or variable declaration (see next paragraph) is 1 bit wide per default, but can be extended by specifying a range, which

makes a vector. The vector definition consists of two constant expressions, representing the most and least significant bits.

Two main groups of data types exist: nets and variables. Nets represent physical connections between structural entities, and shall (in most cases) not store a value. The value of a net will be determined by its drivers, which could for instance be continuous assignments. The basic net type is called `wire`. Various other nets exist, providing the possibility of modeling wired logic.

Variables represent data storage elements, which will store their values between assignments. Variable types include `reg` (registers) and `integer`. As the name indicates, a `reg` can be used to model a hardware register, but can also represent combinational logic. An `integer` is a general-purpose variable used for manipulating quantities that are not regarded as hardware registers, but shall be assigned values in the same manner as `reg` (22).

2.4.2 Assignments

There are two basic kinds of assignments in Verilog; continuous and procedural. Both have conceptual differences to assignments in programming languages that one should be aware of. In brief, continuous assignments drive nets and are evaluated and updated whenever an input operand changes value, while procedural assignments update the value of variables according to the control flow elements that surround them (22). Continuous assignments are used to model combinational logic, while procedural assignments also can model sequential logic.

Continuous assignments connect a net data type to an expression, in practice, an output of a combinational circuit. In C, an expression like `x = 3 + y;` would store the value of `y` at the time of evaluation increased by 3 into `x`, but the similar Verilog continuous assignment `assign x = 3 + y`, makes `x` always equal `3 + y`, that is, `x` gets updated whenever `y` changes.

In contrast, a procedural assignment puts a value into a variable, which will store the value until a subsequent assignment to that variable occurs. This behavior is similar to what a C programmer would expect, with non-blocking assignments as an important exception, explained below. Procedural assignments occur within procedures such as `always`, `initial`, `task` or `function`, and will be evaluated or "triggered" when control flow reaches that statement (22).

A procedural assignment can either be blocking (=) or non-blocking (<=). In sequential code blocks, a blocking assignment will be executed before the following statement is executed, just as in C. In the following example, assume that `a` initially contains the value 2.

```
a = a + 1;
b = a;
a = a – 2;
```

Code 1 – blocking assignment example

When the code is executed, `a` will first increase to 3 and `b` will then copy the same value. Finally, `a` will decrease to 1. In the end, `a` contains 1 and `b` contains 3.

In contrast, a code block of non-blocking assignments are evaluated in two steps when simulating. The first step evaluates all the right-hand-sides and the last step stores the results to the left-hand-sides. This means that they can be considered as executed in parallel, and that the

right-hand-sides always use the "previous" value of the variables. This manner of operation resembles the operation of for instance actual flip-flops. The following example demonstrates how two values can swap without using a temporary variable, as one would have needed to accomplish the same in C.

```
a <= b;
b <= a;
```

Code 2 – non-blocking assignment example

After this, `a` and `b` would have swapped values.

2.4.3 Creating Clocked Logic with the Always Construct

One important control construct is the `always` construct. It executes a statement repeatedly, with a possible event that needs to occur before each execution. This construct is often used to create clocked, or synchronous, sequential logic. Assuming the existence of a clock signal, the event used to let the statement execute is the clock signal changing value. One can specify whether the transition is high to low (`negedge`) or low to high (`posedge`). It is also common to include reset logic, as illustrated in the example below.

```
always @(posedge clk or posedge reset) begin

  if (reset)
    count <= 0;

  else begin

    if (psel && penable && pwrite && (paddr[5:0] == COUNT))
      count <= {buffer, wdata} + 1;

    else
      count <= count + 1;

  end

end
```

Code 3 – clocked logic example

This example sets `count` to 0 if the reset signal is active (high). Otherwise, it checks various control signals and a part of an address bus to determine whether `count` is to be set from the concatenation of `buffer` and `wdata` plus 1, or simply increased by 1. This could correspond to a series of clocked flip-flops (register) with asynchronous reset and an input multiplexed between the output of an adder connected to the register output, and an adder connected to a data bus and a buffer.

2.4.4 Timing

There are several ways to specify timing requirements in Verilog, although not all are synthesizable. When describing synchronous logic, a clock signal is often used to ensure that operations happen simultaneously, and to represent time. The simple statement `always @(posedge clk) if (e) q <= d;` models a flip-flop with enable signal, which is

enabled with the `e` signal and lets `q` store the value `d` had in the previous clock cycle. Note that in clocked statements, the right hand side represents the state of the system *just before* the clock edge, while the left hand side determines where the result is to be stored *right after* the clock edge.

One can also wait for an event to happen just once, by using the `@` operator. `@(negedge clock)` halts execution until the next negative clock edge, which can be useful when needing to introduce delays. It is also possible to wait for a statement to be true, using `wait`. `wait(x == 2)` halts execution until `x` equals 2.

Finally, it is possible to delay execution by a number of simulation time units by using # and the number of units, for instance `#1 x = y;` will delay one time unit before the statement is executed. This can for instance be useful when having to ensure that a statement is not executed at the instant the clock signal changes state in a testbench.

2.4.5 Parallel Execution

It is important to remember that many of Verilog's statements execute in parallel, and to avoid race conditions. For instance, specifying several clocked blocks implies that they are all executed every clock cycle, and by using the non-blocking assignment operator, many assignments happen at the same instant.

Race conditions occur if several assignments to the same variable happen at the same moment. By only manipulating a register in one always-block, as suggested in 2.4.8, it is easier to verify that only one of the assignments can happen at once.

2.4.6 Synthesizable and Non-Synthesizable Constructs

Not all constructs are possible to synthesize, that is, to be used to generate hardware. For instance, the `initial` construct runs only at the beginning of a simulation, and can be used to initialize values in a test bench, but cannot be used to define initial values in a real hardware product. To accomplish this, reset signals must be used.

Synthesis tools may ignore non-synthesizable constructs or return an error. Which constructs are synthesizable or not depends on the tool in use.

2.4.7 Modules and Encapsulation

Modules provide encapsulation of entities in the system, and can for instance be used for partitioning a unit containing a vast amount of logic into several smaller, manageable modules. The interface of a module is defined by its ports, which can be of type input, output or both. A module is defined by the `module` keyword and its name, and its ports are listed in a way that resembles parameters in C functions. A module can contain tasks and functions in addition to its logic statements, and can be regarded as a construct similar to object-oriented programming languages' classes.

The following example defines a simple module with one input and one output, and all it does is to invert the input signal. After reset, the output of the module is 0. This module could further be instantiated in another module.

```
module invert (clock, reset, i, o);
  input clock;
  input reset;
  input i;
  output o;

  reg o;

  always @(posedge clock or posedge reset) begin
    if (reset) o <= 0;
    else o <= ~i;
  end

endmodule
```

Code 4 – module example

2.4.8 Best-practices and Guidelines

The following list consists of best-practices that were given in tutorials or by Atmel's personnel while the author was learning Verilog.

- Only manipulate a register in one always-block
- Use non-blocking assignments in clocked logic
- Use blocking assignments in combinational logic
- Do not mix synthesizable code with test-code for simulation
- Follow internal code guidelines, if such exist

3 Originally Proposed Specification of the TMU

3.1 Introduction

The Time Management Unit (TMU) is a hardware unit intended to keep record of a task's total execution time. It can also interrupt a running task when its execution time budget is depleted. It offers the possibility of low overhead execution time monitoring, as it has built-in 64-bit registers and can store a task's total execution time without the need for extra computations in the CPU. It also supports atomic swapping of values between the currently running task and the next task.

3.2 Specification and High-level Design

The paper "Functional specification for a Time Management Unit" (1) *has been used as a foundation for the intended behavior of the TMU. In addition, a specification written by the same author in SystemC was used to elaborate the desired manner of operation. The author of this thesis' interpretation of these documents is given in this section.* Figure 8 *illustrates the intended connection between the CPU core and the TMU.*

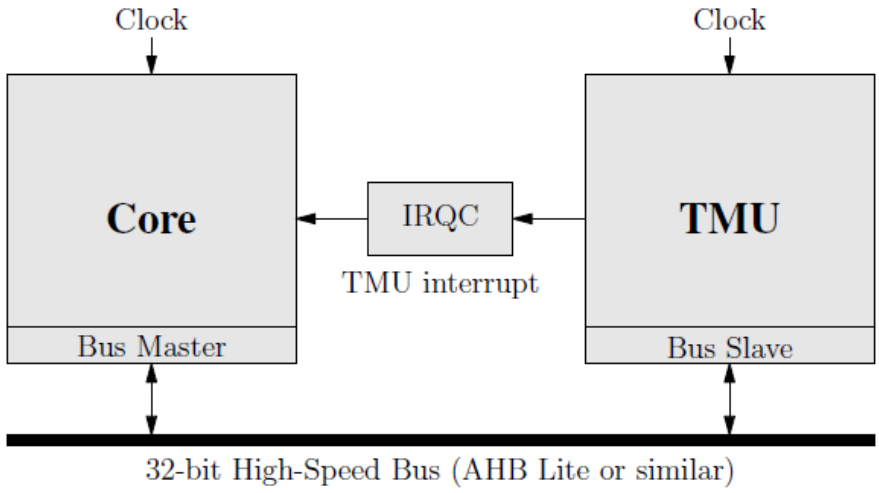

Figure 8 – connection between UC3 core and TMU (1)

Two registers, **COUNT** and **COMPARE**, are essential for the TMU. The **COUNT** register is initially 0, and increases by 1 every clock cycle. It is intended to keep track of the number of clock cycles used by the active task. **COMPARE** contains a user-defined value, and when **COUNT** reaches this level, an interrupt signal is asserted. **COMPARE** is used to represent a budget of clock cycles for the active task, and the TMU will interrupt the processor when a task's budget is depleted to let the run-time system take appropriate action. When a task is interrupted or blocked and another task is resumed, **COUNT** and **COMPARE** are intended to be loaded with the new task's values as a part of the context switch, and the previous values are stored together with the previous task's context.

The registers are 64 bits wide to be able to contain the total running time of a task. The maximum value **COUNT** can take is $2^{64} > 1{,}8 * 10^{19}$, hence a task will be able to run thousands of years at reasonable clock frequencies before the registers overrun. Because of the APB data bus is 32 bits wide in UC3, the values have to be transferred in two parts. Each 64-bit register is

therefore divided into a high and low part, which appears as two 32-bit registers to the CPU core.

Both registers have their corresponding swap registers which are intended to offer atomic operations for exchanging new **COUNT** and **COMPARE** values with previous ones when switching tasks. First, the **SWAP_COMPARE** value is written. After the **SWAP_COUNT** value subsequently has been written, the values in the **COUNT** and **COMPARE** registers are swapped with their respective swap registers. Thus, the previous value of the primary registers are stored in the swap registers in the same clock cycle as they get updated by new values, and are ready to be read back to be stored into the previous task's context. The primary **COUNT** and **COMPARE** values can also be accessed directly if desired, as these registers are available on the TMU's external interface.

Because of the rapid changes to **COUNT**, the lower part is stored in a buffer when the high part is read. This ensures that the value returned is as close as possible to **COUNT**'s value when the read instruction was issued. It also avoids that an invalid value is returned in the case of an overflow of the low part happening between reading the high and low part. **COMPARE** is specified to use the same buffer, although this register can only be modified by the user and thus will not change between reads.

When writing to the high part of **COUNT** or **COMPARE**, the value is stored in the same buffer. Then, when writing to the low part, the written value is stored together with the buffered value into the destination register. This ensures that the entire registers are updated at the same instant, which could otherwise lead to unintended interrupt signals being generated or corruption of the counter value.

As the buffer is shared between **COUNT** and **COMPARE**, and is used for both reading and writing, it is important to complete an operation with one register before accessing other registers in the module.

3.2.1 Context Switch Code Example Explained

An example of how a context switch including the swap operation could be performed was given in (1), however few comments were given. Also, the author of this thesis discovered some bugs in the example, and suggested an optimization as well. The updated code was developed by Gregertsen in cooperation with the author of this thesis, and is explained in this section.

Table 1 – explanation of context switch

Store address of running thread in R9		
`lda.w`	`r8, running_thread`	Load address of running thread's context
`ld.w`	`r9, r8[0]`	Dereference pointer
Save CPU context of running thread		
`sub`	`r9, -(CONTEXT_SIZE + 4)`	Makes space for context
`stm`	`--r9, r0, r1…r7, sp, lr`	Store CPU context
`mfsr`	`r0, SYSREG_SR`	Copy system register
`st.w`	`--r9, r0`	Store system register to context
Store address of first thread in R1		
`lda.w`	`r1, first_thread`	Load address of first thread's context
`ld.w`	`r2, r1[0]`	Dereference pointer
`st.w`	`r8[0], r2`	Set `running_thread` to first thread
Do TMU context switch		
`mov`	`r8, TMU_SWAP_COUNT_HI`	Load address of first swap register
`ld.d`	`r4, r2++`	Load TMU context of first thread
`ld.d`	`r6, r2++`	
`st.d`	`r8++, r4`	Store TMU context to TMU
`st.d`	`r8++, r6`	
`ld.d`	`r6, --r8`	Load TMU context of previous thread
`ld.d`	`r4, --r8`	
`st.d`	`--r9, r6`	Store TMU context to previous thread's context
`st.d`	`--r9, r4`	
Load CPU context of first thread		
`mov`	`r9, r2`	Copy pointer to first thread to R9
`ld.w`	`r0, r9++`	Load system register from context
`mtsr`	`SYSREG_SR, r0`	Restore system register
`sub`	`pc, -2`	Increment program counter
`ldm`	`r9++, r0, r1…r7, sp, pc`	Restore CPU context

Note that the next thread that is scheduled to run is called `first_thread`, because it is first in the ready queue. Also, note the distinction between "context", "CPU context" and "TMU context". The first refers to CPU and TMU context together, which is stored in memory. "CPU context" refers to the normal context of a task, while "TMU context" means the **COUNT** and **COMPARE** registers of the TMU.

3.3 Implementation of the TMU

The TMU is an independent module connected to the microcontroller's bus. Originally, the High Speed Bus (HSB, or AHB) was proposed. As discussed in 8.2.1, the slower Peripheral Bus (PB, or APB) was chosen partly because of its simplicity. By implementing the TMU as an independent module connected to this bus, it is simple to integrate on any processor using the AMBA APB interface.

3.3.1 Interface Registers

4 registers of 64 bits width should be user accessible; **COUNT**, **COMPARE**, **SWAP_COUNT** and **SWAP_COMPARE**. These were implemented as the `reg` type, that is, their values should be retained between assignments in Verilog. Each register has a width of 64 bits, with the MSB and LSB given as named constants.

```
reg [HI_MSB:LO_LSB] count, compare, swap_count, swap_compare;
```

Code 5 – register definitions

These registers are memory mapped and accessed through the APB interface, described in 3.3.6. As the APB data bus is 32 bits wide, two accesses are needed to read or write the 64-bit registers. This has to be done in the correct order, because of the buffer described in 3.3.2 and swap functionality as described in 3.3.4.

3.3.2 Internal Buffer

As described in section 3.2, some of the 64-bit registers need a buffer to store a value between their two 32-bit accesses. This was implemented by using a shared buffer for all registers, and thus it restricts the user from accessing the high part of a register and then accessing another register before completing the first access by reading from or writing to the low part of that register.

When writing to the high part of one of the registers that needs buffering, the value is actually written to the internal buffer. On a subsequent write to the low part of the register, the concatenation of the buffer and what was written to the low part is stored in the target register.

Writing of the **COMPARE** register is used as an example below, as the **COUNT** register has additional side-effects (described in following sections). The offsets are parts of a case statement in a clocked code block, with a part of the APB address bus as input. In the code below, data is buffered when writing the high part, then both the buffer and the written value is stored when writing the low part.

```
COMPARE_HI:
  buffer <= wdata;

COMPARE_LO:
  compare <= {buffer, wdata};
```

Code 6 – buffering registers on writes

Reads are carried out in a slightly different manner. When the high part of a buffered register is accessed, its value is immediately put on the bus, and the low part is stored into the buffer simultaneously. This ensures that both the high and low part of the read value are from the same instant. The first example below displays the clocked write to the buffer that occurs when the high part is read:

```
COMPARE_HI:
  buffer <= compare[LO_MSB:LO_LSB];
```

Code 7 – buffering registers on reads

The next example indicates how the high part is put on the bus when reading. Note that this is a combinational statement (this is not obvious as the context of the statement is not shown here). The buffering in Code 7 and the output of data to the bus in Code 8 are performed at the same time.

```
COMPARE_HI:
  prdata = compare[HI_MSB:HI_LSB];
```

Code 8 – putting data on bus

When the low part then is accessed, it is actually the contents of the buffer that is put on the bus:

```
COMPARE_LO:
  prdata = buffer;
```

Code 9 – putting buffered data on the bus

3.3.3 Increasing COUNT Value

In most clock cycles, the **COUNT** register is simply increased by one by using the clocked statement `count <= count + 1;`. However, if the user is writing to the low part of the counter at that instant (that is, the APB is in its **ENABLE** state and the low part of the **COUNT** register is addressed), **COUNT** is instead loaded with a concatenation of the buffer and data on the data bus. The same block of code handles both normal increasing of the counter and writing to it from the APB. The reason of handling these different actions in the same block is to make it easy to verify that race conditions cannot occur, as detailed in section 2.4.5 and 2.4.8. Readability can suffer when scattering code related to the same feature, in this case, the APB. However, in these cases a comment is written explaining that the read or write is handled in a separate block.

The counter block is implemented as follows (the reset logic is removed and signal names are abbreviated for brevity and readability):

```
always @(posedge clk or negedge rst) begin
  (Reset logic)

  if (psel && penable && pwrite && (paddr[5:0] == COUNT_LO))
    count <= {buffer, pwdata};
  else
    count <= count + 1;
end
```

Code 10 – incrementing counter

Note that in order to avoid the counter from having the same value for a task in two clock cycles, which would effectively be equivalent to "losing" one cycle for a task, the value written back to the counter must be increased in software.

3.3.4 Swapping Values Atomically

The atomic swapping of the registers **COUNT** and **COMPARE** with **SWAP_COUNT** and **SWAP_COMPARE**, respectively, is to be triggered when writing to the low part of **SWAP_COUNT**. The code for implementing the swapping operation is simply as follows:

```
SWAP_COUNT_LO: begin
  compare <= swap_compare;
  swap_compare <= compare;

  count <= {swap_count[HI_MSB:HI_LSB], pwdata};
  swap_count <= count;
end
```

Code 11 – atomic swapping of registers and increasing of count value

As this is a sequential block where statements are executed in parallel, values can easily be swapped simultaneously without using temporary registers. This makes good use of the parallel properties of digital hardware. The **COUNT** register is loaded with the concatenation of the high part of the **SWAP_COUNT** register, and the value currently on the data bus.

3.3.5 Interrupt Signal

In each clock tick in the SystemC specification, **COUNT** is compared to **COMPARE**, and the interrupt signal is set if **COUNT** is equal to or greater than **COMPARE**. The interrupt signal goes low as soon as the condition no longer is true.

The interrupt signal was implemented as a simple combinational statement:

```
assign irq = count >= compare;
```

Code 12 – interrupt signal generation

This way, the interrupt signal will always reflect the relation between the two values, and be reset as soon as **COUNT** becomes less than **COMPARE**. This is not necessarily ideal, as the user has no control over how long the interrupt signal stays active. If the condition would become true during a context switch which blocks interrupts, the condition could become false after the values were updated, and thus not be handled. A task could then overrun its budget without the necessary damage control being performed before the next time the task was scheduled to execute.

The statement could have been made combinational, by letting a register drive the interrupt signal, but this would delay the interrupt triggering by one clock cycle. By using a continuous assignment, the signal would be set at the same clock cycle as when the condition became true.

3.3.6 APB Interface

Because of the 32-bit data buses of the APB interface, each 64-bit register needs to be split in two parts; high (**HI** postfix) and low (**LO** postfix); when being accessed externally.

To follow the APB protocol (9) for read transfers, data from the registers had to be available in the same clock cycle as the **PENABLE** signal went high. As the signal is not specified to be available at the beginning of that clock cycle, combinational logic (without using the clock signal) had to be used for implementing APB read transfers. At the same time, reading from some of the registers should have the side-effect of storing the low part in a buffer, and as a consequence, these parts needed to be implemented using clocked logic. Hence, the implementation of APB reading was split in two parts.

The following code block displays the clocked storing into the buffer, and the next block illustrates the more conventional reading of each register using combinational logic.

```
always @(posedge clk or negedge rst) begin

 (Reset logic)

  if (psel && penable) begin

    (Code for APB write - Code 15)

    if (~pwrite) begin // APB Read

      case (paddr[5:0])

        COMPARE_HI:
          buffer <= compare[LO_MSB:LO_LSB];

        COUNT_HI:
          buffer <= count[LO_MSB:LO_LSB];

      endcase

    end

  end

end
```

Code 13 – clocked storing to buffer

```
always @(*) begin

  prdata = 32'b0;

  if (psel && penable && ~pwrite) begin // APB Read

    case (paddr[5:0])

      COMPARE_HI:
        prdata = compare[HI_MSB:HI_LSB];

      COMPARE_LO:
        prdata = buffer;

      COUNT_HI:
        prdata = count[HI_MSB:HI_LSB];

      COUNT_LO:
        prdata = buffer;

      (Similar code for swap registers, without using the buffer)

    endcase

  end

end
```

Code 14 – combinational reading of registers

Note that the default value of the read data bus is set to be 0 for all bits. As the bus is implemented as a wired-OR, setting the data output bus to 0 in effect releases the bus so other modules can use it.

Another interesting aspect is that even though the APB bus goes through several states before the transfer is started, the TMU does not need to use state machines to conform to the specification – it merely reacts when the necessary signals have their correct states. If all other control signals are as required, the address bus is used to choose which action to take upon in the `case` statement.

Write transfers were implemented purely as clocked logic, as all resulted in one or more values being stored.

```
always @(posedge clk or negedge rst) begin

  (Reset logic)

  if (psel && penable) begin

    if (pwrite) begin // ABP Write

      case (paddr[5:0])

        COMPARE_HI:
          buffer <= pwdata;

        COMPARE_LO:
          compare <= {buffer, pwdata};

        COUNT_HI:
          buffer <= pwdata;

        COUNT_LO:
          ; // Handled in separate block

        SWAP_COMPARE_HI:
          swap_compare[HI_MSB:HI_LSB] <= pwdata;

        SWAP_COMPARE_LO:
          swap_compare[LO_MSB:LO_LSB] <= pwdata;

        SWAP_COUNT_HI:
          swap_count[HI_MSB:HI_LSB] <= pwdata;

        SWAP_COUNT_LO:
          (As shown in section 3.3.4)

      endcase

    end

    (Code for APB read - Code 13)

  end

end
```

Code 15 – APB write transfers

Note that the code for writing the **COUNT** register is not implemented here, but handled in its own block. The reason is to avoid race conditions, where the APB write code would try to set the counter to the new value written, and the counter increase code would try to increase the

counter. By keeping all code for writing to the counter register in one place, it is easy to use control flow constructs to ensure that only one write takes place every cycle.

Of course, logic for ensuring that only one of the statements would be active at one time is required in any case, but the way it is implemented here makes it simpler to verify that only one of the assignments would be active at any instant.

4 Improvements to the Original Specification

During the project, new ideas and opinions about potential for improvement appeared, and the specification was improved and extended with new features. Also, where the level of detail of the specification was insufficient, appropriate choices were taken and the specification updated. After the introduction, each proposed improvement is described in its own section. For the improvements that were implemented in this project, details and discussion about the implementation is then given. An explanation of why the improvement was not implemented is given for the others. But first, some arguments and background information is given to explain the choices taken.

4.1 Introduction

The original specification (1) of the hardware TMU was written by the author who is working on *implementing timing event and execution time control features for Ada 2005 on the UC3 architecture* (10), thus the original TMU specification is tailored for these needs. However, the author of this thesis is of the opinion that the TMU should be created for a more general usage, as the module is intended to become a part of the commercially available UC3 series. To increase the probability for this to happen, changes to the original specification were made so that the TMU could be used in a more versatile way. Another important aspect is that if the TMU is to be a part of UC3, its user interface should be in accordance with other existing modules. A user accustomed to other modules of the UC3 should not have to meet any big surprises while using the TMU. Finally, the module should appear transparent to a user who is not using it or even knowing about its existence. That means it should not start counting (which would consume unnecessary power) or generate interrupts (which would disrupt system behavior) if the user has not explicitly activated the module.

Hence, status and interrupt control registers were introduced, and counting and interrupts were disabled as default. These changes make the TMU non-intrusive and unnoticeable to any user not intending to use the module, and can reduce power consumption.

Also, some improvements were introduced to make the module more efficient in several ways. To reduce die area, an unnecessary buffer was proposed to be removed for reading of the **COMPARE** register. Overhead of context switching can be reduced by not reading the **COMPARE** value back. Also, a quick mode was proposed for applications only needing 32-bit registers. A counter overflow interrupt was introduced. Writes to the counter were automatically increased by one, which would save one operation in software for every context switch.

Finally, some new features and additional details to the specification were suggested. The clock source was tied to the APB clock, which can run at the same frequency as the CPU. The ordering of registers was given special attention due to 64-bit operations of the UC3 core. An option for giving a relative compare value was suggested. Interrupt flags that were valid before a swap would be copied and cleared, to make the state of the TMU fully consistent with the active task while still giving the user the ability to inspect the previous task's state.

The module still needs to support the work of implementing the intended features for Gregertsen and Skavhaug's research in the best way possible. Fortunately, by keeping a frequent discussion with the author of the original specification during the work of implementing the module, improvements and extensions to the specification were made possible while ensuring that the module would still support the ongoing research as planned.

4.1.1 UC3-flavor Registers and Default Behavior

The TMU, as originally specified, would start running as soon as it received a clock signal, and generate an interrupt when the counter reached the maximum value of a 64-bit integer because of **COMPARE**'s default value. Although the interrupt in practice only would occur after thousands of years at a few hundred MHz, it is not desirable that a part of the system generates interrupts uncontrolled. Furthermore, there could be situations where the TMU would be used without any interrupts being desired. Also, running the counter as the default action before any user interaction would cause unnecessary power consumption.

Other modules of the UC3 series feature a set of configuration and control registers, which constitutes the user interface. There is no strict system or guidelines for which registers should exist or even how they should operate, due to the variety of functionality and origin of the various modules. For instance, the status register of some modules are automatically cleared when read, while other modules need a status clear register to be written for the same action to be performed. In any case, having a set of configuration registers similar to other comparable modules is an advantage, both because of the need to enable or disable the module and perform various configuration and control, and to make the TMU easy to use for users confident with other UC3 modules.

Having no strict requirements for the configuration and control registers is an advantage in the sense that one can choose the manner of operation as it best fits the purpose. In some circumstances, the choice might seem arbitrary, if there are no obviously best ways of implementing these registers. In these cases, the Timer/Counter module of the UC3L (17) has been employed as a basis. For instance, the TMU will be enabled and disabled using a special control register, similar to the Timer/Counter.

4.1.2 Atomic Operations

As described in the next sections, only a status register and an interrupt mask register will physically be present in addition to the existing registers of the TMU. However, by implementing these registers with read/write access, the module can suffer from problems due to more than one thing happening at the same time.

For example, assume that the software interrupt handler is active because an interrupt condition occurred. The software would read the status register to determine which interrupt condition was satisfied and then want to clear the interrupt flag. Now, if another interrupt happens, the software would still only know about the first interrupt that should be cleared, and assume that the other flags still were inactive. Thus, when writing back the value of the status register, *both* interrupts would be cleared instead of the intended one, and the latest interrupt could inadvertently go unnoticed.

Creating "registers" with special behavior can omit this problem. By using a read-only status register and a corresponding write-only status clear register, the software can clear the wanted bits by writing a '1' to the corresponding bits in the latter, and leave the other bits unchanged by writing a '0' to them. Similarly, special registers for enabling and disabling interrupts can be created.

Of course, these extra registers will not be implemented as physical registers requiring extra area in the module, but merely exist as logic manipulating the existing registers as a reaction to special addresses being accessed as a part of the user interface.

4.2 Overflow Interrupt

As originally specified, the TMU would generate an interrupt as long as the counter value was equal or greater than the compare value. Overflows were not accounted for, so the user would have to take care of this in software if necessary.

An overflow interrupt source was added, which is activated once the counter reaches its maximum value. As the module now got two sources of interrupts, a status register was created to indicate which interrupt was triggered. This is detailed in 4.5. Also, the interrupts can be separately enabled or disabled using configuration registers, as described in 4.6.

The following code is an extract of the clocked code block of the status register. It simply checks if the counter equals -1, which is the same binary value as the maximum integer value given 2's complement arithmetic and that **COUNT** is treated as an unsigned integer. If this condition is met, the corresponding flag in the status register is set to one, which will activate an interrupt in the next clock cycle if the interrupt and TMU are enabled (as explained in 4.4 and 4.6). There is no `else` construct, which implies that the flag keeps it value if the condition is not met.

```
if (count == -1) sr[INT_OVF] <= 1;
```

Code 16 – setting overflow flag

The choice of keeping the value instead of resetting the flag as soon as the condition is not valid anymore was obvious, as the flag would be reset in the next clock cycle when the counter changed value. That would make it impossible for the software to understand why an interrupt was triggered, thus rendering the overflow flag useless.

As the low-overhead operation and internal storage of 64-bit registers were main features of the TMU, taking care of overflows in software would in effect counteract the advantages of the module. For applications requiring overflow handling, hardware support of this feature is important. Also, the discussion of versatility is valid in this situation – it is wise to expect some users to require a common feature like this. This feature is also optional, as described in 4.6, so it should not impose any restrictions to the user if it is not desired.

Since the proposed counter registers are 64 bits wide, one could argue that an overflow never occurs in practice, as the counter would run for thousands of years at clock speeds of a few hundred MHz before wrapping around. However, as the user is able to specify the counter value by writing to its register, an overflow could easily occur if the counter value is set sufficiently high, hence a hardware mechanism to handle overflows is still desirable.

4.3 Changes to Compare Match Interrupt

As a new source of interrupts (overflow) was introduced, a mechanism for determining the source was needed. Hence, a status flag for compare match was added, as described in 4.5. Also, support for enabling or disabling this interrupt source was added, detailed in 4.6.

The main code for controlling this interrupt flag is shown below:

```
if (count >= compare) sr[INT_CMP] <= 1;
```

Code 17 – setting compare match flag

There are some obvious similarities to the code for the overflow flag (Code 16). However, note that while the condition for the overflow flag occurs very seldom, and disappears in the next clock cycle, the condition for the compare match will be valid until the counter or compare value is changed. This will have impact on clearing the interrupt flag, as it might be set again in the next clock cycle if the mentioned values are not changed first.

Also, another change to the original specification is that the interrupt flag remains set until it is manually cleared by the user. This makes sense because of the consistency with the control of the overflow flag, and the other reasons mentioned in 4.1.1.

4.4 Enable Flag and Halting of the TMU

The original TMU was specified to start counting as soon as it received a clock signal. Increasing the counter's value, doing comparisons to the compare register and performing other logic each clock period means several transistors switching state, which consumes power. The transistors' switching of state is often the most dominant term of power dissipation in for example CMOS circuits (24).

The enable flag was introduced to cope with this challenge. The flag is required to be set for the following actions to happen:

- Counter increase on every clock edge
- Compare match (**CMP**) flag to be set
- Counter overflow (**OVF**) flag to be set
- Interrupt signal to be asserted

All the mentioned features can be disabled simply by disabling the module, which is done by clearing the enable flag. Also, as the enable flag is zero on reset, the TMU is guaranteed not to start counting or generate any interrupts before the user explicitly enables the module.

Some features are still available when the TMU is disabled:

- Reset
- Reading from and writing to all registers in the user interface
- Performing swap operation

That is, functionality that needs to be specifically initiated by the user is always available, but the TMU will not perform any independent actions or change internal state while it is disabled.

The various features that are to be disabled when the enable flag is not set simply check the value of the flag in the status register (**SR**): `if (sr[SR_EN])`, before performing their function. The enabling or disabling of the module is done by writing to the write-only control register, which contains one bit for disabling (**DIS**) and one bit for enabling (**EN**). The code for manipulating the enable flag by writing to the control register is as follows:

```
CTRL: begin
  if (pwdata[CTRL_DIS] == 1)
    sr[SR_EN] <= 0;
  else if (pwdata[CTRL_EN] == 1)
    sr[SR_EN] <= 1;
end
```

Code 18 – manipulating the enable flag by writing to the control register

Note that this code is contained within a case that switches on the APB's address, and is only activated when the control signals for APB write are set. Also, the disable flag overrides the enable flag, so if both are written at the same time, the module will be disabled.

This implementation is chosen mostly because of the desired similarity to other modules in the UC3, that is; a read-only status register that contains status bits and interrupt flags, a status clear register for the interrupt flags and a control register for performing actions to the module.

When the module is disabled, all registers are available, and their contents are stored. Hence, it is safe to disable the module and re-enable it subsequently. This introduces the feature of *halting* the module. Situations where halting the counter would be desirable might arise. For instance, when handling interrupts, the software could simply halt the TMU if the execution time of interrupt service routines (ISRs) should not be taken into account. Of course, the TMU's state can be stored before and restored after the ISR, but this would require several transfers over the bus.

Having the possibility to enable and disable the module is also an advantage in the sense that this is a common feature of similar modules of the UC3 (17).

4.5 Status Register and Status Clear Register

The status register both reflects the state of several features (two interrupt flags and the enable flag), and is manipulated by a variety of sources (reset, write to control register, write to status clear register, write to swap register, and interrupt conditions). Instead of describing the manner of operation of all these elements in this section, some are described in their own sections. This section will only give a summary of the status register, and a description of the status clear register.

As the TMU would be able to generate more than one interrupt as described in 4.2 and 4.3, a read-only status register containing a flag for each interrupt source was implemented. In addition, an enable flag was added as described in 4.4. A separate status clear register was also created, because of the reasons given in 4.1.2. Only the interrupt flags can be cleared by this register. The enable flag can be manipulated by using the control register described in 4.4.

The status clear register is implemented as shown in Code 19. This code simply checks for bits set to 1 in the positions corresponding to each interrupt in the data that was written, and sets

the flags in the status register to 0 accordingly. Because an interrupt condition could occur the same clock cycle, and then should have priority, this is checked first. This avoids a possible race condition. Note that the bits written to 0 imply no change.

```
SCR: begin
  if (!(count >= compare) && pwdata[INT_CMP])
    sr[INT_CMP] <= 0;
  if (!(count == -1) && pwdata[INT_OVF])
    sr[INT_OVF] <= 0;
end
```

Code 19 – clearing status register

Other ways of clearing the interrupt status flags could have been chosen. First, the interrupt flags could automatically be cleared when the status register was read. That would save the user from having to write to the status clear register, and thus be a little more effective. An argument for not doing any changes to the register when it is read is that this would effectively imply a *side-effect* of a read. Reading registers is usually expected to not have any side-effects, and this could cause problems when for instance using a debugger that automatically reads registers in a module.

Second, the status flags could also have been cleared automatically when the condition was no longer satisfied. However, that would be inconvenient for the overflow flag, as it would be cleared right after it was set, as mentioned in 4.2.

Finally, writing to the status register could be allowed, but this solution has the inherent problem with the possibility of race conditions, as described in 4.1.2. Therefore, having a separate status clear register turns out to be the best solution for clearing interrupt flags in the TMU.

4.6 Interrupts and Configuration Registers

The specified initial value for the compare register was -1, which translates to a string of binary 1's (assuming 2's complement arithmetic) and thus is the highest number possible for an unsigned integer. In effect, this would under normal circumstances disable interrupts from the module because of the extremely high value a 64-bit integer can take upon.

However, as this module should be implemented without affecting the microprocessors functionality for normal users; that is, the UC3 should behave equivalently as without a TMU as long as it is not activated by the user's intention; the opinion of this thesis' author is that interrupts generated by the TMU should be completely disabled as default. The user should be in charge of whether interrupts are generated when the counter becomes equal or greater than the compare value.

Also, as the overflow interrupt was not a part of the original specification, the user should have the opportunity to choose if this event would generate an interrupt. Hence, an interrupt mask register (**IMR**) was created, together with an interrupt enable (**IER**) and disable (**IDR**) register. For an interrupt to be generated when one of the bits in the status registers is set, the corresponding bit in **IMR** needs to be set. Also, the TMU must be enabled. Writing a 1 to a position in **IER** enables the corresponding bit in the interrupt mask, and writing a 0 has no

effect. Likewise, writing a 1 to a bit position in **IDR** disables that interrupt, and a 0 results in no change.

IMR is read-only, and is manipulated only through the use of the **IER** and **IDR** registers. This allows enabling and disabling of one interrupt without taking the other interrupts into consideration. Of course, in some situations it would be more desirable to be able to write the entire **IMR** register directly, so this is a trade-off. The solution was chosen because of its similarity to the interrupt configuration registers of the Timer/Counter unit in UC3L (17).

The interrupt signal is merely the output of combinational logic connected to the various registers:

```
assign irq = sr[SR_EN] && (sr & imr & (1 << INT_CMP | 1 << INT_OVF));
```

Code 20 – combinational logic related to interrupt signal

To break it down; `sr[SR_EN]` demands that the TMU is enabled, by checking the enable flag in the status register. `sr & imr` evaluates to the bits that are set in both the status and interrupt mask register, and the binary AND-ing with `(1 << INT_CMP | 1 << INT_OVF)` makes the condition only care about the two interrupt bits (that is, the enable bit or other bits in the status register won't generate interrupts).

In summary, three conditions are required to be fulfilled for an interrupt to be triggered:

1. At least one interrupt flag in the status register must be set (counter overflow or compare match)
2. The corresponding interrupt must be enabled in the interrupt mask register
3. The TMU needs to be enabled

Compared to the original specification, the overhead by requiring users to set up their interrupts is insignificant, as this would only have to be performed during initialization to get the same result as in the original version. The possibility of configuring interrupts gives the user more control over the unit, and is also an important feature if more interrupt sources are to be added later.

Also, note that when the interrupt signal now depends on the output of a register, the interrupt will be triggered in the clock cycle after the condition occurred.

4.7 Clearing and Moving of Previous Interrupt Flags

Performing a context switch which involves swapping of the compare and counter value requires a non-zero amount of clock cycles, which means that the TMU could generate an interrupt sometime during the switch. Independent on whether interrupts are disabled or enabled while the context switch takes place; it would be difficult to determine if the source of interrupt "belonged" to the previous task, or the task being switched in. Also, interrupts belonging to the previous task will in some cases not be relevant after the context switch.

Hence, it was decided to clear the interrupt flags when performing a context switch. This means that it is guaranteed that the current contents of the status register, and thus the state of the interrupt signal, is consistent with the current active task (which in the TMU, is reflected by the **COUNT** and **COMPARE** registers). If the new set of **COUNT** and **COMPARE** values meet the conditions for triggering one of the interrupts, the flags will be set again in the next clock cycle.

Also, there might be situations where knowing the previous state of the interrupt flags is necessary. To accommodate for this need, the status register is extended with two bits: previous compare match interrupt flag (**PREV_CMP**) and previous overflow interrupt flag (**PREV_OVF**). When a swap is issued by writing to the low part of **COUNT**, the two status bits are cleared simultaneously with their previous value being copied to the mentioned positions, as shown below:

```
if (apb_write && paddr[5:0] == SWAP_COUNT_LO) begin

  sr[INT_CMP] <= 0;
  sr[INT_OVF] <= 0;

  if (count >= compare && sr[SR_EN])
    sr[PREV_CMP] <= 1;
  else
    sr[PREV_CMP] <= sr[INT_CMP];

  if (count == -1 && sr[SR_EN])
    sr[PREV_OVF] <= 1;
  else
    sr[PREV_OVF] <= sr[INT_OVF];

end
```

Code 21 – clearing and moving interrupt flags

Note that the interrupt conditions are also checked here, to ensure they will be detected if it happens the same clock cycle as the register is written. In these cases, instead of moving the previous flag, the flag is set directly.

This moving of previous interrupt flags is a natural extension of the swapping mechanism proposed in the original specification, which ensures that the total state of the TMU reflects the current active task.

4.8 Automatic Increase of Value Written to COUNT

As mentioned in 3.3.3, the counter value needs to be increased when it is written back from a stored TMU context in order to not lose track of any cycles. This was suggested to be performed in software. However, as this is an operation that would have to be executed every context switch, a feature of automatically increasing the value of **COUNT** when it is written is introduced.

The code for writing to the register was modified such that the value written would be increased immediately. This was performed both when writing to **COUNT** directly, and when setting the register by using **SWAP_COUNT**.

```
if (apb_write && (paddr[5:0] == COUNT_LO))
  count <= {buffer, pwdata} + 1;
else if (apb_write && (paddr[5:0] == SWAP_COUNT_LO))
  count <= {swap_count[HI_MSB:HI_LSB], pwdata} + 1;
```

Code 22 – automatic increase of COUNT

In most context switches as intended in (1), the counter value written equals the value previously read for the same task. If the counter were to have the same value in the cycle when it was written back, that procedure would cause one cycle to be "lost", and the total number of clock cycles accounted for would drift. The value could also be increased in software, but that would cause unnecessary overhead. This is discussed in 3.3.3 and further illustrated in 5.6. Hence, automatically incrementing the counter can give a slightly reduced overhead in context switches, and also accommodate for simpler software.

For users utilizing the TMU in other ways than what is expected here, this should be no disadvantage. The difference would just be perceived as the write operation took one clock cycle less, as the counter would seem to have increased earlier. One should only be aware of that setting the **COUNT** value equal to the overflow value would not generate an overflow interrupt, as **COUNT** would immediately be wrapped around to 0.

The following sections describe improvements that were not implemented, but are elaborations to the original specification or suggestions to future improvements.

4.9 Clock Source

The clock source was not specified initially, although (1) stated that it did not need to be the same as used by the core. As the TMU is intended to increase its counter value in coherence with the progression of CPU cycles, and the APB clock corresponds with the CPU clock (possibly scaled down) (17), the APB clock was chosen as the clock source.

If the TMU were to be able to be clocked by an arbitrary source, synchronization issues such as metastability could arise, thus increasing the complexity of the module (25). Also, as the module was able to keep 64-bit values and thus is intended to be able to run for a very long time without overflowing, there was no need to let the user connect the module to a low-frequency clock source with overflow in mind.

A possible issue might arise if the user wants to run the peripheral bus on a lower clock frequency than the CPU, but still needs to count the number of clock cycles. A workaround is to take this into account by using efficient operations such as shifting the values after they are read into the memory, or by using correspondingly lower values for the compare register. This will not give the original resolution back, but is a simple way of coping with for instance variable clock frequencies for the peripheral bus.

If the counter needs to run at the CPU clock at all times, and the peripheral bus to run at a lower clock frequency, a simpler solution might be implemented in hardware because of the two clocks being synchronous. For instance, the counter could increase with a number corresponding to the ratio between the CPU's and the TMU's clock.

4.10 No Reading Back of COMPARE During Context Switch

The **COMPARE** value will not change unless initiated by the user, and should not be necessary to read back after a context switch even though this is suggested in the original specification. By only reading the counter value, a significant amount of clock cycles will be saved every context switch. As explained in 7.4, the read operation takes five clock cycles while a write only needs two in the TMU integrated with UC3. Reducing the required amount of reads to half of the original can reduce the overhead of the swap operation by 36%.

This improvement needs no modification to the hardware implementation of the TMU, but it should be mentioned as it would affect the implementation of the software framework using the module. The optimization is of particular interest if the execution time of interrupt handlers is to be monitored, because of the requirement of very low overhead.

4.11 Memory Ordering

The original specification contains an example of possible addresses for each register, but does not point out the importance of correct ordering.

The TMU makes assumptions on the order of access between the high and low part of the 64-bit registers because of its buffer and swap operation, that is, the high part should be accessed first. If the high and low part are accessed in the wrong order, an old buffer value potentially from another register will be returned; hence half of the value returned will be invalid. Also, the swap

operation is performed when writing to the low part of the swap counter register. Thus, it is important to access the TMU's registers correctly.

> *Some instructions access or use doubleword operands. These operands must be placed in two consecutive register addresses where the first register must be an even register. The even register contains the least significant part and the odd register contains the most significant part. This ordering is reversed in comparison with how data is organized in memory (where the most significant part would receive the lowest address) and is intentional. The programmer is responsible for placing these operands in properly aligned register pairs. This is also specified in the "Operands" section in the detailed description of each instruction. Failure to do so will result in an undefined behaviour.*

Quote 4 – memory ordering (11)

The UC3 instruction set specifies instructions for accessing double words (64-bit), including `st.d` and `ld.d`. The AVR32 Architecture Document (11) explains these instructions this way:

```
st.d:
Syntax:
I.    st.d  Rp++, Rs
II.   st.d  --Rp, Rs
III.  st.d  Rp, Rs
IV.   st.d  Rp[disp], Rs
V.    st.d  Rb[Ri << sa], Rs
Operation:
I.    *(Rp) ← Rs+1:Rs;
      Rp ← Rp + 8;
II.   Rp ← Rp - 8;
      *(Rp) ← Rs+1:Rs;
III.  *(Rp) ← Rs+1:Rs;
IV.   *(Rp + SE(disp16)) ← Rs+1:Rs;
V.    *(Rb + (Ri << sa2)) ← Rs+1:Rs;

ld.d:
Syntax:
I.    ld.d Rd, Rp++
II.   ld.d Rd, --Rp
III.  ld.d Rd, Rp
IV.   ld.d Rd, Rp[disp]
V.    ld.d Rd, Rb[Ri<<sa]
Operation:
I.    Rd+1:Rd ← *(Rp);
      Rp ← Rp + 8;
II.   Rp ← Rp - 8;
      Rd+1:Rd ← *(Rp);
III.  Rd+1:Rd ← *(Rp);
IV.   Rd+1:Rd ← *(Rp + (SE(disp16)));
V.    Rd+1:Rd ← *(Rb + (Ri << sa2));
```

Code 23 – syntax and operation of `st.d` and `ld.d`

Note that it is not specified in which order transfers are performed, such as in the third addressing mode for both instructions – taking the load instruction as an example, it is merely

stated that the value `Rp` points to will be transferred to the concatenation of `Rd+1` and `Rd`, not which part of the source register will be read first.

By studying the waveforms of a simulated CPU performing these instructions, it can be determined that the accesses go through as follows, where the first instruction listed is performed first:

```
st.d Rp, Rs:
*(Rp)   ← Rs+1;
*(Rp+1) ← Rs;

ld.d Rd, Rp:
Rd      ← *(Rp+1);
Rd+1    ← *(Rp);
```

Code 24 – order of access when using `st.d` and `ld.d` (3rd addressing mode)

This imposes the following restriction on the module's memory map:

The highest (most significant) part of each double word (64-bit) register needs to be put in the lowest address to be accessed first.

To make the module's interface intuitive, and support usage of the double word instructions, the memory layout was accommodated to this restriction. Hence, the high part of each 64-bit register was placed in a lower address than the low part. This is also important to consider when designing software utilizing the TMU.

4.12 Remove Buffering for Reading the COMPARE Register

As specified, the **COUNT** register has a buffer to "capture" the low part of the counter value when the high part is read. This is due to the register's nature, that is, it changes every clock cycle, and a buffer will ensure that the value returned is the value that was current when the read instruction was issued. The same buffer is used for writing to **COUNT**, which is important because writing to one half of the register can cause problems when its content is simultaneously compared to the **COMPARE** register, and also if the low part overflows right after the high part is written.

The specification also states that **COMPARE** will use this buffer. For writing, this is important, because otherwise a compare match interrupt could be generated because of the inconsistency between the high part and low part before the latter being written.

For reads of the **COMPARE** register, the situation is different. As its value only may be changed by the user, there is no need to capture the low part of this register when reading the high part.

The only reason to keep buffering reads of **COMPARE** is to obtain symmetry of the registers, in the sense that **COUNT** and **COMPARE** behave similarly, and that **COMPARE** behaves the same way for reads and writes. This might make the TMU easier to document and understand.

Advantages of not buffering **COMPARE** when reading includes the possibility of reduced area, mostly because of less wires being needed between the physical **COMPARE** register and the buffer, and the opportunity of freeing the buffer for other uses.

This change to the specification was not implemented in the current version of the TMU, because of the focus of first implementing the TMU as originally specified, and then incrementally implementing improvements in order of importance. That is, it was not considered as an important improvement, but worth mentioning if reducing area becomes important later. On the other hand, if this change is wanted, it is as simple as commenting out the line storing the low part into the buffer when reading the high part, and exchanging the assignment of the buffer to the data bus with an assignment directly from the low part of the register when it is being read.

4.13 32-bit Mode

The module should be able to be used as a 32-bit counter. In the current implementation, there are some obstacles related to the 64-bit registers that must be coped with:

1. By only considering the 32 least significant bits (LSB) of **COUNT**, overflows would not generate any interrupt, but simply be "counted" in the 32 most significant bits (MSB)
2. Using only the low parts of **COUNT** and **COMPARE** will not generate a compare match interrupt if the high part of **COUNT** is less than the high part of **COMPARE,** and would always generate interrupts when the high part of **COUNT** is highest
3. Not writing to the high part of a buffered register would cause the register to store the written value together with the previous value of the buffer
4. Only reading the low part of a buffered register would cause the previous value of the buffer to be returned instead

A separate 32-bit mode could be created, with the following simple changes when active:

- Overflow interrupt when the low part of the buffer reaches its maximum value
- Enforce all high parts of the registers to equal 0 (will both handle obstacle 2 and 3)
- Omit buffer when reading from low parts of registers.

This would allow the registers to be configured using only one instruction each. This improvement was not implemented in the current version of the TMU, as the other improvements were considered more important. However, a mode configuration register is reserved for this or similar purposes.

4.14 Relative COMPARE Value

In the original specification, an absolute **COMPARE** value needs to be computed by the CPU before it is written to the TMU's register in a context switch. Assuming that a task most of the times are granted the same number of clock cycles before it is to be interrupted; this computation will result in overhead that can be avoided.

A virtual register **SWAP_COMPARE_RELATIVE** which will set **COMPARE** to **COUNT** + **SWAP_COMPARE_RELATIVE** can be created, thus making this register specify how many additional cycles the task is granted instead of the absolute time it is to be interrupted.

As the focus of the original specification was to create a module using absolute time values, this improvement was not important to the research; hence it was not implemented in the current version. However, for a more general use, it could be a valuable feature.

5 Description and User Guide of the Final TMU

This chapter gives a description of the TMU in a way that resembles the parts of the UC3 devices' datasheets describing each module. The following sections are based on the sections of the Timer/Counter (TC) module of the UC3L datasheet (17) to ensure resemblance with similar modules in an actual datasheet. Some layout will be closer related to the rest of this report than the datasheets. Atmel's internal Document Standards (3) which give guidelines for writing datasheets are followed as far as possible in this chapter.

The chapter gives a good overview of the module's interface, can serve as support for users of the module, and can be used as a foundation for a part of a datasheet if the module is to be implemented in a real product.

5.1 Features

- One 64-bit counter tailored for execution time monitoring with low software overhead
- Compare register for execution time control
- Easy setup and configuration, simple interface
- Atomic swapping of **COUNT**/**COMPARE** registers with storing of previous interrupt flags
- Interrupt generation by configurable sources
    - Compare match interrupt
    - Counter overflow interrupt
- Timer can be halted or resumed by one memory access
- Clock frequency determined by Peripheral Bus clock
- Supports native 64-bit instructions (`st.d` and `ld.d`)
- Possibility to utilize as simple generic timer with compare match interrupt

5.2 Overview

The Time Management Unit (TMU) is a CPU cycle counter, intended to keep track of individual tasks' execution time in the **COUNT** register. It has the possibility of generating an interrupt when the **COUNT** register reaches a predetermined value set in the **COMPARE** register, and can thereby provide hardware support to a run-time system implementing execution time monitoring and/or control.

By storing the cycle counter value in a 64-bit register, it is possible to run each task for virtually unlimited time without having to handle overflows in software. If overflow handling is desirable, an overflow flag with configurable interrupt exist, so the software does not have to do any computations to determine if an overflow has occurred. This allows for implementing execution time monitoring and/or control with low overhead.

The TMU is connected to the Peripheral Bus A (PBA), and is clocked by the bus's clock source.

5.3 Block Diagram

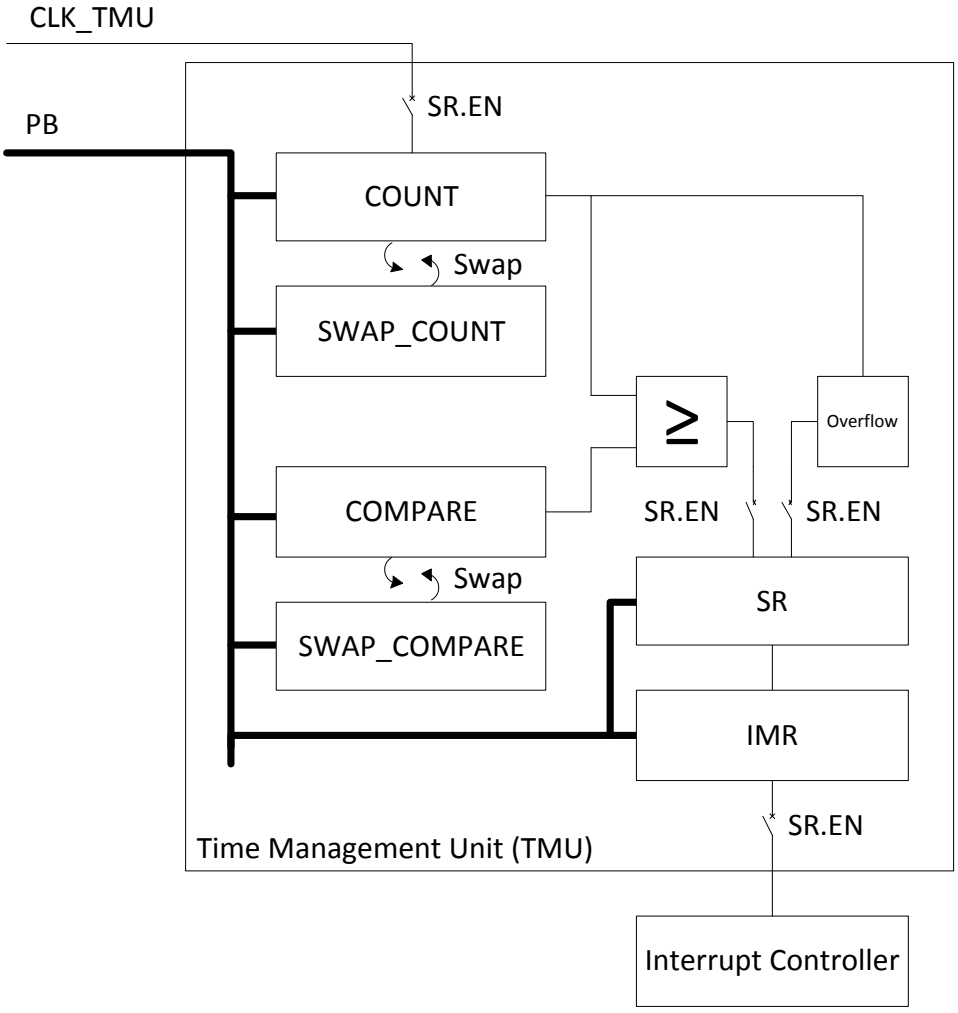

Figure 9 – TMU Block Diagram

5.4 Product Dependencies

In order to use this module, other parts of the system must be configured correctly, as described below.

5.4.1 Power Management

If the CPU enters a sleep mode that disables the Peripheral Bus clock, the TMU will stop counting and resume operation after the system wakes up from the sleep mode.

5.4.2 Clocks

The clock used by the TMU is the Peripheral Bus clock, which is generated by the Power Manager. This clock is enabled at reset, and can be disabled in the Power Manager.

5.4.3 Interrupts

The TMU interrupt line is connected to the interrupt controller. Using the TMU interrupt requires the interrupt controller to be programmed first.

5.5 Functional Description

5.5.1 Controlling the TMU

The TMU can be enabled or disabled by using the Control Register (**CTRL**). Writing a one to the Enable Bit (**EN**) enables the module, and writing a one to the Disable Bit (**DIS**) disables the module. If both are written to one in the same operation, disabling takes priority. The following table summarizes the features that are available in the two states of the TMU.

Table 2 – available features when the TMU is enabled or disabled

Feature	Enabled	Disabled
Counting	✔	⊘
Setting interrupt flags based on conditions	✔	⊘
Generating interrupt signal based on interrupt flags	✔	⊘
Write registers from software	✔	✔
Read registers from software	✔	✔
Atomic swapping	✔	✔
Reset	✔	✔

5.5.2 64-bit Counter

The counter increases by 1 every clock cycle, as long as the TMU is enabled. It is possible to set its value from software, either by accessing the register directly, or by using the swap registers. Either way will increase the value by 1 immediately, that is, the value written on the data bus will be increased by 1 when it is stored in the next clock cycle. This is to facilitate for low overhead context switching, where the previous value can be written back without being increased first.

The **COUNT** register will set the overflow flag (**OVF**) in the Status Register (**SR**) the cycle after it contained the value `0xFFFFFFFFFFFFFFFF` if the TMU is enabled. An interrupt can optionally be generated. The interrupt will also be asserted in the next clock cycle, that is, when the counter value equals 0.

5.5.3 Compare Register

The value of the **COMPARE** register is continuously compared to the current value of the **COUNT** register. The compare match flag (**CMP**) in the Status Register (**SR**) will be set the cycle after **COUNT** becomes equal to **COMPARE**. Also, the flag will always be asserted if **COUNT** is greater than **COMPARE**, even after the flag is cleared by the user. An interrupt signal can optionally be generated when **CMP** is set.

Initially, the Compare Register is set to `0xFFFFFFFFFFFFFFFF`, that is, the highest value possible. This will give the user the enough time to set initial values of **COUNT** and **COMPARE** after initializing the TMU, without concerning about undesired interrupts being triggered. This can allow for making simpler run-time software that does not have to handle the first task as a special case.

5.5.4 Interrupt Configuration and Behavior

For an interrupt signal to be asserted, all of the following need to be satisfied:

1. The TMU needs to be enabled
2. At least one of the interrupt conditions need to be satisfied, indicated by the corresponding bit(s) in the Status Register
3. The corresponding bit(s) need(s) to be set in the Interrupt Mask Register

The Interrupt Mask Register (**IMR**) is a read-only register indicating which conditions will trigger an interrupt. These sources can be configured by using the Interrupt Enable Register (**IER**) and Interrupt Disable Register (**IDR**). When writing to **IER**, each bit set to 1 will enable the corresponding bit in **IMR**. Bits set to 0 will be ignored. Similarly, each 1 written to **IDR** will disable the corresponding bit in **IMR**, while the zeros are ignored.

When an interrupt condition occurs, the corresponding interrupt flag will be set in the Status Register (**SR**) in the beginning of the next clock cycle, independent of the **IMR** register. The interrupt flags will never be cleared automatically, but can be cleared by writing to the Status Clear Register (**SCR**). In the same way as the **IDR** register, writing a 1 to a position in **SCR** clears the corresponding bit, while bits written to 0 cause no change.

5.5.5 Atomic Swapping

The **COUNT** and **COMPARE** registers are closely related to the current running task, and should be changed simultaneously. To avoid having to disable or halt the module while updating the values, which would cause an undetermined number of clock cycles not being accounted for and additional overhead, atomic swapping functionality is provided.

The action is atomic in the sense that all registers are updated at the same instant. Preparation by writing several registers is necessary before the swap, and finalization by reading the previous values can be performed after the swap.

The swap action performs the following actions:

- The counter value of the current running task is swapped with the stored counter value of the new task (**COUNT** and **SWAP_COUNT**)
- The compare value of the current running task is swapped with the compare value of the new task (**COMPARE** and **SWAP_COMPARE**)
- Any interrupt flags in the status register is cleared and copied to a field storing the previous interrupt flags

Note that the copy of the previous flags cannot be cleared directly, and cannot trigger any interrupts. They are provided to make the software able to determine if the interrupt conditions were satisfied by the previous task.

An example of the swap operation, as it is intended to be used, is given in 5.6.

5.5.6 Accessing 64-bit Registers

The peripheral data bus of UC3 is 32 bits wide. Hence, an access to a 64-bit register has to be performed by using two 32-bit accesses. As described in the user interface description (A–2 and 5.7), each 64-bit register is accessed by using two addresses pointing to the high and low part of the registers, respectively. The address layout of the registers is designated to accommodate for

native 64-bit instructions such as `ld.d` and `st.d`. Also, to access several registers in one instruction, `ldm` and `stm` can be used.

The registers can be accessed by performing two 32-bit instructions as well, but the high part of the registers needs to be accessed before the low part because of the buffer described below. In addition, as the buffer is shared between several registers, accesses to other registers between accessing the high and low part of a register is not allowed. Undefined behavior will occur if these guidelines are not followed.

As the counter register's value changes rapidly as long as the module is enabled, direct writing to it could cause unexpected side-effects. For instance, an unwanted compare match or overflow interrupt could be triggered or the register's value could be corrupted given the right circumstances. Hence, the value written to the high part of the **COUNT** register is buffered, and not stored into the real **COUNT** register before the low part is written.

Furthermore, when a read is initiated by accessing the high part of the register, the low part is stored in the same buffer. This is to prevent an invalid value to be returned if the counter overflows between accesses to the high and low part of the register.

The buffer is also used when accessing the **COMPARE** register to prevent unintended compare match interrupts from occurring.

The following examples demonstrate how single assembly instructions can be used to write all swap registers (128 bits total) and read **SWAP_COUNT** (64 bits), and how to access the registers as 64-bit registers in C. The assembly example illustrates the use of the Store Multiple Registers (`stm`) and Load Doubleword (`ld.d`) instructions.

```
mov     r8,   TMU_ADDRESS          // Pointer to TMU
sub     r8,   -SWAP_COMPARE_HI     // Offset to swap registers
stm     r8,   r4, r5, r6, r7       // Write values stored in registers R4-R7
                                   // to SWAP_COMPARE and SWAP_COUNT
ld.d    r4,   r8[8]                // Read SWAP_COUNT back to R4 and R5
```

Code 25 – assembly example of performing swap operation

```
count = *((volatile uint64_t*) &AVR32_TMU.count_hi);
*((volatile uint64_t*) &AVR32_TMU.compare_hi) = compare;
```

Code 26 – C example of reading count value and writing compare value

5.6 Example of Swap Operation

A swap between two tasks can be performed in the following way.

1. Write the compare value of the new task to the **SWAP_COMPARE** register
2. Write the previously stored counter value of the new task to the **SWAP_COUNT** register
   *The swap operation is automatically performed in the beginning of the cycle right after the low part of **SWAP_COUNT** is written.*
3. Read and store the counter value of the previous task (optional)
4. Read and store the compare value of the previous task (optional)

The following is an example of swapping between two tasks. Note that the example is idealized in the sense that each 64-bit read or write operation requires only two clock cycles, which is not necessarily true when the module is integrated in a microprocessor. Even if any operations need more time to complete, no other modifications of the example are needed than extending these operations. Furthermore, to make it possible to illustrate the example, the tasks run for an unnaturally short time, and some cycles are contracted in Figure 10.

First, the "pseudo task" 0 is active. This reflects the program running when the TMU is reset, and is typically the initialization procedures of the system. At cycle 3, an activation of Task 1 is initiated by writing the **SWAP_COMPARE** register with the desired budget. Then, **SWAP_COUNT** is written with the stored counter value of Task 1, which is 0 because this is the first run of the task. Clock cycle 11 is the first cycle after completing writing to **SWAP_COUNT**, and the counter is now valid for Task 1. Note that it begins counting at 1, because 0 was written and automatically increased. Then, the **SWAP_COUNT** register which now contains the previous **COUNT** value belonging to Task 0 is read, and the value 10 is stored into Task 0's context. Note that this was the value of the **COUNT** register immediately before Task 1 was activated.

When Task 1 has been running for 7 clock cycles, Task 2 is to be activated. The same procedure goes for this switch. As it takes 8 cycles to write the swap registers, a cycle count of 15 is stored into Task 1's context. Task 2 is allowed to run for 45 cycles before the scheduler initiates reactivation of Task 1. The cycle count previously stored for Task 1, 15, is written back to the **SWAP_COUNT** register. When Task 1 is activated, the counter's value is 16 because of the automatic increment. This is to ensure that all cycles are accounted for; otherwise, either two cycles would be counted as one (in this example, cycle 25 and 70 would both have the value 15), or the increment would have to be performed in software causing more overhead.

Note that this example does not cover the source of task switch initiation. Normally, it could be the interrupt handler for the compare match interrupt that switches task when the budget of one task is depleted, or the scheduler that switches task of other reasons.

	Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33		71	72	73	74	75	76	77	78	79	80	81	82	83
TMU	COUNT	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1	2	3	4	5	6	7	8		46	47	48	49	50	51	52	53	16	17	18	19	20
	COMPARE																																															
	SWAP_COUNT							W1	W1	W1	W1	R0	R0	R0	R0								W2	W2	W2	W2	R1	R1	R1	R1									W1	W1	W1	W1	R2	R2	R2	R2	
	SWAP_COMPARE			W1	W1	W1	W1												W2	W2	W2	W2														W1	W1	W1	W1									
CPU/mem	(Task 0)															10																																
	Task 1	0																													15																
	Task 2	0																																															53

Figure 10 – illustration of ideal swap example

The illustration displays the number of CPU cycles that have passed by since the TMU was reset in the first row, horizontal movement to the right indicates the time proceeding. The shaded column indicates a contraction of clock cycles to save space. Next, the contents of the **COUNT** register is shown, and is color coded according to which task is currently running, that is, which task the counter is reflecting. The content of the **COMPARE** register is not relevant to this example, and is thus omitted.

The next two rows indicate operations performed to the swap registers. Wx means that the register is written with the value corresponding to task x, and Rx indicates a read of the value to be stored for task x.

Then, three tasks are displayed. Task 0 is the program active when the module is reset, typically initialization procedures. Task 1 and Task 2 are the two tasks relevant to this example. A dark green box indicates that the task is currently active, and tasks that have their TMU-context currently being written or read are light green. After the **COUNT** value for a task has been read into memory, the stored value is displayed in the task's row.

5.7 User Interface

The user interface parts of the datasheets consist of several pages of tables. As this takes much space without adding the corresponding value to the report, only the register summary is presented here. The register descriptions of the TMU are located in the appendices (A–1).

Table 3 – TMU Register Memory Map

Offset	Register	Register Name	Access	Reset
0x00	CTRL	Control Register	Write-only	0x00000000
0x04	MODE	Mode Register (Reserved)	-	0x00000000
0x08	SR	Status Register	Read-only	0x00000000
0x0C	SCR	Status Clear Register	Write-only	0x00000000
0x10	IER	Interrupt Enable Register	Write-only	0x00000000
0x14	IDR	Interrupt Disable Register	Write-only	0x00000000
0x18	IMR	Interrupt Mask Register	Read-only	0x00000000
0x1C	COMPARE_HI	Compare Register (High Part)	Read/Write	0xFFFFFFFF
0x20	COMPARE_LO	Compare Register (Low Part)	Read/Write	0xFFFFFFFF
0x24	COUNT_HI	Count Register(High Part)	Read/Write	0x00000000
0x28	COUNT_LO	Count Register (Low Part)	Read/Write	0x00000000
0x2C	SWAP_COMPARE_HI	Compare Swap Register (High Part)	Read/Write	0x00000000
0x30	SWAP_COMPARE_LO	Compare Swap Register (Low Part)	Read/Write	0x00000000
0x34	SWAP_COUNT_HI	Count Swap Register (High Part)	Read/Write	0x00000000
0x38	SWAP_COUNT_LO	Count Swap Register (Low Part)	Read/Write	0x00000000

6 Software Drivers and Framework

6.1 Introduction

A software framework is provided for convenient usage of all of the module's functionality. As users might have their own opinion of preferred programming language and level of abstraction, several user interface layers are available. Each layer is built using features from the layer below exclusively. The layers are briefly discussed below, and summarized in a table. The C driver will be given more attention in later sections of this chapter. The actual files are provided in appendix A–2.

In the first layer, all registers are directly accessible through their offsets in the module's memory area as listed in 5.7. This allows for fine-grained control over the module, and can be necessary if the module is to be accessed from a language different from C.

For the second layer, a header file giving access to all the module's registers through structs is available. This file was generated based on an XML file describing the registers and their bits by using proprietary Atmel tools. Each register can be accessed directly by its name, such as `AVR32_TMU.compare_hi`, and individual bits can also be accessed via the register written in uppercase, such as `AVR32_TMU.SR.ovf`.

The third layer is the low-level C driver which gives access to all registers, and is further described below. Finally, the fourth layer constitutes the high-level C driver and provides functions combining several register accesses, such as initialization or swapping. The high and low-level parts of the driver are provided in the same file, and the user may use functions from both as it best suits the application.

The drivers are written in a way that can be compiled to only a few instructions. For instance, the function `TMU_getCount` consist of the sole statement `return *((volatile uint64_t*) &AVR32_TMU.count_hi);` that is is translated into the two instructions `mov r8, -33756` and `ld.d r10, r8[0]` which creates a pointer to the register and reads the value contained. Further compiler optimizations could cause the TMU's address to be stored as a constant in a register, and offsets from this pointer could be referenced to directly, which then only requires one assembly instruction to read or write a register.

Table 4 – C user interface layers

Layer	Description	Access method
Registers	Registers for configuring and reading state of the module as described in 5.7 and A–1	Direct pointers to addresses in C or assembly
Header file	Structs with pointers to registers and constant definitions for addresses and offsets	References to the module's registers in statements in C
Low-level driver	C functions providing setters and getters for, or control over, all registers	Function calls can be used directly in expressions
High-level driver	C functions for performing configuration and useful actions	Function calls which can be used when needing to perform common actions

6.2 Low-level Part of C Driver

The low-level driver provides "setter" functions for registers that can be written and "getter" functions for registers that can be read. Write-only registers performing actions are named after their function, which closely resembles the register name.

In the following tables, the prefix `TMU_` is removed from all function names to save space. Return types and parameters are integers if not otherwise is explicitly stated, and their size in bits is given in parentheses.

Table 5 – low-level C driver interface

Name	Return type	Parameters	Description
getStatus	(32)	-	Returns the contents of the status register
getInterruptMask	(32)	-	Returns the interrupt mask
getCount	(64)	-	Returns the counter value
getCompare	(64)	-	Returns the compare value
getSwapCount	(64)	-	Returns the swap counter value
getSwapCompare	(64)	-	Returns the swap compare value
setCount	-	count (64)	Sets the counter value to `count`
setCompare	-	compare (64)	Sets the compare value to `compare`
setSwapCount	-	swapCount (64)	Sets the counter swap value to `swapCount`
setSwapCompare	-	swapCompare (64)	Sets the compare swap value to `swapCompare`
control	-	mask (32)	Writes `mask` to control register
statusClear	-	mask (32)	Writes `mask` to status clear register
interruptEnable	-	mask (32)	Writes `mask` to the interrupt enable register
interruptDisable	-	mask (32)	Writes `mask` to the interrupt disable register

Note that no checking of the validity of parameters is performed. This is to reduce overhead. However, writing to bits that are not defined in e.g. the status clear register results in no action.

6.3 High-level Part of C Driver

The high-level part of the C driver consist of functions for performing common tasks, and this part of the driver should be sufficient to use the TMU as intended. Of special interest is the `TMU_swap` function, which performs the complete swap in one function call.

Table 6 – high-level C driver interface

Name	Return type	Parameters	Description
clearStatus	-	bool ovf bool cmp	Clears the interrupt flags in the status register corresponding to the parameters with true Boolean value
enable	-	-	Enables the module
disable	-	-	Disables the module
enableInterrupts	-	bool ovf bool cmp	Enables the interrupts corresponding to the parameters with true Boolean value
disableInterrupts	-	bool ovf bool cmp	Disables the interrupts corresponding to the parameters with true Boolean value
init	-	bool ovf bool cmp	Enables the module and enables the interrupts corresponding to the parameters with true Boolean value
swap	(64)	count (64) compare (64)	Performs a swap operation with `count` and `compare` as new values, return previous counter value
checkOverflow	bool	-	If overflow flag is set in status register, flag is cleared and true is returned. Otherwise, false is returned.
checkCompareMatch	bool	-	If compare match flag is set in status register, flag is cleared and true is returned. Otherwise, false is returned.

7 Testing and Evaluation

7.1 Introduction

Well-planned and thorough functional testing is important and beneficial for several reasons. First, the tests give a clear illustration of how the module is intended to operate, and can thus be used as a fundament for discussing manner of operation and verifying that the specification is interpreted in a satisfactory way. Second, creating tests forces the developer to think through how the module is intended to work, and this can lead to discovering new ideas, reveal potential flaws or weak points in the specification, and making a more thorough implementation. Third, automatic tests provide a very useful way of verifying that the module is still working as intended after making changes. This will be very useful if other students are going to further develop this module. Finally, the tests and their results prove that the module works the way the tests specify.

The most detailed tests are performed directly on the TMU as a stand-alone module. These tests exercise various part of the module by writing its registers directly or through its interface ports, and are written in Verilog. These tests will be described in the second section. The third section describes tests that run on a simulated UC3 processor with the TMU integrated. These tests are also functional, but test the module on a higher level.

The fourth section presents measurements of the module's performance and the final section briefly assess the size and cost of the module. All tests, example code and the testbench are provided in appendix A–2.

7.2 Functional Testing of Stand-alone Module

As the tests have been developed in parallel with the TMU itself, they have also been updated during the evolvement of the module. The tests as described here and included with the final product corresponds to the final version of the TMU. If the module is to be developed further, the tests should also be extended or modified to reflect the new changes.

The tests are not intended to cover all possible combinations of inputs and states, but are developed with error situations that are likely to occur and elements that are important to verify in mind.

A testbench with various support functions, common initialization/finalizing of a test run, clock generation, instantiation of the TMU and configuration options was created. Then, each test was developed by using the testbench's functions. Some tests required their own support functions, which were included in the actual test's file. This way all tests could be created in a similar manner, with a minimum of code. This enhances readability and is important when using the tests for discussing desired functionality.

The following functions were implemented in the testbench:

```
reset_dut()
wait_cycles(num_cycles)
apb_write(address, data)
apb_read(address, result)
expect_equals(description, actual, expected)
```

Code 27 – testbench interface

`reset_dut` generates a reset signal to the device under test. `wait_cycles` waits a specified number of cycles before proceeding, synchronized to the clock so that it waits for the given number of rising edges. `apb_write` and `apb_read` simulates AMBA APB communication in compliance with the standard (9), and facilitate reading and writing of data from and to addresses specified by the user. They support back-to-back transfers to ensure that the module can be tested with the highest speed possible.

`expect_equals` compares two values, and displays an error message if they are not equal, or an OK message otherwise, both together with the supplied description and values. This function is used to compare the current value of a register to an expected value calculated or written as a constant in the test program, and is used extensively in the tests. It can be used to verify values stored in internal registers of the TMU, thus it is a very powerful tool. One important aspect is the use of the `!==` (case inequality) operator in the implementation of the function, which demands that unknown values (`x`) or high impedance (`z`) are matched on both sides for the operands to be considered equal. This allows for tests that *expect* an unknown or high impedance value, and at the same time, it ensures that an unknown value (`x`) is not matched with *any* value.

Tests have access to all of the TMU's internal registers; hence values can be read or written directly. This is of particular use when verifying that timing is correct, and also makes it easier to set initial values for registers. In the descriptions of the various tests, this will be referenced to as accessing a register "directly". Note that this method must be used with care when tests depend on assignments being synchronized with the clock, so that race conditions do not occur. Using the testbench's APB functions for writing or reading registers ensures interactions synchronous with the clock.

The testbench also includes a separate clock cycle counter called **cycles**, which will keep track of the number of cycles that have passed by since reset, and not be influenced by writes to **COUNT**. This is useful when verifying for instance that no cycles are lost track of due to context switching.

The total number of errors and tests are displayed at the end of the test. A shell script was also created, which would run all tests and only display any errors and a summary of each test. This allows for rapid verification after making changes to the module, and will aid detecting errors in an early stage.

All tests are presented and discussed in the following sections. The tests are commented in the code to explain their purpose and what they do. When this text states that a register or net is expected to contain a value in one of the tests, it is implied that this is verified and required for the test to be considered as passed. In the test source code, the function $display is used both to print a description to the terminal, and to explain what the following tests do to anyone reading the code. Most tests use constant 32-bit hexadecimal values resembling regular words, like "`BA5EBA11`", or numbers that are easy to recognize, when a unique value is required. This is because such values are easy to recognize when inspecting registers or signal values in a waveform viewer.

A summary of all tests is given in the table below. Some tests require that other parts than what is being tested works correctly; these dependencies are stated in the table. The file name of each test is `test_x_y.v` where `x` is the number of the test and `y` is the name, both given in the table.

Table 7 – stand-alone test summary

	Name	Verified elements	Depends on	Pass
1	`amba`	Correspondence with the AMBA APB protocol, particular focus on timing	**COMPARE** and buffer	✔
2	`after_reset`	• Initial values of registers • Module is inactive in startup • Module can be enabled	-	✔
3	`read_write`	Read and write of all accessible registers	APB interface	✔
4	`swap`	Swap functionality for **COUNT**/**COMPARE** and status flags	APB interface	✔
5	`irq`	Correct interrupt signals and status flags being generated when module is enabled/disabled and interrupt mask is enabled/disabled for each interrupt source	APB interface	✔
6	`irq_count_compare`	Interrupt signal after all combinations of chronological and numerical orders of setting **COUNT** and **COMPARE**	APB interface	✔
7	`lost_cycles`	Simulates swapping of tasks, no cycles are lost	APB interface	✔

As all tests passed, no further discussion about the tests results is given.

7.2.1 Correspondence with the AMBA APB Specification

The first test was created with the AMBA APB specification (9) as a foundation. Correct behavior regarding reading and writing using the APB is an important aspect of the module, as it is to be integrated with UC3 and coexist with other modules on the same bus. Compliance with the timing requirements is also an important aspect of the test.

The APB support functions of the testbench were not used in this particular test, because full control over the various signals was needed to perform the specific tests. Also, it is the TMU that should be tested, not the APB functions of the testbench.

First, a single write to the APB is issued. The specification states that for a write transfer, data can be latched at the following points:

- on rising edge of **PCLK**, when **PSEL** is high
- on rising edge of **PENABLE**, when **PSEL** is high

The test expects data to be latched into the register at the rising clock edge, and thus reads the register directly after a minimal delay after the rising edge of **PCLK**.

Second, a burst write consisting of two write cycles with no **IDLE** state in between is tested. After each cycle, the corresponding register is checked to verify that data is latched in at the correct time.

Then, a single read is tested. One of the internal registers is first set directly to a value. After asserting the control signals and address bus, the data bus output of the module is read to verify that the same value is put on it. Also, as the module is to co-exist with other modules on the bus, it is verified that the data bus is released as soon as the **IDLE** state is entered after the transfer.

Finally, a burst read is started. Similar to the other tests, data on the data bus output from the module is expected to equal a value set directly in one of the internal registers at the correct time. Again, it is verified that the data bus is released immediately after entering the **IDLE** state when the transfers have finished.

7.2.2 Correct State and Behavior after Reset

After a reset, the TMU is intended to have an initial state where the interrupt signal is disabled, and all internal register values equal 0, expect from **COMPARE** which has the highest possible binary value. Without any user interaction, the TMU shall not start counting or perform any other actions.

First, the test resets the module, and verifies all internal registers and the interrupt signal. After five clock cycles, the counter is checked to verify that it has not started counting. The module is then enabled, and after another five clock cycles all registers are verified again. This time, **COUNT** should have increased, and the status register should have the enable flag set.

7.2.3 Reading from and Writing to All Registers Perform as Expected

This test exercises all registers of the TMU, except **SWAP_COUNT**. This register has another side-effect – atomic swapping – and its behavior is verified in a separate test.

The 64-bit registers are to be read or written in two stages because of the 32-bit data bus, and some of them use an internal buffer. Correct operation of each register is tested. Also, configuration, status and control registers are exercised. The testbench's internal APB functions are used to read from and write to the registers.

This test first reads the high and low part of the **COUNT** register and expects its initial value 0. Then, writes to the high and low parts are performed to the **COMPARE**, **COUNT** and **SWAP_COMPARE** registers, followed by reads in the same order. Register values are verified after each read/write. The **COUNT** is automatically increased by 1 when it is written, and this must be taken into account when computing the expected value of the counter.

The write-only interrupt configuration registers are expected to return 0 if they are read, which is verified next. Then, the read-only status and interrupt mask registers are read.

The two interrupt conditions of the interrupt mask registers are enabled and disabled in all possible combinations using the interrupt enable or disable registers, and the interrupt mask register is read and verified after each operation.

The module is then enabled and disabled using the control register, and the enable flag in the status register is verified to correspond with the TMU's state. Finally, bits of the status register are set directly, and the status clear register is then tested. This is performed on each bit individually and both bits at once.

7.2.4 Registers Swap as Expected when Writing to SWAP_COUNT

This test exercises the special event that happens when writing to the low part of **SWAP_COUNT**, that is, atomic swapping of **COMPARE** and **COUNT** with their respective swap registers.

First, both **COUNT** and **COMPARE** are directly set to different initial values, and the overflow flag in the status register is set. Then, **SWAP_COMPARE** and **SWAP_COUNT** are written using the APB write functions of the testbench. After writing the low part of **SWAP_COUNT**; **COUNT** and **COMPARE** are expected to immediately have taken the values that were written to the swap registers. Then, the swap registers are read back using the APB read function, and verified against the initial values with the natural increase of **COUNT** taken into account. Also, the status register is read and expected to have only the **PREV_OVF** flag set, because the overflow flag should be moved due to the swap operation.

Next, the moving of interrupt flags are exercised further, where all combinations of the two possible interrupt flags are set, and verified to be moved into their previous copy positions.

Note that this test is performed while the module is disabled, hence no counting occurs. Swapping with the module enabled is thoroughly exercised in the test verifying that no cycles are lost track of due to swap, described in 7.2.7.

7.2.5 Interrupt Perform as Expected

This test both creates situations where interrupts are not expected to happen, and vice versa. As stated in 5.5.4, the TMU will only generate interrupts when all three following conditions are satisfied: the TMU is enabled, at least one of the interrupt conditions is satisfied, and the corresponding flags in the interrupt mask are set. Hence, it is important to verify both that interrupts happen when they are intended to, and that they are not generated when all conditions are not satisfied.

First, **COUNT** and **COMPARE** are set equal, which would normally generate an interrupt. As the module has not yet been enabled, it is expected that neither any interrupt flags or the interrupt signal are active. The same procedure is then performed for the overflow condition, by setting **COUNT** so that it overflows.

Then, the module is reset and enabled. The same tests are performed, but the status flags are expected to be set this time. Still, the interrupt signal is expected to remain silent, as no interrupts are enabled in the mask register.

Next, **COUNT** is set to a value higher than **COMPARE**, and the compare match interrupt is enabled. Now, both interrupt flags are expected to be set, and the interrupt signal should be generated. The compare match flag is also cleared and expected to re-occur, because **COUNT** still is higher than **COMPARE**.

COUNT is then set to 0 and **COMPARE** increased, and both the interrupt flags and signal are expected to still be set because the flags have not been manually cleared yet. The next test consists of clearing both interrupt flags, and then verifying that none of the interrupt flags are set and the interrupt signal is inactive. The test then waits for **COUNT** to reach **COMPARE**, and expects an interrupt signal and only the compare match flag to be set.

Finally, the overflow interrupt is tested by disabling the compare match interrupt, enabling the overflow interrupt, setting **COUNT** to a value that will overflow after a short while, and then expecting both interrupt flags and the interrupt signal to be high.

To make the test more readable and simplify development, the function `expect_irq` was created. It takes the expected compare match flag, overflow flag and interrupt state as parameters, and verifies all of them.

7.2.6 Interrupt State with Different Combinations of COUNT and COMPARE

To ensure that the interrupt signal has a correct state at all times, all possible chronological and numeric orders of setting **COUNT** and **COMPARE** was verified, by request from Gregertsen. These combinations are described in Table 8, and the results are computed on the basis that the interrupt should be set whenever **COUNT** is greater than or equal to **COMPARE**. Registers given in the first column are changed, with a value relative to the other register as described in the second column. Expected interrupt state is given in the third column.

Table 8 – test combinations

Register changed	Value	Interrupt
COUNT	< **COMPARE**	None
COUNT	> **COMPARE**	Immediately
COUNT	= **COMPARE**	Immediately
COMPARE	< **COUNT**	Immediately
COMPARE	> **COUNT**	None
COMPARE	= **COUNT**	Immediately

For instance, the first row in Table 8 means that **COMPARE** initially contains a value, and **COUNT** is set to a value lower than this. No interrupt is expected to happen in that situation.

7.2.7 No Clock Cycles are "Lost" hen Swapping Registers

When switching tasks, it is important that no cycles are "lost", for instance if the TMU by error stops counting while writing to registers. This test simulates switching of tasks by initiating swaps and keeping track of the returned number of clock cycles. It has a task that sums up the total number of clock cycles stored in the internal test registers (similar to tasks' contexts), excluding the active task, and then adds the current value of **COUNT**. This is compared with the actual number of clock cycles the module has been active since it was enabled after reset.

To make simulation easy, a context switch task named `activate` writes the **COUNT** value of the new task into the swap register, and then reads the **COUNT** value for the previous task and stores it into the internal test registers.

Another task, `run_task`, activates a task by using `activate` and lets it run a given number of clock cycles by waiting before continuing. The actual test is performed by switching tasks several times by using `run_task`. Instead of verifying the expected values of the TMU counter and the inactive tasks' counters only at specified points, this verification is performed automatically every clock cycle. This is to ensure that the total count is correct at all instants.

7.3 Functional Testing of Module Integrated with UC3

By using Atmel's tools for verification in a simulated environment, it is possible to run programs written in C on the UC3 microcontroller which includes the TMU module. Tests were developed in this environment of several reasons. First, all functions of the driver explained in chapter 6 were exercised, which is important to verify that the driver works correctly. Second, this allows for high-level testing of the complete module, and verifies that it works correctly together with the UC3 and other peripherals. Third, the ability to run C programs enables creating demonstrations that displays how the module performs in reality, with for instance delays caused by buses and other units being detected. This has been used when assessing the performance of the module, as described in 7.4. Finally, these tests can very well be used as example code demonstrating how the TMU can be used from software.

Atmel's tools for verification include simple libraries, where for instance a `printk` function can be found. This resembles `printf` in some ways, but does not allow printing of 64-bit values, and prints a newline after each call. A separate `print64` function was created to allow printing of 64-bit values. The print function executes on the simulated microcontroller, and thus affects the execution time of the code.

Each test resides in the folder `local/module_test` in the verification tools folder, where `module` is replaced by the module's name, and `test` is replaced by the name of the actual test. When a test is to run, the name of the folder is specified, and the tools will run a C or assembly file by the same name inside the folder. An `options` file also exists, which includes the value the program is expected to return on successful completion. This enables returning a different value if the test does not complete successfully, and can be used to display useful diagnostics by returning the value of the erroneous tested variable.

Table 9 gives a short summary of the various tests. Each test program is further described in the following sections.

Table 9 – integrated UC3 and TMU test summary

	Name	Verified elements	Depends on	Pass
1	`tmu_lowlevel`	Low-level C driver functions	C driver	✔
2	`tmu_highlevel`	High-level C driver functions	C driver	✔
3	`tmu_tasks`	Remaining high-level C driver functions	C driver	✔
4	`tmu_swap`	Demonstrates execution time of swap	-	✔

7.3.1 Low-level C Driver Functions

This test exercises all function described in the low-level part of the driver (6.2). First, the status register is read by using `TMU_getStatus` and expected to equal 0. The module is then enabled and disabled using `TMU_control`, and each time the enable flag of the status register is verified.

Then, enabling and disabling of interrupts by `TMU_interruptEnable` and `TMU_interruptDisable` is performed, and the resulting interrupt mask is verified by using `TMU_getInterruptMask`.

All read/write registers are tested using their respective setter and getter functions, hereby

- `TMU_setCount`
- `TMU_getCount`
- `TMU_setCompare`
- `TMU_getCompare`
- `TMU_setSwapCompare`
- `TMU_getSwapCompare`
- `TMU_setSwapCount`
- `TMU_getSwapCount`

Last, interrupts are provoked by setting the counter and compare register to corresponding values. The status register is read again each time, and `TMU_statusClear` is executed to verify successful clearing of the status flags.

7.3.2 High-level C driver Functions

Some of the high-level functions of the C driver are tested by initializing the module using `TMU_init` and enabling both interrupts, setting **COMPARE** to a value that will eventually generate a compare match interrupt, and waiting for that interrupt. When the interrupt occurs, the source is found by calling `TMU_checkOverflow` and `TMU_checkCompareMatch`. These functions also exercise `TMU_clearStatus`.

Then, the compare match interrupt is disabled, so only the overflow interrupt remains. The counter is set to a very high value, and the program waits again. When the counter overflows, the interrupt handler is again called, and the program can finish.

The remaining high-level functions are used in the test described in the next section.

7.3.3 Task Switching Example

This example simulates cycling through a given number of tasks a specified number of times. The tasks' **COUNT** values are stored when they are swapped out, and restored when swapped in. The tasks have varying budgets. The stored **COUNT** value, the overhead computed by subtracting the budget (increment in **COMPARE** value) from the increase in **COUNT** value since last run, and the allowed budget, are logged for each task switch. This is printed just before the program exits to reduce overhead while tasks execute.

The interrupt handler keeps track of how many times the program has cycled through all tasks, and stops the program when finished. It also checks for overflow and displays a message if this occurs. Finally, it checks for a compare match and activates the next task when appropriate.

This program exercises the remaining high-level functions of the driver from the previous program, but more important, it serves as a demonstration for how the TMU is intended to operate by swapping in and out tasks' contexts and storing the inactive tasks' values in memory. However, it does not serve as a good example of the performance in context switching as the logging facilities and other code cause large overhead. The final program which is described in the next section gives a better indication of swapping performance.

7.3.4 Assembly Swap Example

This program consists of a series of assembly instructions, wrapped in a C main function for simplicity. The test first writes the address of the TMU to **R8**, and the value 1 to **R7** which is both the value needed to be written to the control register to enable the module, and to the interrupt enable register to enable compare match interrupts. The module and compare match interrupt is then enabled. Next, registers **R4-R7** are filled with values intended to be swapped with the TMU's values. The values written to **COUNT** and **COMPARE** are 0 and 16, respectively, so that a compare match interrupt will be generated shortly.

Then, the TMU's swap registers are addressed by adding the corresponding offset from **R8**, and registers **R4-R7** are stored to this location. This fills the swap registers and executes the swap operation in the TMU. The swap registers are then immediately loaded back to register **R4-R7**, which after this operation contains the TMU's previous values.

This program is very simple, but allows for determining how many cycles are required to perform a swap. This is further explored in 7.4.

7.4 Performance Measurement

How many clock cycles are required for various operations is an important measure of the TMU's performance. If it is going to be used to monitor the execution time of interrupt handlers, where low overhead is very important, the number of clock cycles required to switch context can have a significant impact of the performance of the total system.

As the TMU is implemented, it supports burst reads and writes on the APB bus; hence each 32-bit operation can take as little as two clock cycles. However, it is not certain that the UC3 will provide control signals that rapidly. For instance, as the APB is connected through a bridge connected to the AHB, delays and pipeline stalls might occur.

To measure the performance of read and write operations, the program described in 7.3.4 was executed on the simulated UC3 with TMU. The waveforms of the relevant signals are shown in Figure 11, and the required amount of clock cycles for the reading and writing is summarized in Table 10.

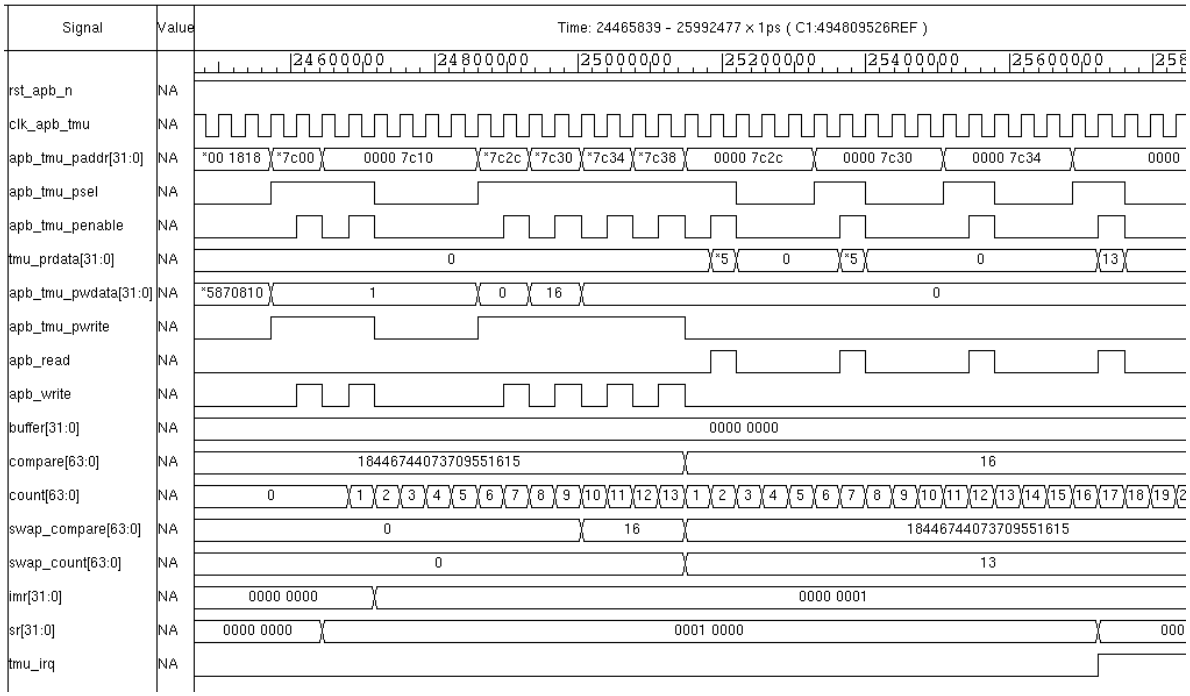

Figure 11 – simulation of swap operation

First, the module is enabled, and then the compare match interrupt is enabled. This can be seen in the four first clock cycles where `apb_tmu_psel` is high.

In the four next cycles, the TMU is not interacted with, as the CPU's internal registers are filled with values to be written to the TMU.

Then, the swap operation is initiated, by writing to the two 64-bit swap registers. This is performed in a total of 8 clock cycles, which is the minimum possible because of the APB's two-cycle access to a 32-bit register. This operation can be found in the figure by looking at the next (and longest) time `apb_tmu_psel` is high.

Next, the swap registers are read back. Note that this requires considerably more clock cycles, 5 for each read operation, which gives a total of 20. This is due to that the APB bridge does not pipeline read accesses.

This slow reading emphasizes the importance of the optimization suggested in 4.10, as the swap operation could then be reduced from 28 to 18 clock cycles, a decrease of 36%.

Table 10 – number of cycles for reading and writing registers on the UC3 with TMU

Operation	Number of cycles
Write register	2
Read register	5

Also, note that the compare register is set to 16 and the counter is set to 1 in the swap. The status register changes state and the interrupt signal goes high the clock cycle after the counter reaches 16, in accordance with the explanation in 4.6.

7.5 Size and Cost

The TMU was synthesized with the purpose of computing its area. The required amount of gates was 4 678, which can be considered as relatively small in a microcointroller according to discussions with Chong-Fatt Law. This suggests that the module can be integrated into a real microcontroller without having to make too many compromises regarding die area.

8 Discussion

This chapter first presents the approach taken in performing this project, and discusses some of its advantages and drawbacks. Then, choices related to the implementation of the TMU are discussed. The third section presents and discusses three alternative ways of supporting execution time control in hardware, and gives a comparison of the alternatives to the TMU implemented in this project. Then, some considerations about the instant of task swapping and the accounting of clock cycles are presented. Next, the performance, cost and flexibility of the module is briefly discussed, before some considerations about the impact of other research this TMU will have are given. Then, limitations of the implementation are mentioned. Finally, the range of application of the TMU is discussed.

8.1 Approach

As the author of this thesis had no experience with Verilog before this project, and little experience with digital hardware design; the first part of the project consisted of gaining knowledge about this field and language. At the same time, there were several articles and books containing information relevant to this project that needed to be read.

Because of the lack of former experience, it was difficult to tell when it was possible to begin implementing the TMU. Also, at the beginning of the project, it was uncertain whether the project would get support from Atmel, which would affect the overhead of the project with regards to tools and support significantly.

Hence, it was decided to first implement the TMU as originally specified, and then suggest improvements and implement them as the time allowed. This would significantly increase the probability of being able to finish the project with a functional product, which was important because of Gregertsen and Skavhaug's research.

Even though learning Verilog and principles of digital design required a significant amount of time, there was time both to implement the module as originally specified, and to suggest and implement improvements. This should presumably be credited to Atmel, as the time needed to set up and configure tools and software was reduced in a large amount compared to what would be the case if the author of this thesis would have to implement the module without this support. Advices about design methodology were given as well, more specifically; that it is not necessary or even usual to synthesize the module to run it on an FPGA and that simulation could be performed instead. This allowed development time to be further reduced.

While waiting for a response from Atmel in the beginning of the project, some time was spent exploring development environments and tools for prototyping. This research did not contribute much to the project, mainly because of operating system incompatibilities and outdated software from the vendors. With the benefit of hindsight, this time could be better utilized by reading literature or doing other tasks that would have to be performed whether Atmel would support the project or not. However, the rest of the planning worked out well, which is also proven by the successful results of the project.

8.2 Choices Taken

8.2.1 Interfacing the TMU with the APB

(1) suggests implementing the TMU by interfacing it to the UC3 using the AHB, which is faster but has a more complicated interface than the APB. Because of the limited time allowed for implementing the TMU, Atmel's supervisor Dr. Chong-Fatt Law recommended to interface the TMU to the UC3 via the APB instead. This gives a performance penalty compared with a device interfaced with the AHB, and using the AHB could be considered if the TMU is required to respond and transfer data quicker.

However, as the UC3 microcontrollers implement the AHB as a matrix with direct connections between masters and slaves, this would cause a significant increase of die area. On products like the UC3L, only a few units are connected to the AHB (17) (named High Speed Bus Matrix in the datasheets). Hence, interfacing the TMU with the AHB instead of APB does not seem like a viable option.

However, the UC3 models do have a CPU local bus, where some of the registers in the GPIO module are mapped. The same registers are also mapped on the Peripheral Bus (APB). By mapping the registers on the local bus, read and write operations can be performed by a single clock cycle. Also, the transfer time becomes deterministic, as the CPU and GPIO are the only modules connected to this bus (17).

If deterministic single-cycle access is desired, it is recommended to explore the option of connecting the TMU to the CPU local bus as well as the APB.

8.2.2 Implemented Improvements

As explained in chapter 4, several improvements were suggested, and a subset of these was implemented. The details of which improvements that were implemented is given in that chapter, but the chosen improvements can be summarized as the ones that complete the module, in the sense that it can be regarded as a fully functional module in the same way as other modules currently on the UC3 models. These features mainly focus on giving the user the possibility of controlling the module in various ways, and would be difficult to implement at a later stage. By implementing these features, the module contains a basic set of status and configuration registers that can easily be extended if new features arise.

The improvements that were not implemented are mainly optimizations that could easily be implemented later if they seem beneficial. Hence, the resulting product is a fully usable module that can be utilized both for research and real world applications, and is a good foundation for further optimizations and customizations.

8.3 Differences to Other Implementations

Several other implementations that could serve as hardware support for execution time monitoring and control exist, and have various advantages and drawbacks with respect to the needs of Gregertsen and Skavhaug's research. Three other implementations are studied in this section, and a comparison between these and the TMU implemented in this project is summarized in a table in the end.

8.3.1 Forsman's Implementation of a TMU

In 2008, Bjørn Forsman implemented a TMU (7) based on a specification given in (8) as his master thesis. The TMU included a task timer similar to the TMU in this project, in addition to a timer for each interrupt level.

Interrupt Blocking

As opposed to the implementation in this project, Forsman's unit intervenes in interrupt signals. Although (1) states that it is possible to "protect the systems from bursts of interrupts that could otherwise result in tasks missing their deadline", without physically blocking the interrupt signal to the processor, the interrupt handler will be called as long as interrupts are enabled for that level. This can lead to starvation for other processes.

Similar advantages of blocking interrupts could be attained by letting software mask out interrupts for a given level when they exceed their budgets. This would result in slightly more overhead, but allows for a more dynamic and portable approach, and reduces the die area required greatly.

Versatility

The TMU implemented by Forsman supports the Deferrable Server scheduling algorithm for the interrupts. By doing this in hardware, no overhead is imposed on interrupt limiting, though, the user is bound to use this algorithm. The implementation in this project requires the scheduling policies to be implemented in software, which requires some extra overhead, but gives the user full flexibility.

Size and Cost

As one timer is required for each interrupt level in Forsman's implementation, the die area grows with the number of interrupt levels that need to be handled. With 16 interrupt levels, Forsman's implementation required 6 348 LUTs (Look-Up Tables). 4 levels required 1 471 LUTs. The TMU without IRQ timers used only 262 LUTs. In comparison, the LEON3 core including 8 + 8 Kbyte cache required approximately 4 300 LUTs.

Unfortunately, LUTs do not directly translate to a number of gates, and (7) does not specify if other logic of the FPGA is employed. Hence, a direct comparison of the TMU implemented in this project and the TMU Forsman implemented is impossible. Still, depending on the functions the LUTs represent, one can assume that a function implemented in a LUT can be implemented by using 4-6 gates.

By assuming that no additional functions of the FPGA were used, and using the most conservative ratio (4) and the same amount of interrupt levels that exist on the UC3L (4), Forsman's TMU would roughly require 5 900 gates. In comparison, the TMU implemented in this project was synthesized with an area of 4 678 gates. This suggests that the latter is a relatively small unit, especially when its 64-bit registers are taken into account.

It is worth noting that the TMU implemented in this project scales well with an increasing number of interrupt levels, as it needs no additional logic as opposed to Forsman's implementation.

Possibility of Integration in Commercial Products

As discussed in the previous sections, Forsman's TMU has some advantages regarding overhead, but lacks the versatility and smaller size of the TMU implemented in this project. This suggests that the latter is a better candidate for integrating into commercial products.

8.3.2 Microcontroller Timer

Many microcontrollers have built-in timers, with the possibility of generating interrupts on various conditions. These could have been used for supporting a software implementation of execution time control, but lacks some important features:

- Most have limited resolution, for instance the Timer/Counter of UC3L is only 16-bit (17). The hardware support for measuring absolute execution time, as needed for implementing execution time monitoring for POSIX (26) and Ada (27), will be limited with only 16 bits and require calculating absolute execution time in software, resulting in larger overhead.
- Atomic swapping is not supported, and one would thus have to take extra measures to ensure that no cycles can be lost, and no race conditions can occur.

Hence, the built-in timers will cause significant overhead as the timers will frequently overflow. On the other hand, they have a widespread availability as timers are a common feature of commercial microcontrollers.

8.3.3 COUNT and COMPARE of AVR32

The AVR32 architecture specifies **COUNT** and **COMPARE** registers (11) which can be used for execution time management. **COUNT** increases by one every clock cycle, and can both be read and written. The value in **COUNT** is compared against the value in **COMPARE**, and an interrupt is generated and **COUNT** is reset to zero when they match.

There are two disadvantages for using these for hardware support of the run-time system implemented by Gregertsen and Skavhaug (10). The registers contain only 32 bits, which would require overflow handing in software and thus generate extra overhead.

As **COUNT** is automatically reset to zero when a compare match occurs, the implementation is not useful for measuring absolute execution time. However, the UC3 implementation of the AVR32A architecture additionally implements an option to disable automatic clearing (20).

This implementation also lacks support of atomically swapping **COUNT** and **COMPARE** registers.

An advantage of the UC3 implementation is that the registers are clocked by a dedicated clock with the same frequency as the CPU clock, which allows them to operate in some sleep modes (20). Also, it has the benefit of already being implemented on all UC3 models.

8.3.4 Comparison of the Various Implementations

Table 11 – feature comparison of the various implementations

Feature	Forsman's implementation	Generic microcontroller timer	AVR32 COUNT and COMPARE	This project's implementation
64-bit registers	⊘	⊘	⊘	✔
Atomic swapping	⊘	⊘	⊘	✔
Overhead	✔ Low for tasks, no for interrupts	⊘ Overflow handling	⊘ Overflow handling	✔ Low for tasks and interrupts
Hardware interrupt limiting	✔	⊘	⊘	⊘
Flexibility	⊘ Fixed scheduling algorithm	✔	✔	✔
Versatility	⊘ Predetermined usage	✔	✔ Can be used as a simple timer	✔ Can be used as a simple timer
Availability	⊘ Does not exist in current products	✔	✔	⊘ Does not exist in current products
Additional size and cost	? Larger size	✔ Often built-in	✔ Built-in	? Smaller size
Monitoring execution time	✔	✔	✔	✔
Budget deplete interrupt	✔	✔	✔	✔

✔ feature is available/sufficient
? sufficiency or availability is limited or unknown
⊘ feature is not available or sufficient

8.4 Instant of Task Swapping

Preparing for a swap between tasks takes a non-zero amount of clock cycles. At least, a new value of the counter will be written, and the previous value then read back, both using the swap register. Also, a new compare value will often be written. Reading of the previous compare value is expected to be unnecessary, and thus omitted, as proposed in 4.10.

As each value consists of 64 bits and the AMBA buses are 32 bits wide in UC3, the values need to be read or written using two 32-bit transfers. Each transfer takes at least two clock cycles, but the core, AHB interface or APB interface might delay transfers, for instance if other bus masters are using one of the buses or new transfers are delayed because of the core's internal manner of operation. In the following, it is assumed that no delays are present, so that transfers can be performed using two cycles with no delay between transfers.

A context switch consisting of a write of a new compare and count value (using **SWAP_COMPARE** and **SWAP_COUNT** registers, respectively) and a subsequent read of the previous count value (again, using **SWAP_COUNT**) would then require a total of 12 clock cycles. At one instant, the TMU must stop accounting the previous task of the clock cycles used, and start charging the new one, by setting the counter's value to the previous task's cycle count. This can be considered as the instant the tasks are swapped. The way the TMU is implemented, this moment is the beginning of the clock cycle after **SWAP_COUNT** has been written, that is, in the fifth cycle after the TMU received the first 32-bit word of the new compare value.

Intuitively, the swap process consists of two writes related to the new task (new count and compare value), and one read related to the previous task (previous count value). Note that the writes of values related to the new task need to be performed while the previous task is still active, and the read of the previous task's values must take place after this. At some point during these transfers, the task that is charged for the clock cycles is changed. This leads to the new task being charged for the previous task's bookkeeping (as the previous counter value is read after the switch of counter value) and vice versa. The amount of clock cycles belonging to the new and previous task is not necessarily the same, because of the imbalance in the number of registers required to be written or read in a swap, and also the possible difference between the time needed for reading and writing registers.

When using the TMU to implement execution time monitoring this can be important when considering which tasks are to be charged for overhead of context switching. Still, assuming that each switch requires the same amount of clock cycles on average, the number of cycles wrongly charged to a task when it is activated, will be "given back" when a new task is activated. In that sense, the tasks are not accounted with an incorrect number of clock cycles; the charging of cycles related to the context switch is merely delayed.

As shown in the performance measurements (7.4), the UC3 might use extra clock cycles when issuing an APB read, although a write has been observed to be performed without delays. Various factors such as pipeline stalls, cache misses and occupied buses might lead to further delays being introduced. This indicates that the assumption of the swap requiring 12 cycles is not reliable, and should not be used when it is important to know the exact number of clock cycles required to issue a task swap. However, the discussion above focuses on when a new task is charged for clock cycles, and the validity of the assumption is sufficient for illustrating the example.

8.5 Performance, Cost and Flexibility

The final product of this project is a TMU with the possibility of monitoring execution time with high accuracy and relatively low overhead. The overhead could further be reduced by implementing the scheduling algorithm on the module. This will however reduce the flexibility of the module, as the algorithm would have to be predetermined. In that sense, the implementation is a good compromise between performance and flexibility.

The performance could also be improved by having a dedicated timer for each interrupt, but this would increase its size as explained in 8.3.1. The argument of flexibility is also valid for not having dedicated interrupt timers.

Hence, this TMU gives a good balance of performance, cost and flexibility.

8.6 Impact of Other Research and Development

One of the main problems to be addressed in real-time systems is to ensure that the deadlines of all tasks are being met (schedulability). In lack of facilities for meeting this demand, research has been focusing on how to prove schedulability by using simplified models which do not necessarily reflect the real world. Tools for meeting the requirements have mostly been utilizing indirect means such as priorities based on deadlines. Relying on scheduling for the worst case execution time, which is hard or intractable to compute, can lead to low CPU utilization because of conservative execution time budgets. On the other hand, using heuristics for determining priorities might imply having to risk missing deadlines because of too optimistic budgets.

The implementation of this TMU will first and foremost have an impact on Gregertsen and Skavhaug's research, as it was based on their specifications and created because of their needs. On the other hand, with the successful implementation of Ada's timing event and execution time control features (10), and furthermore the real-time framework (12), this research will demonstrate the possibility of having accurate and precise execution time monitoring without having to afford large overhead or complex hardware support.

With the introduction of dedicated hardware support for execution time management, research can hopefully be directed towards addressing the main problem directly, that is, tasks meeting their deadlines, and how to handle the situation if tasks fail to complete within their execution time budget. Also, the results of this project can be used directly to exploit a CPU's capacity to a larger extent.

The research and development performed in this project can also serve as a foundation for master students extending the project by further developing the module.

8.7 Limitations

Even though the final product meets the specification and includes improvements, it is not perfect. As interrupt handlers should have as little overhead as possible, it would be desirable to reduce the number of cycles required for a swap to the largest possible extent. As the bus it is implemented on does not use it to its full potential as explained in 8.2.1, the TMU gives more overhead than what could be achieved. Theoretically, it should be possible to transfer a 32-bit register in one cycle if the TMU was integrated in the fastest possible way.

One could also argue against having a 64-bit counter on a 32-bit processor. The counter needs to transfer the carry bit between all 64 flip-flops in the register, hence a lot of combinational

circuits will be connected in series. This increases the critical combinational path, and can possibly introduce a new bottleneck in the system, which further determines the maximum clock speed. Alternative solutions, such as splitting the counter in two and perform the increment over two clock cycles could resolve this challenge. This would add complexity to the module in other ways as well, as some protection against reading the counter when in an inconsistent state would have to be implemented.

The current implementation restricts the TMU to be clocked with the APB clock. If this clock is running with the same frequency as the CPU clock, this is satisfactory. However, the APB clock can be prescaled by the user for various reasons, which will make the clock run at lower clock frequencies ($\frac{1}{2}, \frac{1}{4}$ and so on). Even though this is easy to handle in software, it might not always be ideal to run the TMU at a lower speed than the CPU clock or have to run the APB at its full clock speed.

Finally, the peripheral bus makes the TMU non-deterministic, as the number of clock cycles required for each operation is not guaranteed. For instance, if the bus is occupied, the transfer would be delayed. This can be a disadvantage for a real-time system, where exact control over various operations often is critical.

8.8 Application Range

Primarily, this module supports Gregertsen and Skavhaug's implementation of Ada's timing event and execution time control (10) and the real-time framework (12) based on the former, and will be used for execution time monitoring and control in real-time systems. Previously, implementations of Ada have been allowed to charge interrupt handlers to an arbitrary task, if any, because of the fast response requirements. By having the possibility of measuring execution time with very low overhead, the execution time of interrupt handlers can now also be measured and accounted correctly. However, the range of use is not limited to this.

Compared to the existing timers of UC3 models, the TMU has a very simple interface. This can be an advantage if the user wants to start a timer with low overhead (to start counting, it is sufficient to enable the module), or does not require all features of a conventional counter.

Also, applications which require measuring absolute time without having to consider overflow handling would benefit from having a 64-bit counter available.

As the module has been implemented with a focus on configuration options for the user and versatility, the application range evidently extends beyond the primary intended application.

9 Conclusion and Further Work

This project has lead to a successful implementation of a Time Management Unit as specified by Gregertsen and Skavhaug (1). Furthermore, important improvements additional to the specification, such as configurable interrupts and the possibility to enable and disable the module, have been implemented. A user guide similar to a module's part of an UC3 datasheet and a framework consisting of drivers in several abstraction layers are provided. Finally, thorough tests have been created and performed both on the module as a stand-alone unit, and on the module integrated with an AVR32 UC3 processor in a simulated environment.

Together with the software created in (10; 12), the TMU implemented in this project allows for precise and accurate execution time monitoring and control with low overhead. As the module has been specified and implemented with small footprint as an important aspect, the TMU can be integrated into future microcontrollers without having to make too many compromises. This can benefit real-time systems by allowing for higher utilization of the CPU core, without having to risk timing failures due to tasks exceeding their execution time budget. Also, by exploiting the concept of primary and alternative tasks, the TMU makes it practically possible to guarantee that either the primary task will be executed to completion within its deadline, or that the alternative task will be executed, even for tasks with very short deadlines.

At the time of project completion, the module is fully usable, and can benefit both research activities and be included in new models of processor series currently available on the market. However, further potential improvements are identified in the project. For instance if lower overhead is demanded, the module could be connected to a higher speed bus, or even be integrated with the CPU core itself. More suggestions and advice for further work is given in the report.

The TMU is also prepared for further modifications, by for instance reserving bits in the status register and having developed code with focus on maintainability. The thorough set of tests can support further development by indicating whether the module is still functional and backwards compatible. Finally, an introduction to Verilog for programmers familiar with C has been created, in order to give additional support to students continuing the work with this project.

The author of this thesis, as well as the others that have been involved in this research, hope that this module will be included in future commercial microcontrollers. Many of the choices taken in this project have been made with this goal in mind. However, a requirement for the TMU to be included in a product line is that it must be regarded as a selling point by the manufacturer. This implies that the module must offer a desirable feature to the end user, which means that software taking advantage of the module also should be available. Hence, the ongoing research of *Implementing the new Ada 2005 timing event and execution time control features on the AVR32 architecture* (10), and creating *A real-time framework for Ada 2005 and the Ravenscar profile* (12) is paramount for the wish of seeing the module in commercial microcontrollers.

References

1. *Functional specification for a Time Management Unit.* **Gregertsen, Kristoffer Nyborg and Skavhaug, Amund.** Vienna, Austria : SAFECOMP, 2010. 29th International Conference on Computer Safety, Reliability and Security.

2. **Wikimedia Foundation, Inc.** Advanced Microcontroller Bus Architecture. *Wikipedia.* [Online] December 10, 2010. [Cited: December 12, 2010.] http://en.wikipedia.org/wiki/Advanced_Microcontroller_Bus_Architecture.

3. **Atmel Corporation.** *Document Standards (ATMEL CONFIDENTIAL).* - : Unpublished, 2010. I7000-0006.

4. **Burns, Alan and Wellings, Andy.** *Real-Time Systems and Programming Languages.* Essex : Pearson Education Limited, 2001.

5. **Wilhelm, Reinhard et al.** The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems.* 2008, Vol. 7, 3.

6. **Krishna, C. M. and Shin, Kang G.** *Real-Time Systems.* New York : McGraw-Hill, 1997.

7. **Forsman, Bjørn.** *En Time Management Unit (TMU) for sanntidssystemer.* Trondheim : ITK, NTNU, 2008.

8. *Hardware Support for On-Line execution Time Limiting of Tasks in a Low-Power Environment.* **Skinnemoen, Håvard and Skavhaug, Amund.** Linz : Institute of System Science, Johannes Kepler University, 2003. EUROMICRO/DSD Work in Progress Session. ISBN: 3-902457-21-X.

9. **ARM Limited.** AMBA Specification (Rev 2.0). *ARM.* [Online] 1999. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0011a/index.html.

10. *Implementing the new Ada 2005 timing event and execution time control features on the AVR32 architecture.* **Gregertsen, Kristoffer Nyborg and Skavhaug, Amund.** 10, New York, NY, USA : Elsevier North-Holland, Inc., 2010, Journal of Systems Architecture: the EUROMICRO Journal, Vol. 56. ISSN: 1383-7621.

11. **Atmel Corporation.** AVR32 Architecture Document. *AVR Solutions.* [Online] November 2007. http://www.atmel.com/dyn/resources/prod_documents/doc32000.pdf.

12. *A real-time framework for Ada 2005 and the Ravenscar profile.* **Gregertsen, Kristoffer Nyborg and Skavhaug, Amund.** Patras, Greece : EUROMICRO, 2009. 35th Euromicro Conference on Software Engineering and Advanced Applications.

13. **Bovet, Daniel P. and Cesati, Marco.** Process Scheduling. *Understanding the Linux Kernel.* [Online] O'Reilly, October 2000. [Cited: December 19, 2010.] http://oreilly.com/catalog/linuxkernel/chapter/ch10.html.

14. **Wikimedia Foundation, Inc.** Earliest deadline first scheduling. *Wikipedia.* [Online] December 4, 2010. [Cited: December 19, 2010.] http://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling.

15. **Atmel Corporation.** AVR32 32-Bit Microcontroller AT32UC3A Preliminary Datasheet. *AVR Solutions.* [Online] November 2009. [Cited: November 28, 2010.] http://atmel.com/dyn/resources/prod_documents/doc32058.pdf.

16. —. AVR and AVR32 - Quick Reference Guide. *AVR Solutions.* [Online] [Cited: October 21, 2010.] http://atmel.com/dyn/resources/prod_documents/doc4064.pdf.

17. —. AVR32 32-Bit Microcontroller AT32UC3L Preliminary Datasheet. *AVR Solutions.* [Online] November 2010. [Cited: November 28, 2010.] http://atmel.com/dyn/resources/prod_documents/doc32099.pdf.

18. —. 8- and 32-bit low power, high performance MCU. *AVR Solutions.* [Online] [Cited: December 15, 2010.] http://www.atmel.com/products/AVR/cpu.asp?family_id=607.

19. **Wikimedia Foundation, Inc.** Orthogonal instruction set. *Wikipedia.* [Online] November 17, 2010. [Cited: December 15, 2010.] http://en.wikipedia.org/wiki/Orthogonal_instruction_set.

20. **Atmel Corporation.** AVR32UC Technical Reference Manual. *AVR Solutions.* [Online] 2010. http://atmel.com/dyn/resources/prod_documents/doc32002.pdf.

21. **Tala, Deepak Kumar.** Design And Tool Flow. *World of ASIC.* [Online] May 30, 2010. http://www.asic-world.com.

22. **IEEE Computer Society.** *IEEE Standard Verilog Hardware Description Language.* New York : The Institute of Electrical and Electronics Engineers, Inc., 2001. 1364-2001.

23. **Wikimedia Foundation, Inc.** Verilog. *Wikipedia.* [Online] October 21, 2010. [Cited: October 28, 2010.] http://en.wikipedia.org/wiki/Verilog.

24. **Chandrakasan, Anantha P., Sheng, Samuel and Brodersen, Robert W.** Low-Power CMOS Digital Design. *IEEE Journal of Solid-State Circuits.* 1992, Vol. 27, 4.

25. **Cadence Design Systems, Inc.** Techincal Paper - Clock Domain Crossing. [Online] [Cited: December 12, 2010.] http://w2.cadence.com/whitepapers/cdc_wp.pdf.

26. **IEEE and The Open Group.** *IEEE 1003.1 Standard for Information Technology - Portable Operating System Interface (POSIX).* New York : Institute of Electrical and Electronics Engineers, Inc., 2001.

27. **Ada-Europe.** Execution Time. *Ada Reference Manual.* [Online] 2006. [Cited: December 12, 2010.] http://www.adaic.org/resources/add_content/standards/05rm/html/RM-D-14.html.

28. **Williams, David M.** Just what makes Linux tick. *iTWire.* [Online] February 21, 2008. [Cited: December 18, 2010.] http://web.archive.org/web/20080227234050/http://www.itwire.com/content/view/16778/1141/.

Appendices

A–1 TMU User Interface

This is the remaining register descriptions of the user interface description in 5.7.

A–1.1 Control Register

Name: CTRL
Access Type: Write-Only
Offset: 0x00
Reset Value: 0x00000000

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-

23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-

7	6	5	4	3	2	1	0
-	-	-	-	-	-	DIS	EN

- **DIS: Disable Module**
  1: Writing a one to this bit will disable the TMU. Registers are still available.
  0: Writing a zero to this bit has no effect.
- **EN: Enable Module**
  1: Writing a one to this bit will enable the TMU to count, generate interrupt flags and interrupt signals.
  0: Writing a zero to this bit has no effect.

A–1.2 Mode Register (Reserved for Future Use)

Name: MODE
Access Type: -
Offset: 0x04
Reset Value: 0x00000000

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-

23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-

7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-

A–1.3 Status Register

Name: SR
Access Type: Read-Only
Offset: 0x08
Reset Value: 0x00000000

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-

23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	EN

15	14	13	12	11	10	9	8
-	-	-	-	-	-	PREVOVF	PREVCMP

7	6	5	4	3	2	1	0
-	-	-	-	-	-	OVF	CMP

- **EN: Enable Status**
  0: The TMU is disabled.
  1: The TMU is enabled.
  This bit is cleared when CTRL.DIS is written to one.
  This bit is set when CTRL.EN is written to one.
- **PREVOVF: Previous Overflow Flag**
  This bit is cleared when a swap operation is issued and the previous value of OVF was 0 and no overflow occurred during the swap.
  This bit is set when a swap operation is issued and the previous value of OVF was 1 or an overflow occurred during the swap.
- **PREVCMP: Previous Compare Match Flag**
  This bit is cleared when a swap operation is issued and the previous value of CMP was 0 and no compare match occurred during the swap.
  This bit is set when a swap operation is issued and the previous value of CMP was 1 or a compare match occurred during the swap.
- **OVF: Overflow Flag**
  This bit is cleared when a one is written to SCR.OVF or a swap operation is issued.
  This bit is set when an overflow of COUNT occurs.
- **CMP: Compare Match Flag**
  This bit is cleared when a one is written to SCR.CMP or a swap operation is issued.
  This bit is set when COUNT is equal to or greater than COMPARE.

A–1.4 Status Clear Register

Name: SCR
Access Type: Write-Only
Offset: 0x0C
Reset Value: 0x00000000

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-

23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-

7	6	5	4	3	2	1	0
-	-	-	-	-	-	OVF	CMP

Writing a zero to a bit in this register has no effect.
Writing a one to a bit in this register will clear the corresponding bit in SR and the corresponding interrupt request.

A–1.5 Interrupt Enable Register

Name: IER
Access Type: Write-Only
Offset: 0x10
Reset Value: 0x00000000

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-

23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-

7	6	5	4	3	2	1	0
-	-	-	-	-	-	OVF	CMP

Writing a zero to a bit in this register has no effect.
Writing a one to a bit in this register will set the corresponding bit in IMR.

A–1.6 Interrupt Disable Register

Name: IDR
Access Type: Write-Only
Offset: 0x14
Reset Value: 0x00000000

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-

23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-

7	6	5	4	3	2	1	0
-	-	-	-	-	-	OVF	CMP

Writing a zero to a bit in this register has no effect.
Writing a one to a bit in this register will clear the corresponding bit in IMR.

A–1.7 Interrupt Mask Register

Name: IMR
Access Type: Read-Only
Offset: 0x18
Reset Value: 0x00000000

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-

23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-

15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-

7	6	5	4	3	2	1	0
-	-	-	-	-	-	OVF	CMP

0: The corresponding interrupt is disabled.
1: The corresponding interrupt is enabled.
A bit in this register is cleared by writing a one to the corresponding bit in IDR.
A bit in this register is set by writing a one to the corresponding bit in IER.

A–1.8 Compare Register (High Part)

Name: COMPARE_HI
Access Type: Read/Write
Offset: 0x1C
Reset Value: 0xFFFFFFFF

31	30	29	28	27	26	25	24
COMPARE_HI[31:24]							

23	22	21	20	19	18	17	16
COMPARE_HI[23:16]							

15	14	13	12	11	10	9	8
COMPARE_HI[15:8]							

7	6	5	4	3	2	1	0
COMPARE_HI[7:0]							

- **COMPARE_HI: Compare Register (High Part)**

  COMPARE_HI contains the high part of the 64-bit COMPARE value.

A–1.9 Compare Register (Low Part)

Name: COMPARE_LO
Access Type: Read/Write
Offset: 0x20
Reset Value: 0xFFFFFFFF

31	30	29	28	27	26	25	24
COMPARE_LO[31:24]							

23	22	21	20	19	18	17	16
COMPARE_LO[23:16]							

15	14	13	12	11	10	9	8
COMPARE_LO[15:8]							

7	6	5	4	3	2	1	0
COMPARE_LO[7:0]							

- **COMPARE_LO: Compare Register (Low Part)**

  COMPARE_LO contains the low part of the 64-bit COMPARE value.

A–1.10 Count Register (High Part)

Name: COUNT_HI
Access Type: Read/Write
Offset: 0x24
Reset Value: 0x00000000

31	30	29	28	27	26	25	24
COUNT_HI[31:24]							

23	22	21	20	19	18	17	16
COUNT_HI [23:16]							

15	14	13	12	11	10	9	8
COUNT_HI [15:8]							

7	6	5	4	3	2	1	0
COUNT_HI [7:0]							

- **COUNT_HI: Count Register (High Part)**

  COUNT_HI contains the high part of the 64-bit COUNT value.

A–1.11 Count Register (Low Part)

Name: COUNT_LO
Access Type: Read/Write
Offset: 0x28
Reset Value: 0x00000000

31	30	29	28	27	26	25	24
COUNT_LO[31:24]							

23	22	21	20	19	18	17	16
COUNT_LO [23:16]							

15	14	13	12	11	10	9	8
COUNT_LO [15:8]							

7	6	5	4	3	2	1	0
COUNT_LO [7:0]							

- **COUNT_LO: Count Register (Low Part)**

  COUNT_LO contains the low part of the 64-bit COUNT value.

A–1.12 Compare Swap Register (High Part)

Name: SWAP_COMPARE_HI
Access Type: Read/Write
Offset: 0x2C
Reset Value: 0x00000000

31	30	29	28	27	26	25	24
SWAP_COMPARE_HI[31:24]							

23	22	21	20	19	18	17	16
SWAP_COMPARE_HI [23:16]							

15	14	13	12	11	10	9	8
SWAP_COMPARE_HI [15:8]							

7	6	5	4	3	2	1	0
SWAP_COMPARE_HI [7:0]							

- **SWAP_COMPARE_HI: Compare Swap Register (High Part)**
  SWAP_COMPARE_HI contains the high part of the 64-bit SWAP_COMPARE value.

A–1.13 Compare Swap Register (Low Part)

Name: SWAP_COMPARE_LO
Access Type: Read/Write
Offset: 0x30
Reset Value: 0x00000000

31	30	29	28	27	26	25	24
SWAP_COMPARE_LO[31:24]							

23	22	21	20	19	18	17	16
SWAP_COMPARE_LO [23:16]							

15	14	13	12	11	10	9	8
SWAP_COMPARE_LO [15:8]							

7	6	5	4	3	2	1	0
SWAP_COMPARE_LO [7:0]							

- **SWAP_COMPARE_LO: Compare Swap Register (Low Part)**
  SWAP_COMPARE_LO contains the low part of the 64-bit SWAP_COMPARE value.

A–1.14 Count Swap Register (High Part)

Name: SWAP_COUNT_HI
Access Type: Read/Write
Offset: 0x34
Reset Value: 0x00000000

31	30	29	28	27	26	25	24
SWAP_COUNT_HI[31:24]							

23	22	21	20	19	18	17	16
SWAP_COUNT_HI [23:16]							

15	14	13	12	11	10	9	8
SWAP_COUNT_HI [15:8]							

7	6	5	4	3	2	1	0
SWAP_COUNT_HI [7:0]							

- **SWAP_COUNT_HI: Count Register (High Part)**

  SWAP_COUNT_HI contains the high part of the 64-bit SWAP_COUNT value.

A–1.15 Count Swap Register (Low Part)

Name: SWAP_COUNT_LO
Access Type: Read/Write
Offset: 0x38
Reset Value: 0x00000000

31	30	29	28	27	26	25	24
SWAP_COUNT_LO[31:24]							

23	22	21	20	19	18	17	16
SWAP_COUNT_LO [23:16]							

15	14	13	12	11	10	9	8
SWAP_COUNT_LO [15:8]							

7	6	5	4	3	2	1	0
SWAP_COUNT_LO [7:0]							

- **SWAP_COUNT_LOW: Count Register (Low Part)**

  SWAP_COUNT_LOW contains the low part of the 64-bit SWAP_COUNT value.

  When SWAP_COUNT_LOW is written, the swap operation is issued as described in 5.5.5.

A–2 File archive

Due to the restrictions put by Atmel Corporation, the file archive will not be published with the digital version of this report. The printed booklet delivered to the examiners will have a CD with the following contents attached:

Location	Description
\Integrated tests and examples	Tests and example code for TMU integrated with UC3
\Report	Source files for the report
\Software	Support software and drivers
\Stand-alone tests	Testbench and stand-alone test files
\TMU	The Verilog files of the TMU implementation