**NTNU – Trondheim**
Norwegian University of
Science and Technology

# Implementing a Time Management Unit for the OR1200 Processor.

Kyrre Erlend Aspelund
Gonsholt
Lars Ødegaard

Electronics System Design and Innovation
Submission date: July 2014
Supervisor: Bjørn B. Larsen, IET
Co-supervisor: Amund Skavhaug, ITK

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

**Title:** Implementing a Time Management Unit for the OR1200 Processor
**Student:** Kyrre E. A. Gonsholt, Lars Ødegaard

**Problem description:**

The project will focus on implementing a time management unit to improve the real-time capabilities of the OR1200 OpenRISC processor. The implementation will be based on previous implementations of hardware based TMUs, but will be implemented as an integrated part of the processor's internal registers. The TMU will have to be able to work with an existing operating systems scheduler.

This should be verified by hardware simulation, firmware testing and software testing through an existing operating system. The final solution should run on an FPGA.

The project should also consider the usefulness of an hardware solution to the time management problem associated with scheduling.

**Responsible professor:** Bjørn B. Larsen
**Supervisor:** Amund Skavhaug

# Abstract

This thesis presents a Time Management Unit (TMU) that provides assistance to the scheduler and the interrupt handling of a real-time operation system. The unit provides functionality for monitoring task execution time and a mechanism for signalling when a task depletes its resources. This applies to both regular tasks and the handling of sporadic events. By putting the TMU inside a processor core, it has a more predictable impact on the overhead related to task switching.

The implemented TMU is tested as a stand-alone unit with a hardware testbench, and then integrated into the OpenRISC 1000 based OR1200 processor as special purpose registers. The behaviour of OR1200 is verified through hardware simulation, using compiled software as input. The Or1ksim instruction-set simulator is modified to include the TMU functionality, which provides a reference point for the behaviour of the altered processor.

The real-time operating system FreeRTOS is adapted to utilize the functionality of the TMU. Its behaviour is verified through simulation on Or1ksim, simulated hardware and execution on a Cyclone V FPGA.

Analysis of runtime statistics shows that the module is working as expected through all phases of verification, and that it can increase determinism, reliability and user control. Tests have shown that the TMU is able to recover a faulting task from spin-locks and aid in fail-soft operations for software faults. By placing the TMU inside the processor core, a fixed overhead of 131 cycles is achieved during a context switch when no caches are used.

# Sammendrag

Denne oppgaven presenterer en tidsovervåkingsenhet (TMU) som assisterer tidsplanleggeren og avbruddshåndteringen i et sanntids-operativsystem. Enheten har funksjonalitet for å overvåke kjøretiden til prosesser og signalere prosessoren når en prosess bruker opp sine tildelte ressurser. Dette gjelder både vanlige prosesser og sporadiske hendelser. Ved å plassere TMUen inne i en prosessor får man en mindre, og mer forutsigbar innvirkning på tidstillegg under bytting av prosesser.

Den implementerte TMUen har blitt testet som en enkeltstående enhet med en testbenk, og deretter integrert inn i den OpenRISC 1000-baserte OR1200 prosessoren som spesial-register. Oppførselen til OR1200 er verifisert av maskinvare-simulering med kompilert programvare som inngangsstimuli. Instruksjonssett-simulatoren Or1ksim er modifisert til å inkludere TMU-funksjonaliteten, som gir et referansepunkt for oppførselen til den modifiserte prosessoren.

Sanntids-operativsystemet FreeRTOS er tilpasset for å kunne ta i bruk funksjonaliteten til TMUen. Oppførselen til FreeRTOS er verifisert ved bruk av instruksjonssett-simulatoren, maskinvare-simuleringer og kjøring på en FPGA.

Analyse av kjøretidsstatistikk viser at enheten fungerer som forventet gjennom alle faser av verifikasjon, og at den øker determinismen, stabiliteten og brukerens kontroll over systemet. Tester har vist at TMUen gjør det mulig å gjenopprette prosesser som har feilet fra en spinn-lås og assisteres i mykfeil-operasjoner for programvarefeil. Ved å plassere TMUen inne i en prosessorkjerne ble det oppnådd en fast tilleggstid på 131 klokkesykler under et kontekstbytte, når det ikke brukes hurtigminne.

# Preface

This thesis is the final part of our graduate degree at the department of Electronics and Telecommunications at the Norwegian University of Technology and Science.

The work cover many different areas within digital hardware design, computer architecture and real-time systems programming, this thesis is therefore a collaboration between two students. Both of us are studying digital systems design at the department of Electronics and Telecommunications.

We would like to thank our supervisors Amund Skavhaug, who gave us this assignment and provided guidance and advice throughout the work on this project, and Bjørn B. Larsen who has guided us through the work on this project and through writing the final report.

Trondheim, 29th June, 2014

Kyrre Gonsholt

Lars Ødegaard

# Table of Contents

# List of Figures

# List of Tables

# Listings

# Abbreviation

**ALM** Adaptive Logic Module

**CPU** Central Processing Unit

**DTLB** Data TLB

**EEAR** Exception Effective Address Register

**ELF** Executable and Linkable Format

**EPCR** Exception Program Counter Register

**ESR** Exception Status Register

**FIFO** First-In-First-Out

**FPGA** Field Programmable Gate Array

**FPP** Fixed Priority Preemptive

**GPR** General Purpose Register

**HDL** Hardware Descriptive Language

**ISA** Instruction Set Architecture

**ITLB** Instruction TLB

**LED** Light Emitting Diode

**LIFO** Last-In-First-Out

**LRU** Least-Recently Used

**LSU** Load/Store Unit

**LUT** Look-Up Table

**MAC** Multiplier Accumulator

**MMU** Memory Management Unit

**MUX** MultipleXer

**OS** Operating System

**PC** Program Counter

**PC** Process Control Block

**PIC** Programmable Interrupt Controller

**PICSR** PIC Status Register

**PSW** Program Status Word

**PTE** Page Table Entry

**RMS** Rate Monotonic Scheduling

**RSP** Remote Serial Protocol

**RTOS** Real-Time Operating System

**SPR** Special Purpose Register

**TCB** Task Control Block

**TLB** Translation Look-aside Buffer

**TMU** Time Management Unit

**UART** Universal Asynchronous Receiver/Transmitter

**UPR** Unit Present Register

**USB** Universal Serial Bus

**VR** Version Register

**WCET** Worst Case Execution Time

Chapter 1

# Introduction

## 1.1 Motivation

In a real-time system the correct behaviour of a task is dependent on both the logic result of its computation and the time of which the result arrives[1]. Failure to meet a deadline can in some cases have very severe consequences, thus the quality of a real-time system is highly dependent on its timing capabilities.

Scheduling of tasks in real-time systems is usually dependent on the worst-case execution time (WCET) of that task[13], but finding this measure may in some cases be very challenging. The average execution time will often be much smaller then WCET, and by scheduling tasks based on the worst-case situation the processor utilization will be poor. If scheduling is based on the average execution time, deadlines may be lost in cases where a task overruns this time. By using execution time control, less conservative budgets than WCET can be used and the overruns can be handled dynamically. To allow for this option, a unit in the system has to provide information about the elapsed time of the running tasks and interrupt the processor when an overrun has to be handled.

This thesis will present a solution for a unit designed to monitor task execution time called Time Management Unit (TMU) implemented in hardware as a part of an OR1200 processor. The unit will measure the execution time of running tasks and measure the arrival interrupts. By implementing a TMU inside a processor core it is believed that a smaller and more predictable overhead than a bus implementation can be achieved.

## 1.2 Main contributions

The main contributions from this thesis is the design and implementation of a TMU as a part of the OR1200 processor, modifying the OpenRISC ISA simulator Or1ksim to include this TMU and modifying FreeRTOS to take advantage of the functionality provided by the TMU.

Tests are provided to verify the behaviour of the TMU as a stand-alone unit, as a part

of the processor and as a part of Or1ksim. The modified FreeRTOS is tested through Or1ksim, and is then executed on a computer system, containing the modified OR1200, on an FPGA.

The complete system is tested through FreeRTOS on both Or1ksim and on an FPGA. Test were included to verify the behaviour of the TMU as well as demonstrates some of the features it provides.

## 1.3 Outline of the report

The first chapter is an introduction chapter containing information about the report. After the introduction, relevant background theory necessary to understand the report is presented. Chapters 3 to 6 outlines the development of different parts of the system, where:

**Chapter 3** describes the functionality, development and testing of the TMU.
**Chapter 4** describes the integration of the TMU into a larger system, and the testing of this system.
**Chapter 5** describes the operation and modifications to the instruction set simulator for the OpenRISC architecture.
**Chapter 6** describes the operation and modifications done to the real-time operating system FreeRTOS.

Chapter 7 describes the testing of the full system, using the modified processor core and the adapted version of FreeRTOS, it also presents the results from these performed in the different environments. Chapter 8 has a discussion of the implementation and test results of the whole system. The final chapter concludes this thesis, and provides some points for further work on this subject.

Each of the development chapters starts with a short introduction and summary. Throughout the thesis a lot of results and decisions are presented as the report progress, smaller discussions surrounding these will be done where the result or decision is presented. Larger discussions will be made at the end of the chapter.

# Chapter 2

# Background

## 2.1 Operating system

*This section will to give the reader an understanding of the basic principles relied on in the chapter describing FreeRTOS.*

An Operating system(OS) is the collection of software that provides a layer of abstraction between user applications and hardware. It manages resources and provides services like memory management, scheduling and input/output-control. [14, ch.2.1].

### 2.1.1 Real-time operating systems

For real-time systems, the correctness of the results depends not only on the logical result, but also at which time the result arrives [14, p.463]. The degree of how dependant a task is on timing is divided into two categories, hard and soft real-time tasks. For hard real-time tasks the deadline is essential, a missed deadline can have fatal consequences for the system. Soft real-time tasks on the other hand, relates to preferred deadlines, missing one is not severe, but there should still be made an effort to schedule the task within its deadline. Real-time tasks have another characteristic, whether it is periodic or aperiodic. Periodic tasks has a period of time in which they must be scheduled, or scheduled an exact number of units apart. Aperiodic tasks has an absolute time which they must start or finish before. A real-time operating system (RTOS) can be described with having unique requirements in five general areas [14, p.463]

**Determinism**
: Determinism refers to what degree an OS performs operations at specific times, or within time intervall. Like issuing or acknowledging an interrupt.

**Responsiveness**
: Related to determinism, but is rather a measure of how long, from acknowledging, it takes to handle the event.

**User control**
: How much control does the user have over aspects like scheduling and memory.

Normally a standard OS allows for much less user control than an RTOS.

**Reliability**
Errors in a real-time systems can end in degraded functionality and performance, while for a standard OS it might be solved by simply rebooting, which may not be an option for an RTOS.

**Fail-soft operation**
To which degree can the system fail to preserve as much capability and data as possible.

To meet these requirements an RTOS usually include the following [14, p.465]:

- Fast task switch
- Small size
- Low response time for external events
- Multitasking, with synchronization tools like semaphores, signals and events
- Use of special sequential files for fast data storage
- Preemptive scheduling based on priority
- Functionality for task delay
- Special alarms and time-outs

The most important part for an RTOS is the scheduler. Real-time scheduling is explained shortly.

### 2.1.2 Scheduling

In general the term scheduling is used for the division of resources over a period of time[12].
Two main categories of scheduling types are :

**Pre-emptive:**
A task can and will be switched out for a higher prioritized task should one become available.

**Co-operative:**
Tasks are responsible for themselves releasing control of the processor.

#### Scheduler in Real-time operating systems

For real-time scheduling Stallings[14, p.467] lists four approaches:

**Static table-driven:**
Performing scheduling analysis prior to runtime, the resulting table decides the scheduling order.

**Static priority-driven preemptive:**
Performing a scheduling analysis prior to runtime and assigning priorities, which is used to determining execution order during runtime.

**Dynamic planning-based:**
Scheduling analysis is done during runtime, new tasks are only accepted for execution if there is sufficient resources available.

**Dynamic best effort:**
All tasks are accepted, the system tries to meet all deadlines and aborts tasks with missed deadlines.

The different approaches are used in different scheduling policies. The scheduling policy used in FreeRTOS is known as Fixed Priority Preemptive Scheduling(FPP), this falls under the *static priority-driven preemtive*-approach. During implementation the user performs an analysis of which priority each task should have. This is very similar to Rate Monotonic Scheduling(RMS). RMS is basically assigning priorities to task disproportional to their period, low period - high priority [14, p.472].This policy also provides a way of analysing the the schedulability of a set of tasks. This is done by calculating the combined processor utilization of the set of tasks over a period. This relies on good estimates of a tasks execution time and period. It can be shown that the set of $n$ tasks are schedulable within a period if the combined processor utilization is below $n(2^{1/n} - 1)$ over the same period.

One of the most basic scheduling policies is the round robin scheduler. Here processes are given resources in the order at which they arrive, first-in-first-out(FIFO)[14]. Each process will receive a pre-determined amount of time, regardless of the actual runtime needed by the process. When its timing budget is depleted, a clock interrupt will trigger a context switch to the next process in the queue. The switched out process then enters the back of the ready/blocked queue. FPP is very similar to round robin, the difference is that for FPP the task with the highest priority gets control of the CPU next.

### 2.1.3 Processes

The following section is based on chapter 3 in Operating Systems: Internals and Design Principles by William Stallings [14].

A process is defined by Stallings as "The entity that can be assigned to and executed on a processor", among other definitions. This entity is a collection of data and instructions used to describe itself and its behaviour, this may contain the following:

**Identifier:**
A unique identifier for the process.

**State:**
Showing the current state of the process.

**Priority:**
Priority of the process.

**Program Counter:**
The address of the next instruction to be executed.

**Memory pointers:**
Pointers to the location of instructions, data and shared data allocated to the process.

**Context data:**
Data present in the processor's registers during execution.

**I/O Status:**
Includes current I/O requests and accessible I/O devices/files.

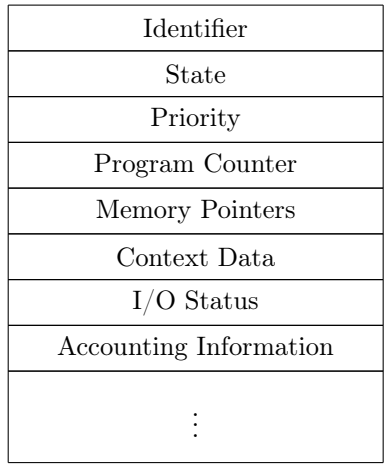



Figure 2.1: Example of Process Control Block

**Accounting Information:**
May include information such as processor time, real-clock time, time limits and other tracing information.

All this information is gathered in a data structure called process control block, Figure 2.1. This block contains enough information about the process to allow it to be interrupted and resumed later as if the interrupt never occurred. This is what allows for multiprocessing systems. When an interrupt arrives, the context data and program counter are saved in their respective memory locations and the state of the process is updated to blocked or ready. At this point it is the operating systems job to load a new process onto the processor.

### Process States

This section is a summary of information relevant to this thesis, for the full explanation of all states and transitions refer to chapter 3.2-4 in the Operating Systems-textbook by William Stallings [14].

During the life cycle of a process it may go through multiple states, new, ready, running, blocked and exit. These states are organized by different lists managed by the operating system. When a process is created it is added to the new-list, once the operating system decides it has sufficient resources available to accommodate an extra process, the process will be added to the ready-list.The different states and what they represent is listed below, Figure 2.2 shows the transitions between the states.

**New:**
Process control block is initialized and the process is ready to enter the main execution loop, *ready-running*.

**Ready:**
The process is ready to enter the execution state, *Running*.
**Running:**
Indicates that the process is currently executing on a processor.
**Blocked:**
The process is waiting on an event such as timing-event, I/O- or file-access
**Suspended:**
The process is placed outside main execution loop, waiting for some event or resources becoming available.
**Exit:**
Indicates that the process is terminated and awaiting cleanup.

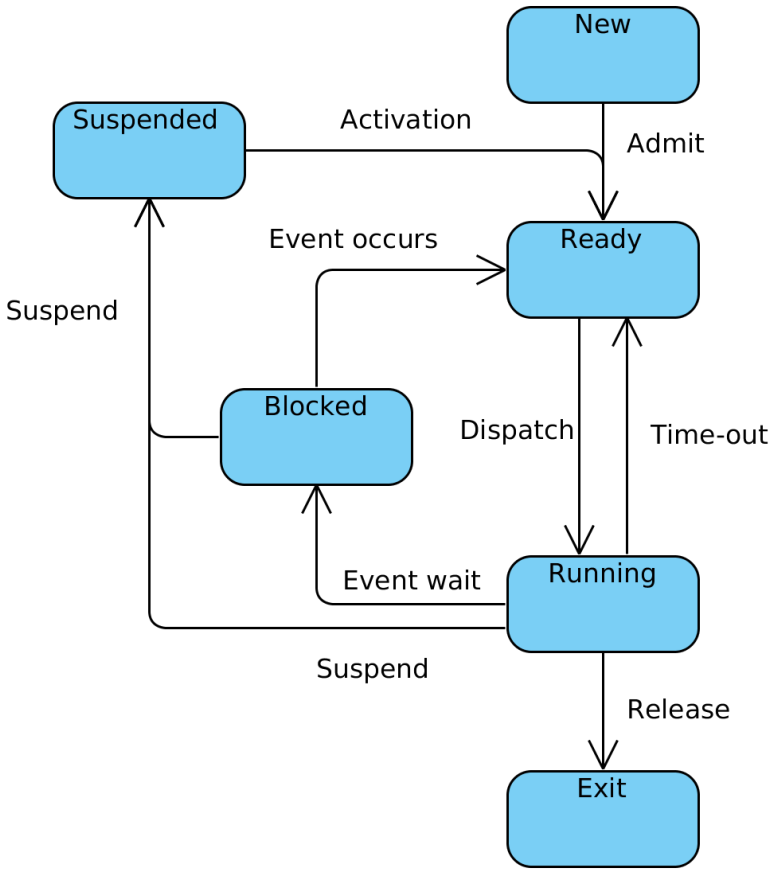



Figure 2.2: Process states transitions, based on [14, Fig.3.9]

### Process Creation

Creating a process can be done through the following steps, but not strictly locked to the following order [14, p.156]:

1. Assign a unique identifier.
2. Allocate memory to hold the new process.
3. Initialize the process control block.
    - Setting the program counter to the start of the program's instructions
    - Set Priorities and resource rights
    - Set stack pointers to the correct memory area
    - Set the process state
4. Add the process to the necessary lists and data structures

### Process Switching

There are multiple mechanics within an operating system that may force a process switch [14, p.157].

**Timing interrupt:**
If there is a time budget in place, the operating system may decide that the currently running process has exceeded its time limit and therefore want to switch process.

**I/O interrupt:**
Upon handling an interrupt, a higher priority task may have been unblocked. After switching back from the interrupt the OS will load the higher prioritized task, assuming pre-emptive kernel and no temporary raised priority levels.

**Process yielding:**
A process can issue commands that directly or indirectly forces a process switch, yielding or issuing a request to another process or I/O-device, thus blocking itself.

When a process switch is initiated an operating system follows these steps to switch out the currently executing process[14, p.159]:

1. Save the current context, including the program counter and other registers.
2. Update the process control block that is currently loaded on the processor. Setting the new state, and explaining the reason for this process to leave the running state and accounting information.
3. Add the process to the correct queue(*ready*/*blocked*/*suspended*/*exit*)
4. Select the next process
5. Update the selected process control block, including setting the state to running.
6. If required, update memory management structures.
7. Restore the context: Setting the program counter and registers to the values from the time this process was switched out of the running state.

### Memory

For the programmer the memory layout of a process can be seen as something like Figure 2.3. The layout in physical memory may differ, depending on the memory placement

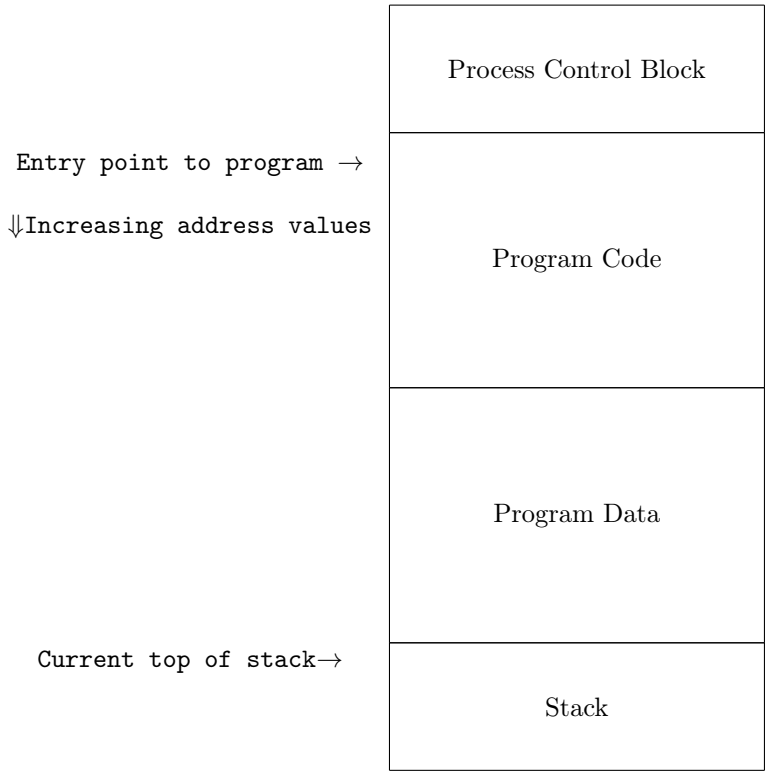



Figure 2.3: Process Control Block in Memory, based on [14, Fig.7.1]

algorithms used [14, ch.7.2-4]. The figure shows the process control block followed by the program code, program data and finally the process stack. The program code contains the binary code for all the instructions in the order of execution. Program data contains all variables and structures. The stack is a first-in-last-out (FILO) data structure which is used to allocate local variables, and grows in respect to function calls and returns [8, p.A-27]. Depending on the architecture a stack may grow upwards or downwards, meaning new elements will have a higher or lower address related to the stackpointer. Local variables are stored with an offset to the stack pointer.

### 2.1.4 Exceptions and interrupts

The terminology exception and interrupt are not used consistently throughout litterature, in this thesis these definitions will be primarily used:

**Exception:**
An internal event inside the CPU.

**Interrupt:**

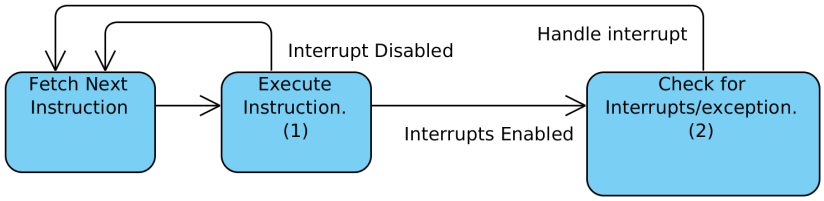



Figure 2.4: S
imple instruction cycle with interrupt]Simple instruction cycle with interrupt, based on [14, Fig.1.7]

An external event which signals the CPU.

Typical interrupts would be environmental events, a UART module signalling that it is done with the transfer. Examples of exceptions are errors in alignment, meaning the CPU is trying to read/write to some address that is not a power of two or suitable for the data size.

Hennessy and Patterson lists these qualities an exception/interrupt can have [8, p.C-44]:

**Synchronous vs. asynchronous:**
If the exception occurs at the same time during execution, assuming the same data and memory layout, then the exception is synchronous, otherwise it is asynchronous. Asynchronous events can be handled after the current instruction, thus they are easier to handle.

**User requested vs. coerced:**
User requested exceptions are predictable in nature. Coerced are exceptions from some module that is not directly under the users control.

**User maskable vs. user nonmaskable:**
If an interrupt can be masked by a user task, it is user maskable. The effect of this is whether or not the hardware will respond to signals from the masked source.

**Within vs. between instructions:**
Wheter or not an exception prevents the completion of the current instruction(1), or is recognized between instructions(2). Figure 2.4 shows a simplified model of the instruction cycle.

**Resume vs. terminate**
If the exception/interrupt causes the current program/process to terminate or allows it to continue after handling the event.

### Handling Exceptions and Interrupts

During the normal execution of a task an interrupt may occur. Figure 2.6 shows Process A in normal execution until an interrupt source requires handling. The interrupt source

does this by e.g. setting a bit in a status register. If interrupts are enabled, the CPU will check this register during the instruction cycle, Figure 2.4. The interrupt raised in Figure 2.6 is registered in step two in the instruction cycle. The interrupt is handled through the following steps [14, p.39], Figure 2.5:

1. Interrupt source signals the processor.
2. The processor finishes the current instruction before responding.
3. The processor checks the status registers to determine if there has been issued an interrupt. It also checks which source generated the interrupt and sends an acknowledgement so that the source can remove its signal.
4. The processor prepares itself to execute the interrupt handler. It starts with saving the information that is needed to resume the currently running process from the point the interrupt occurred. The minimum information required is the program status word (PSW)[1] and program counter. These can be stored on a control stack.
5. The program counter is set to the start of the operating systems interrupt routine, either a specific PC for that specific interrupt handler or a generic handler which in turn may call a specific routine.
6. The instruction cycle continues, fetching the instruction at the new PC.
7. The first instructions in the interrupt routine must save the registers and other essential information about the last process to the stack. The stack pointer is updated and the program counter is set to the code handling the actual interrupt.
8. Depending on the interrupt source this may be a user defined function or a predefined routine inside the operating system.
9. Upon completing the interrupt service, the register values of the last process will be restored from stack.
10. And finally the PSW and program counter is retrieved from the stack. This allows for the interrupted process to start from the point of interrupt as if nothing happened.

An exception is handled very much in the same way, the difference is exceptions usually have specific operating system routines unlike interrupts which usually shares one or more generic routines.

## 2.2 OR1200

The processor core which will host the TMU is the OpenRISC 1000 based OR1200. This processor was chosen because it is an open-sourced and free processor, and it is supported by the GCC compiler. It also has support for programmable interrupts, 32-bit

[1] The PSW contains status information about the current process e.g. status register, memory usage information, condition codes.

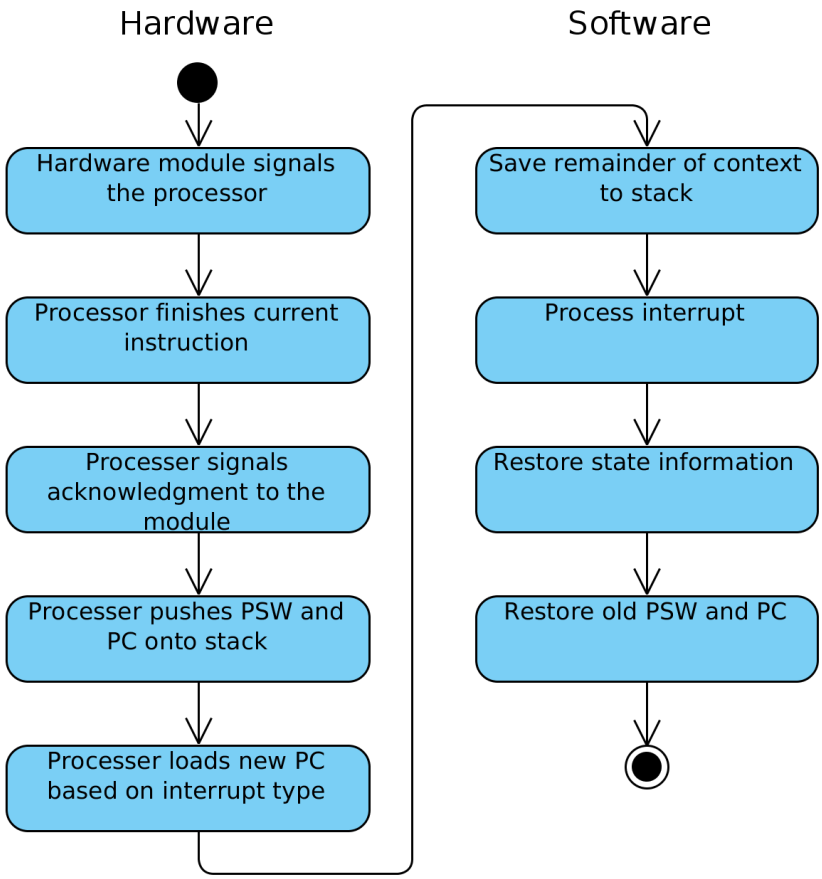



Figure 2.5: Interrupt handling, based on [14, Fig.1.10]

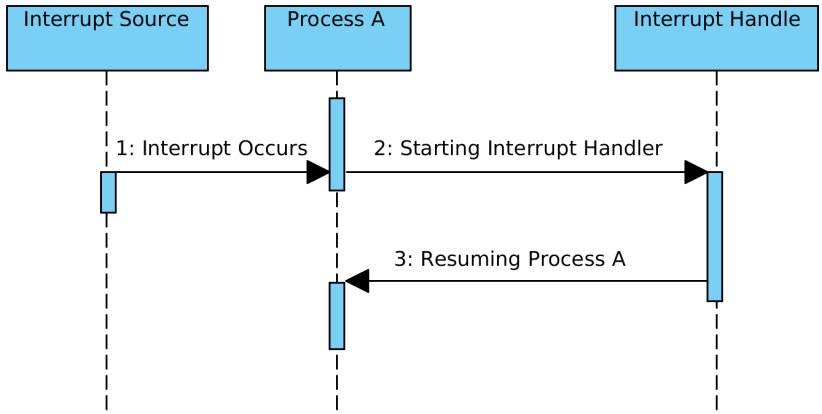



Figure 2.6: Interrupt Sequence

instructions and can be programmed on an FPGA. More details about the choice of the hardware platform can be found in the preliminary study for this thesis[6].

This section is a summary of the most relevant information found in the OpenRISC 1200 IP Core specification[9].

### 2.2.1 Overview

The OR1200 processor is an implementation of the OpenRISC 1000 architecture[9]. It is a 32-bit scalar RISC processor with Harvard microarchitecture, five stage pipeline and virtual memory support. The processor implements the ORBIS32 instruction set.

It has separate Memory Management Units (MMU) for instructions and data, as well as separate instruction and data caches. Both the instruction and data caches are 1-way direct-mapped of 8KB, with 16-byte line size.

The core also contains a power management unit, a tick timer, a programmable interrupt controller and a debug unit. An illustration of the architecture of the OR1200 is shown in Figure 2.7

### 2.2.2 CPU

The central processing unit of the OR1200 contains the following:

- Instruction unit
- Exception unit
- System unit
- General purpose registers
- Integer execution pipeline unit
- Multiplier accumulator (MAC) unit
- Load/Store unit

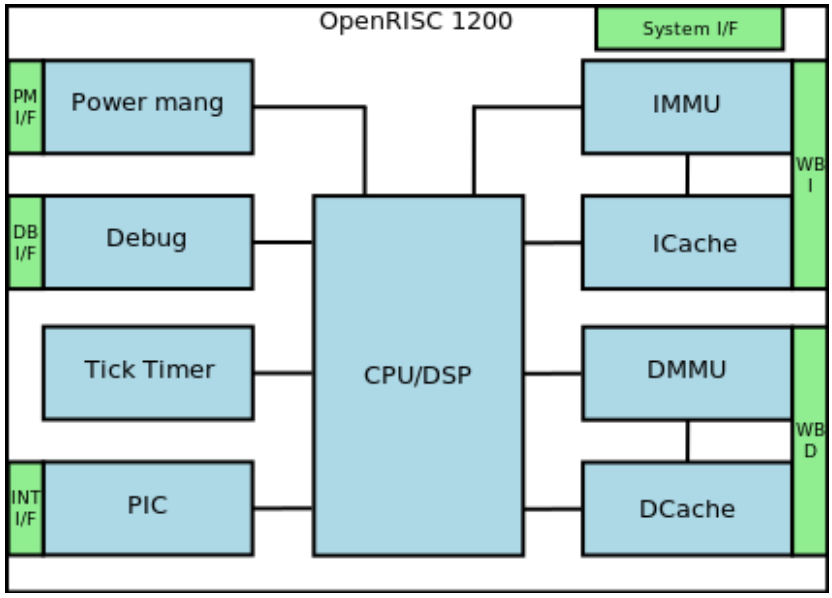



Figure 2.7: OpenRISC 1200 architecture

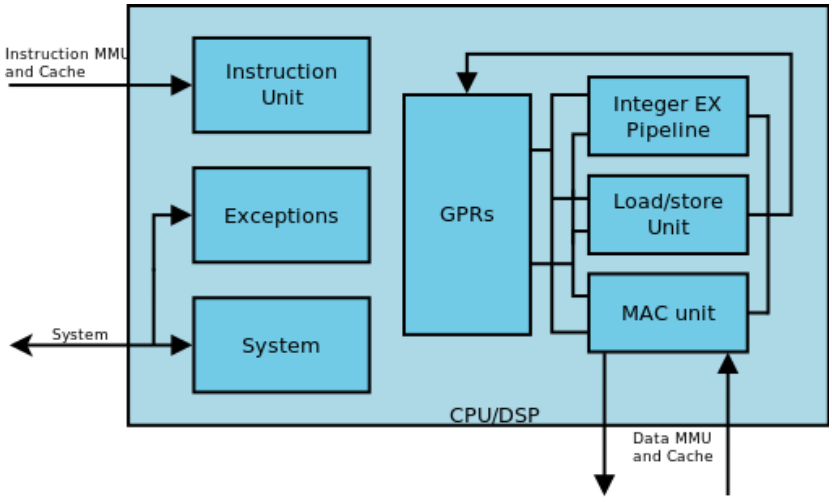



Figure 2.8: OR1200 CPU block diagram

**Instruction unit** The Instruction Unit implements the basic instruction pipeline. It has a line to the instruction MMU and cache units, and is able to execute conditional branch and unconditional jump statements.

**GPR** The general purpose register(GPR) file is implemented as two synchronous dual-port memories. It has a capacity of 32 words, where one word is 32 bit.

**Load/Sore unit** The Load/Store unit is responsible for all data transfers between the GPRs, the data cache and memory. It implements all load/store instructions in hardware, has an address entry buffer and the units operation is pipelined.

**Integer EX** The Integer Execution Pipeline performs 32-bit integer instructions. It handles arithmetic, compare, logical and rotate/shift instructions.

**MAC** The Multiplier Accumulator (MAC) unit is fully pipelined and executes DSP MAC operations that are 32x32 bit with 48-bit accumulator.

**System unit** The system unit is responsible for connecting all signals of the CPU that are not connected through other interfaces. it also contains some special purpose registers.

**Exceptions** The Exceptions unit in the CPU generates core exceptions, i.e. external interrupts and internal errors.

### 2.2.3 Caches and memory management

Since the OR1200 has a Harvard architecture, it has separate caches and MMUs for instructions and data. But as both the instruction- and data-caches and MMUs are implemented in the same manner, the are described only once.

**Caches** The default implementation of the caches are 8KB, 1-way direct mapped cache. The directory is physically addressed and it has a least-recently used (LRU) replacement policy.

**MMUs** The memory management units enables the OR1200 to provide a virtual memory management scheme. Both the instruction and data MMUs are by default implemented as 1-way, direct mapped hash-based translation look-aside buffers (TLB), with a page size of 8KB.

### 2.2.4 Registers

#### SPRs

In OR1200 special purpose registers are divided into 32 different groups. The register address consists of a 5-bit group address and 11-bit register index. A complete list of the registers in OR1200 can be found in [9, table 17]. If accessible, the special purpose

Table 2.1: OR1200 execution times

| Instruction group | Clock Cycles to Execute |
|---|---|
| Arithmetic | 1 |
| Multiply | 3 |
| Compare | 1 |
| Logical | 1 |
| Rotate and Shift | 1 |
| Others | 1 |

registers can be written to with the `l.mfspr`[2] and `l.mtspr`[3] instructions. When reading from a register that is not implemented `l.mfspr` will return zero. Writing to a register that is not implemented will have no effect.

#### GPRs

OR1200 has 32 general-purpose 32-bit registers, labelled R0-R31. Some of these registers have reserved functionality; R0 is fixed to zero, and has to be initialized by software. R1 is the stack pointer, R2 is the frame pointer and R9 is the link register. A complete list of the GPRs is found in the OpenRISC 1000 architecture manual [3, table 16-4].

### 2.2.5 Functional operation

During operation, the **Instruction Unit** will generate the effective address of an instruction and fetch instructions from the instruction cache. The instruction memory management unit then translates this into a physical address. Based on the type of instruction, it is forwarded to the corresponding execution unit; load/store, integer execution or MAC.

Most of the instructions in OR1200 is executed on a single cycle. Table 2.1 shows the executions time for different instruction types.

### 2.2.6 Exception handling

Exceptions in OR1200 are handled precisely, meaning that all instructions up until the start of an exception are valid, instructions in the program flow after the start of an exception are discarded and the causing instruction's address is saved in Exception Program Counter Register (EPCR), the current value of SR is stored in Exception Supervision Register(ESR), if a memory related exception is raised the effective address of the instruction is stored in Exception Effective Address Register(EEAR). A list of all the exceptions handled by OR1200 is shown in Table 2.2.

[2] move from special-purpose register
[3] move to special-purpose register

Table 2.2: Exceptions in OR1200

| **Type** | **Vector offset** | **Pri.** | **Conditions** |
|---|---|---|---|
| Reset | 0x100 | 1 | HW or SW reset |
| Bus error | 0x200 | 4/9 | Attempt to access invalid address |
| Data page fault | 0x300 | 8 | No matching PTE in page table for load/store operations |
| Instruction page fault | 0x400 | 3 | No matching PTE in page table for instruction fetch |
| Tick timer | 0x500 | 12 | Tick timer interrupt asserted |
| Alignment | 0x600 | 6 | Load/store to unaligned location |
| Illegal instruction | 0x700 | 5 | Illegal instruction in the instruction stream |
| External interrupt | 0x800 | 12 | External interrupt asserted |
| DTLB Miss | 0x900 | 7 | No matching entry in DTLB |
| ITLB Miss | 0xa00 | 2 | No matching entry in ITLB |
| Range | 0xb00 | 10 | Caused by certain flags in SR |
| System call | 0xc00 | 7 | System call initiated by SW |
| Floating point | 0xd00 | 11 | Caused by floating point instructions |
| Breakpoint | 0xe00 | 7 | Hardware breakpoint |

Table 2.3: Wishbone signals

| **Common signals** | **Description** |
|---|---|
| clk_i | Clock input |
| rst_i | Reset signal input |
| dat_i | Data in |
| dat_o | Data out |
| tgd_i | Data tag type input |
| tgd_o | Data tag type output |
| **Master signals** | **Description** |
| ack_i | Acknowledge signal from slave |
| cyc_o | Cycle output, indicates a cycle in progress |
| stall_i | Indicates slave is stalled |
| err_i | Slave indicates abnormal cycle termination |
| rty_i | Slave indicates that cycle should be retried |
| sel_o | Slave select signal |
| stb_o | Strobe output |
| we_o | Write enable |

### 2.2.7 External communication

OR1200 uses a Wishbone interface for both data and instructions. Wishbone is an interconnection architecture for portable intellectual property (IP) cores developed by the OpenCores community.

The Wishbone used by OR1200 is shown in Table 2.3, and is taken from the Wishbone specification by OpenCores[11]. Only the signals for the master is shown here, but all connected slaves has a corresponding input/output port.

During operation a read or write through the Wishbone protocol requires three clock cycles[11], and the valid data is placed on the output on the last positive clock edge.

Chapter 3

# Design and implementation of the TMU

*This chapter presents the Time Management Unit (TMU). In the first section the purpose and responsibilities of the unit are described, in the second section some previous implementations are reviewed. Based on the previous work, a specification for the internal unit is devised and some alternative placements are examined. The third and fourth section describes the design requirements and development process of the TMU, and the fifth section presents a testbench for the designed TMU as a stand-alone unit and the results of these tests are shown in the sixth section.*

## 3.1 General description of a TMU

A TMU in a real-time system provides aid in solving two problems. It measures the execution time of a normal task and signal if a task exceeds its budget, hence enable for more optimistic scheduling budgets. And it aids in controlling the time spent handling interrupts.

For counting the execution time of normal tasks, the TMU have to be told when the task starts executing and when it is stopped. It also have to know the time-limit for the running task. If a task then reaches its limit, the TMU will generate an exception so that other tasks can get runtime, and possibly correct the faulting task.

For monitoring time spent handling interrupts, there are two main approaches. Either count the actual time spent handling an interrupt in the same manner as with regular tasks, or count the number of arrived interrupts during a time frame. Both of these have its advantages and disadvantages. If the execution time is counted some overhead will be added for loading, starting and stopping the TMU, which could be critical in the handling of interrupts. One also have to address the problem of which process will be charged for the overhead in masking out the active interrupt. If the number of arrived interrupts are counted, the time it takes to handle a specific interrupt will be unknown if the user does

not know the execution time of the interrupt service routine. But with this approach, counting and masking interrupts can be performed with zero overhead.

## 3.2 Previous work with a TMU

This section present some of the previous work that was studied for this thesis, and provided the base for the implementation of the TMU.

### 3.2.1 On-Line Execution time limiting, 2005

In [13] Håvard Skinnemoen and Amund Skavhaug proposes a hardware counter with nano-second precision in systems with variable CPU clock frequencies. This counter has 64-bits and would act as a wall-clock time-keeper, and by adding an execution time counter one can achieved a deferrable server scheduling approach. The counting of interrupt execution time can be done with one counter or with a separate counter for each interrupt line. The counting of task execution time is done with one counter register, where the counter value is saved and restored as a part of the task context.

In addition the paper specifies a method for avoiding the "babbling idiot problem", by counting the arrival of events instead of execution time for interrupts.

### 3.2.2 A TMU for real-time systems, 2008

The master thesis by Bjørn Forsmann describes the implementation of a TMU for the LEON3 processor as a peripheral bus unit[4]. The thesis also discusses different possible solutions for the placement of the unit, where the alternatives to a bus implementation is either as a coprocessor or inside the processor as a register.

This unit has a count, limit and control register for counting task execution time, and a count, limit, period and control register for each interrupt line for interrupt execution time counting. Forsmann modified the real-time operating system eCos to incorporate the TMU functionality, and values for count and compare are loaded as part of the running context.

### 3.2.3 Functional specification for a TMU, 2010

Kristoffer Gregersen and Amund Skavhaug presented a functional specification for a bus implementation of a TMU in 2010[7]. This paper proposed a simple unit with 64-bit count and compare registers, and swap register. In this specification, the count and compare values are loaded during a context switch or handling interrupts. Hence the implementation of the TMU itself is a simple counter with support for atomic swapping of registers. The specification was tailored for the work being done in implementing the Ada Ravenscar profile for the Atmel AVR architecture.

### 3.2.4 Hardware implementation of a TMU, 2010

In his master thesis Stian Søvik improved the functional specification by Skavhaug and Gregertsen and implemented a TMU as a bus unit inside the Atmel AVR UC3

microcontroller[15]. In addition to the count and compare registers, Søvik introduced a status and control register. The implemented module also adhered to the behaviour of similar modules inside the UC3.

Values for the TMU registers are loaded as a part of the running context, and the overhead added by the TMU is dependent on the availability of the bus. The minimum overhead caused of the TMU during a context switch is claimed to be 28 clock cycles.

## 3.3 Design of the TMU

As described by Bjørn Forsmann in his master thesis, a possible placement for the TMU is inside the CPU[4]. He discarded this as a to intrusive implementation, but mentioned that it could possibly be much faster than a bus implementation. Implementing the TMU internally in the OR1200 CPU core was the focus of this thesis.

### 3.3.1 Possible implementations placements

There are multiple solutions as to where to place the TMU inside the OR1200 core, and how it should communicate with the rest of the processor. Three different positions has been examined, all of which have pros and cons.

**In the register file** The TMU can be placed as a separate module inside the register file module. This would result in some extra addressable registers, but it would be faster because the load/store unit could write values to the TMU directly. The problem with this implementation is the width of the addresses for the registers, which is five bits and able to address 32 registers. To solve this problem at least one bit has to be added to the register operand address-width and some of the custom instructions could be used to load and store the TMU registers. Using custom instructions would also require logic for decoding these instructions.

This solution is highly intrusive and adds the need for a lot of modifications to exiting modules, mainly in the **register file**, **load/store unit**, **instruction decode unit** and **control unit**.

**As a part of the GPRs** Instead of adding the TMU as a separate module, it could be possible to reserve some of the existing general purpose registers to serve as TMU registers. This would remove the need for additional bits in register addresses, but would result in fewer registers available for general use. Depending on the number of interrupt lines, one could end up using all of the general purpose registers to count interrupts. To make sure that these registers are never used for anything other than TMU functionality the compiler tool chain have to be modified.

**As special purpose registers** The TMU could be implemented as a separate module inside the processor and have all registers be addressed as special purpose registers. As compared to the other possibilities, it would be easier to load and store values in the module since there are already several modules that uses the SPR interface inside the

processor. But reading and writing values would have to be done with two instructions, resulting in a larger overhead than a GPR implementation.

### 3.3.2 Chosen design

Of the three alternatives, the chosen solution is the one that utilizes special purpose registers. This alternative was chosen because it minimizes the need to make large modifications to the existing modules and makes it possible to have the TMU as a separate and independent unit. It also makes communication with the module simple, because the existing interface for special purpose registers is used for reading and writing all data. Implementation of the TMU as a separate module corresponds well to the way the rest of OR1200 is written, with a module for a specific functionality[9]. The added overhead of one extra instruction for reading or writing registers in the module, as compared to an implementation directly in the register file, is acceptable.

### 3.3.3 Requirements

The requirements for the TMU implemented for the OR1200 processor is based on the article by Skinnemoen and Skavhaug[13] and Bjørn Forsman's master thesis[4]. The design of the unit should be as simple as possible, but still allow the user full control control over the operation. One of the main problems in counting execution time for both tasks and interrupts is defining the moment the counter should start and stop. The start and stop point for each counter decides which task should be charged with the overhead of switching tasks, or if any task should be charged.

The following requirements are set for the TMU:

- To increase the flexibility of task counting and allow for high clock frequencies, the count and compare registers is set to 64 bits.
- The addition of a TMU in the OR1200 should not change the behaviour of the processor in any way if the user of the system does not specify that the TMU should be used.
- The coding and implementation should be similar to the style used in OR1200, with separate blocks for different synchronous functionality.
- Similar to the tick timer and interrupt exceptions the TMU exception should be active until it is handled, meaning the exception should be cleared by the handler.
- An exception from the TMU should not interfere with the handling of other exceptions.
- The unit should be able to take the processor state into account.

## 3.4 Implementation

This section contains excerpts from the TMU Verilog file, the full code can be found in appendix A

### 3.4.1 Required signals for the TMU

As a part of the special purpose registers in OR1200, the TMU has to conform to the internal SPR interface for reading and writing data. Table 3.1 contains the signals required by this interface. In addition to the SPR signals, the TMU needs inputs for unmasked interrupts and outputs for masked interrupts. The number of interrupt lines can vary from two to 32 lines depending on the processor settings, this must be set prior to compilation. All the system signals for the TMU are listed in Table 3.2.

Table 3.1: Spr signals

| **Signal** | **Direction** | **Width** | **Description** |
|---|---|---|---|
| spr_addr | In | 32 | Register address |
| spr_dat_i | In | 32 | Data to the unit |
| spr_cs | In | 1 | Chip select |
| spr_write | In | 1 | Write data |
| spr_dat_o | Out | 32 | Data from the unit |

Table 3.2: TMU system signals

| **Signal** | **Direction** | **Width** | **Description** |
|---|---|---|---|
| clk | In | 1 | Clock input |
| rst | In | 1 | Reset input |
| intr | Out | 1 | Exception output |
| ex_freeze | In | 1 | Execution freezed input |
| except_started | In | 1 | Exception has started |
| pic_ints | In | OR1200_PIC_INTS | Interrupts input |
| pic_ints_masked | Out | OR1200_PIC_INTS | Masked interrupts output |

### 3.4.2 HDL design

Based on the description of signals in Table 3.1 and 3.2, a high level description of the TMU was constructed. This diagram contains the TMU with connecting signals to the other modules inside the processor, illustrated in Figure 3.1. This figure shows how the lines of the TMU are connected to the necessary modules. The *intr* signal is connected to the exception module, the *spr* signals are connected to the spr module, the *ex_freeze* signal is connected to the freeze module, *pic_ints* originates at the various interrupt sources and *pic_ints_masked* are connected to the programmable interrupt controller.

From this diagram an HDL module was created. Since the OR1200 processor is written in VerilogHDL, this language was chosen for easy integration into the system. A

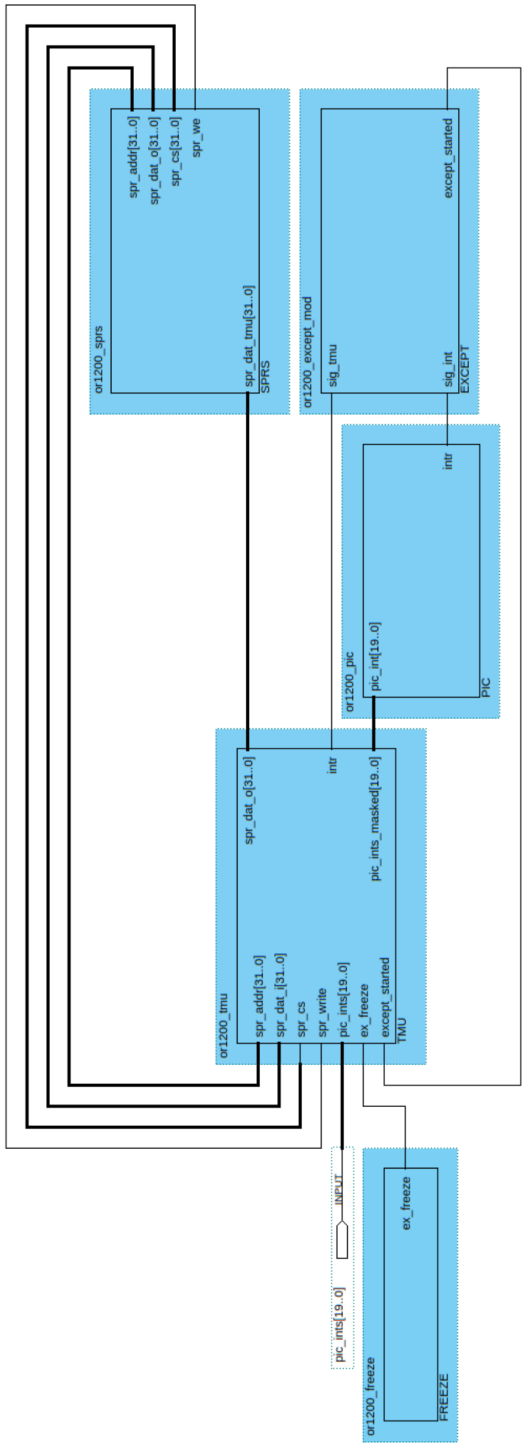



Figure 3.1: Illustration of the TMU and connected modules

VerilogHDL compiler also uses a preprocessor similar to other high level programming languages, which makes parametrization of modules very simple.

The TMU is implemented as a VerilogHDL **module**, which can contain input-/output-ports and synchronous-/asynchronous-logic. The module declaration along with the port definitions of the TMU is shown in Listing 3.1.

Listing 3.1: TMU module declaration

```
module or1200_tmu (
  clk, rst
  , spr_addr, spr_dat_i, spr_cs, spr_write
  , spr_dat_o// SPR interface
  , intr, except_started//Exception interface
  , pic_ints, pic_ints_masked//Interrupt interface
  , ex_freeze
);
input               clk;
input               rst;
input    [31:0]     spr_addr;
input    [31:0]     spr_dat_i;
input               spr_cs;
input               spr_write;
output   [31:0]     spr_dat_o;
input               except_started;
input               ex_freeze;
output              intr;
input    [`OR1200_PIC_INTS-1:0]   pic_ints;
output   [`OR1200_PIC_INTS-1:0]   pic_ints_masked;
```

### 3.4.3 The internals of the TMU

Since all the interaction from the software with the TMU will be through the SPR interface, a register for the current status was needed to provide information about the state of the TMU. This register is read only, since it only displays the current state of the TMU.

To control the operation of the TMU a separate control register was needed, the user can issue commands to the TMU by writing to this register. This register is used to set different modes of operation and to issue general run-time commands. This register was implemented as a read/write register, although reading this will not give any information about the effect of an issued command. Writing to this register will also start and stop the task execution-time counting, enabling the user to specify the start time of task.

#### Configuration parameters

To increase the flexibility of the TMU some configuration parameters was added to set different modes of operation. To adhere to the implementation of other modules it is possible to enable and disable exception generation. This is also included to allow for task execution time counting without producing exception.

If an exception from another module than the TMU arrives during the execution of a task, the designer have to choose if the running task should be charged with the time spent handling this exception. An option to stop the counter when an exception is started

was thus included. This way the user can choose if the running task should be charged for the exception or not.

During normal operation, a CPU will freeze execution during different stages of the pipeline, for instance during load and store. The TMU can take this into account, by stopping the counter during an execution freeze. This will make the count value represent the time which the task has actually been executing instructions.

To provide control over counting and masking interrupts, this feature can be enabled or disabled by the user. If this feature is disabled the TMU should not count or mask any interrupts.

### 3.4.4 TMU register details

The core functionality of the TMU is based on registers which are accessed through the SPR interface. Table 3.3 has an overview of all the accessible registers inside the TMU.

Table 3.3: TMU register list

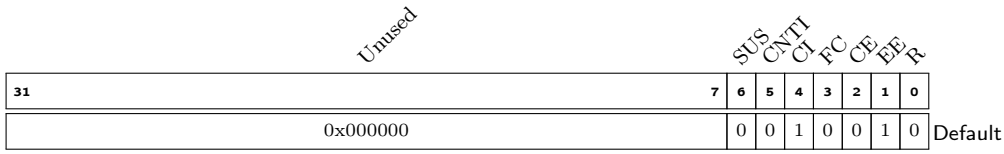
| Register number | Register name | Width | Access |
|---|---|---|---|
| 0 | Status | 32 | R |
| 1 | Control | 32 | W |
| 2 | Compare high | 32 | R/W |
| 3 | Compare low | 32 | R/W |
| 4 | Count high | 32 | R/W |
| 5 | Count low | 32 | R/W |
| 6 | Replenish compare high | 32 | R/W |
| 7 | Replenish compare low | 32 | R/W |
| 8 | Replenish count high | 32 | R |
| 9 | Replenish count low | 32 | R |
| 10 | Masked interrupts | 32 | R |
| 11 | Interrupt compare base | 32 | W |
| 43 | Interrupt count base | 32 | R |

The register number in Table 3.3 also indicates the offset of the address in the TMU SPR address group.

Since the SPR data interface is only 32 bits wide, the high and low values of the 64 bit registers have to be loaded separately. This applies to *count*, *compare* and *replenish compare*.

Register 3.1 lists the different fields of the *status* register. This register is read-only and has a summary of the TMU's status and configuration. Some of the bits in the status register can be controlled by writing to the correct field in the *control* register described in Register 3.2. The default values of the TMU is shown in the lower part of the register description, this indicates which configuration the TMU will have after a reset. By default the TMU is set to generate exceptions, count interrupts and stop counting upon an exception.

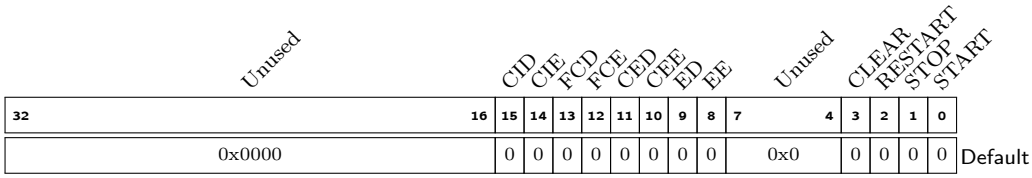
Register 3.1: TMU STATUS REGISTER (32)




| | |
|---|---|
| **R** | Indicates if the TMU is running |
| **EE** | Indicates if TMU exception is enabled |
| **CE** | Indicates if the TMU should continue counting when an exception arrives |
| **FC** | Indicates if the TMU should continue to count when *ex_freeze* is set, added for debug purposes |
| **CI** | Indicates if the TMU should count interrupts |
| **CNTI** | If set the value of count is invalid |
| **SUS** | If the TMU is suspended |

Register 3.2 lists the different fields in the *control* register. The *control* register is used to send commands to the TMU and set different modes of operation.

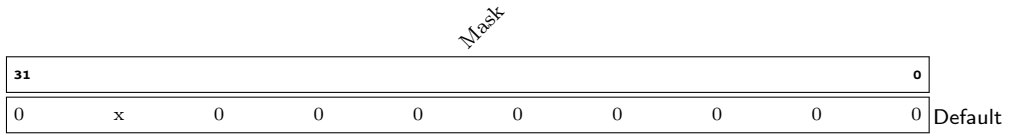
Register 3.2: TMU control register (32)




| | |
|---|---|
| **START** | Start the TMU |
| **STOP** | Stop the TMU |
| **RESTART** | Restart the counter |
| **CLEAR** | Clear count and compare values |
| **EE** | Enable exception generation |
| **ED** | Disable exception generation |
| **CEE** | Enable counting during an exception |
| **CED** | Disable counting during an exception |
| **FCE** | Enable counting during freeze |
| **FCD** | Disable counting during freeze |
| **CIE** | Enable counting of interrupts |
| **CID** | Disable counting of interrupts |

Register 3.3 shows the interrupt mask register of the TMU. When an interrupt line reaches its limit, the bit corresponding to the line number will be set.

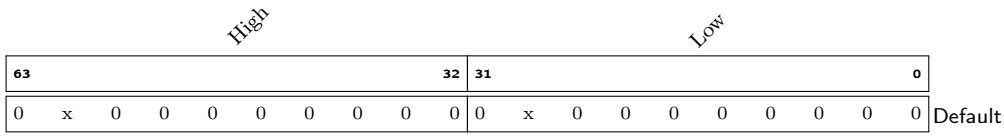
Register 3.3: TMU interrupt mask register (32)




| | |
|---|---|
| **Mask** | Bit $n$ indicates a masked interrupt |

Register 3.4 shows the organization of the 64 bit registers of the TMU, namely *count*, *compare*, *replenish count* and *replenish compare*. On a reset all these registers will be set to zero.

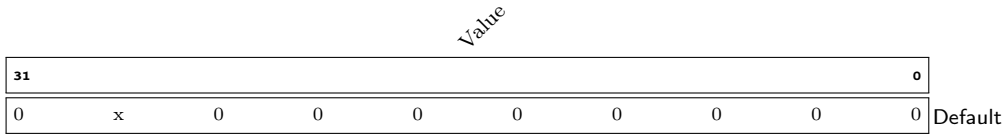
Register 3.4: TMU 64-bit registers (64)




| | |
|---|---|
| **High** | High bits of the register value |
| **Low** | Low bits of the register value |

Register 3.5 shows an overview of the 32 bit count and compare registers of the TMU, which applies to the interrupt counting and compare registers.

Register 3.5: TMU 32-bit registers (32)




| | |
|---|---|
| **Value** | Value of the register |

### 3.4.5 Operation

#### Reading and writing data

Reading and writing data to or from the TMU is done through the SPR interface. Figure 3.2 shows how the SPR signals are decoded inside the module, and how data is read/written.

An excerpt from the Verilog implementation displaying the read and write through the SPR interface is shown in Listing 3.2. This listing only shows the write operation to the *control* register, but all registers are written in the same manner.

The write operation for all registers consists of a synchronous and an asynchronous part. Here the asynchronous part is *ctrl_sel*, which is updated continuously during operation, and the write to the register itself is synchronous. This listing also shows that the *control* register is cleared one clock cycle after it is written because the change in operation is recorded and stored in the *status* register.

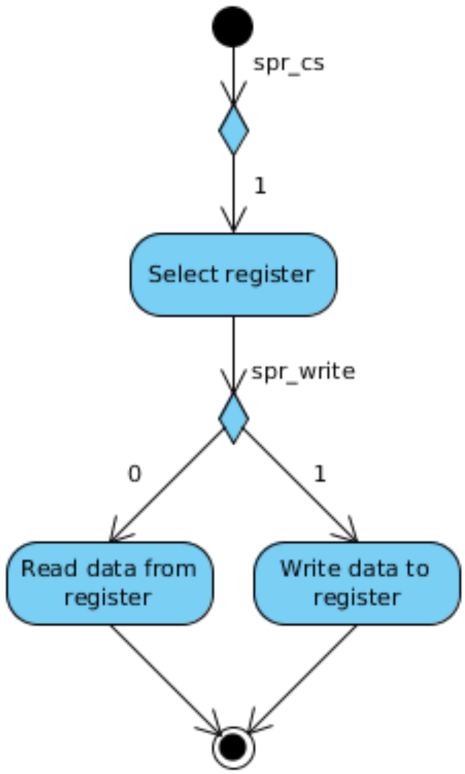



Figure 3.2: SPR operation

Listing 3.2: Read and write SPR

```
assign ctrl_sel    = (spr_cs && (spr_addr[`OR1200_TMUOFS_BITS] ==
    `OR1200_TMU_OFS_CTRL)) ? 1'b1 : 1'b0;
always @(posedge clk or `OR1200_RST_EVENT rst) begin
  if (rst == `OR1200_RST_VALUE) begin
    ctrl <= 32'b0;
  end else if (ctrl_sel && spr_write) begin
    ctrl <= spr_dat_i;
  end else begin
    ctrl = 32'b0;
  end
end

always @(spr_addr or sr or compare or count) begin
  case (spr_addr[`OR1200_TMUOFS_BITS])
    `OR1200_TMU_OFS_STATUS: spr_dat_o = sr;
    `OR1200_TMU_OFS_CTRL: spr_dat_o = ctrl;
    `OR1200_TMU_OFS_COMPARE_HI: spr_dat_o = compare[`OR1200_TMU_HI_BITS];
    `OR1200_TMU_OFS_COMPARE_LO: spr_dat_o = compare[`OR1200_TMU_LO_BITS];
    `OR1200_TMU_OFS_COUNT_HI: spr_dat_o = count[`OR1200_TMU_HI_BITS];
    `OR1200_TMU_OFS_COUNT_LO: spr_dat_o = count[`OR1200_TMU_LO_BITS];
    default: spr_dat_o = sr;
  endcase
end
```

### Configuration

The configuration options of the TMU is listed in Register 3.2, and the TMU is configured by writing to this register. Most of the options enable or disable different features of the TMU, and the writing to control will affect the corresponding bit in the *status* register.

The implementation of enabling and disabling features can be written as a logic function, shown in Table 3.4. This can be simplified to the logic expression found in equation 3.1. Disabling a feature was given precedence over enabling if both are set at the same time. In this equation SR is the current bit in the *status* register, EN is the enable bit in *control*, DIS is the disable bit in *control* and TO_SR is the new value of SR.

Table 3.4: Logic function for TMU features

| SR | EN | DIS | TO_SR |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

$$TO\_SR = EN \cdot \overline{DIS} + SR \cdot \overline{DIS} \tag{3.1}$$

Enabling and disabling features applies to exception generation, counting through exceptions, counting during freeze and counting interrupts, they are all decoded in the same manner. The decoding and setting of the exception generation feature is displayed in Listing 3.3.

Listing 3.3: Setting operation modes

```
wire ee;
always @(posedge clk or `OR1200_RST_EVENT rst) begin
  if (rst == `OR1200_RST_VALUE) begin
    //Set default operation
    sr[`OR1200_TMU_SR_EE] <= 1'b1;
  end else begin
    sr[`OR1200_TMU_SR_EE] <= ee;
  end
end
assign ee = ctrl[`OR1200_TMU_CTRL_EE] & ~ctrl[`OR1200_TMU_CTRL_ED] |
            sr[`OR1200_TMU_SR_EE] & ~ctrl[`OR1200_TMU_CTRL_ED];
```

### Counting task time

One of the two main functions of the TMU is counting the execution time of tasks running on the system and generate an exception if the active task exceeds its budget.

To do this, the TMU needs the current budget, time spent this far and when to start and stop counting. All this information has to be loaded by software through the SPR interface. Figure 3.3 shows how the TMU can be loaded and started. This figure shows what will happen when the different values are loaded into the module, and the counter is started. When the *compare* value is written the *count invalid* bit in the *status* register is set to indicate that the value stored in count might not belong to the current process. This bit is cleared when a new count value is written. To start counting, a logic 1 must be written to the *start* bit in the *control* register. The TMU will then increment the value of count until *stop* is written to the *control* register, or the value of *count* reaches the value of *compare*. To make sure that that the TMU does not generate an exception by mistake, the TMU cannot be started if no value is loaded into *compare*.

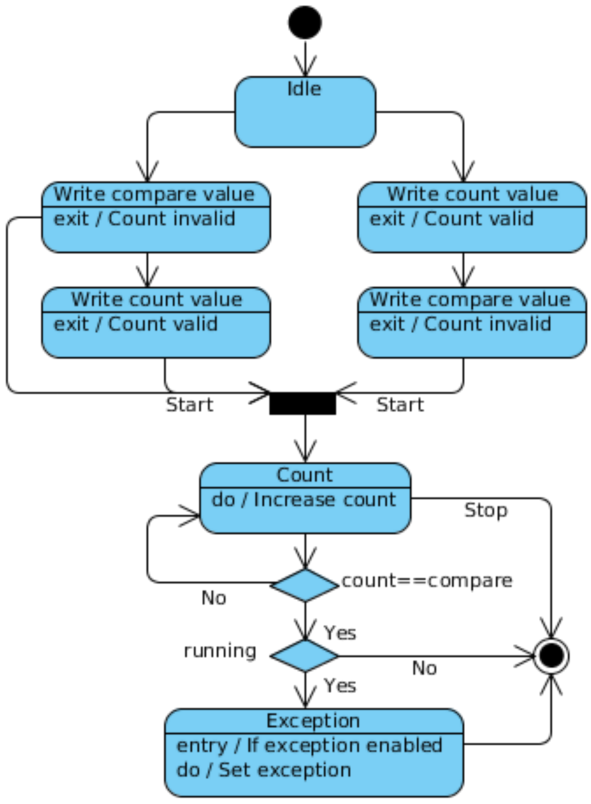



Figure 3.3: Couting task time

The VerilogHDL implementation of *count* and *compare* along with the exception

generation is shown in Listing 3.4. To avoid multiple drivers for *count invalid* when synthesizing the design, writing these two registers are done in the same block. From this listing it is also shown that it is not possible to write a new value to either *count* or *compare* when the TMU is running. This is done to protect the values of these registers.

The *intr* signal is the signal for exceptions from the TMU, and it is implemented in asynchronous logic, and if at any point all the expressions evaluates to true is set to a logic `1`. The TMU will stop the counter when an exception occurs, and the *intr* will be set until the count value is cleared.

The TMU can be restarted by writing to bit 2 in *control*, then the current counter value is set to zero and the counter value will be indicated as valid. If the TMU is cleared by writing to bit 3 in *control*, both *count* and *compare* is set to zero. The counter must be stopped prior to attempting to clear or restart.

Listing 3.4: Implementation of count and compare registers

```
always @(posedge clk or `OR1200_RST_EVENT rst) begin
  if (rst == `OR1200_RST_VALUE) begin
    compare <= 64'b0;
    count <= 64'b0;
    count_invalid <= 1'b0;
  end else if (running && !suspended) begin
    count <= count + 64'b1;
  end else if (count_hi_sel && spr_write && !running) begin
    count[`OR1200_TMU_HI_BITS] <= spr_dat_i;
    count_invalid <= 0;
  end else if (count_lo_sel && spr_write && !running) begin
    count[`OR1200_TMU_LO_BITS]  <= spr_dat_i;
    count_invalid <= 0;
  end else if (clear == 1'b1) begin
    count <= 64'b0;
    compare <= 64'b0;
    count_invalid <= 0;
  end else if (restart == 1'b1) begin
    count <= 64'b0;
    count_invalid <= 0;
  end else if (comp_hi_sel && spr_write && !running) begin
    compare[`OR1200_TMU_HI_BITS] <= spr_dat_i;
    count_invalid <= 1'b1;
  end else if (comp_lo_sel && spr_write && !running) begin
    compare[`OR1200_TMU_LO_BITS] <= spr_dat_i;
    count_invalid <= 1'b1;
  end
end
assign intr = ((count >= compare) && (sr[`OR1200_TMU_SR_EE] == 1'b1) &&
    compare != 0 ) ? 1'b1 : 1'b0;
```

### Counting of interrupts

To aid in the handling of interrupts, the TMU has functionality for limiting the number of interrupts during a specific time period. Instead of counting the time spent handling each interrupt, this implementation counts the arrival of interrupts. This is done to avoid the problem of which interrupt should be charged with the overhead of starting the general interrupt handler. Another reason is to eliminate the need for starting and stopping the

TMU inside an interrupt handler, because adding overhead to these routines could result in failure in some critical systems. The interrupts also share a replenishment budget, because scheduling of tasks is usually done over a limited period of time and this way it is easier to calculate the limit for each interrupt for this period. If interrupt routines are kept small and short, it is easy to calculate the time it would take to execute a specific handler.

The counting of interrupts is done by having a dedicated 32 bit *count* and *compare* register for each available interrupt line, and a a 64 bit *count* and *compare* register for the replenishment budget which is reset automatically when it reaches the *compare* value. Figure 3.4 shows the operation of the replenishment counter.

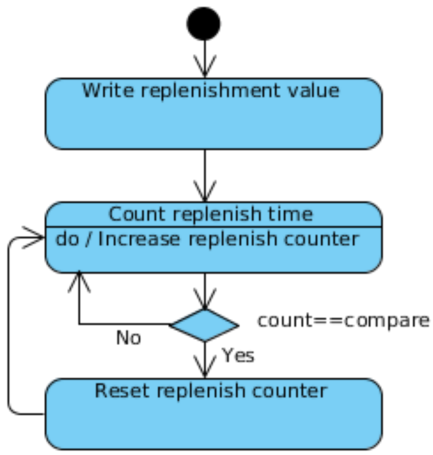



Figure 3.4: Counting interrupt replenishment

The VerilogHDL implementation of the replenishment counter is shown in Listing 3.5. Here the asynchronous part is the replenishment signal, which is set to a logic `1` if *rep_ count* equals *rep_ compare*. When a value is loaded into *rep_ compare* the counter is started, and continues until zero is written back in *rep_ compare*.

Listing 3.5: Implementation of replenishment count

```
assign replenish = (rep_count == rep_compare) ? 1'b1 : 1'b0;
always @(posedge clk or `OR1200_RST_EVENT rst) begin
  if (rst == `OR1200_RST_VALUE) begin
    rep_count <= 64'b0;
  end else if (replenish) begin
    rep_count <= 64'b0;
  end else if (rep_compare != 0) begin
    rep_count <= rep_count + 1;
  end else begin
    rep_count <= 64'b0;
  end
end
```

To start counting interrupts the replenishment counter should be set and the *intr_compare* register corresponding with the interrupt line should be set. If the number of arrived interrupts reaches the limit, the interrupt line is masked until the start of the next period. Figure 3.5 shows the flow of counting and masking an interrupt line.

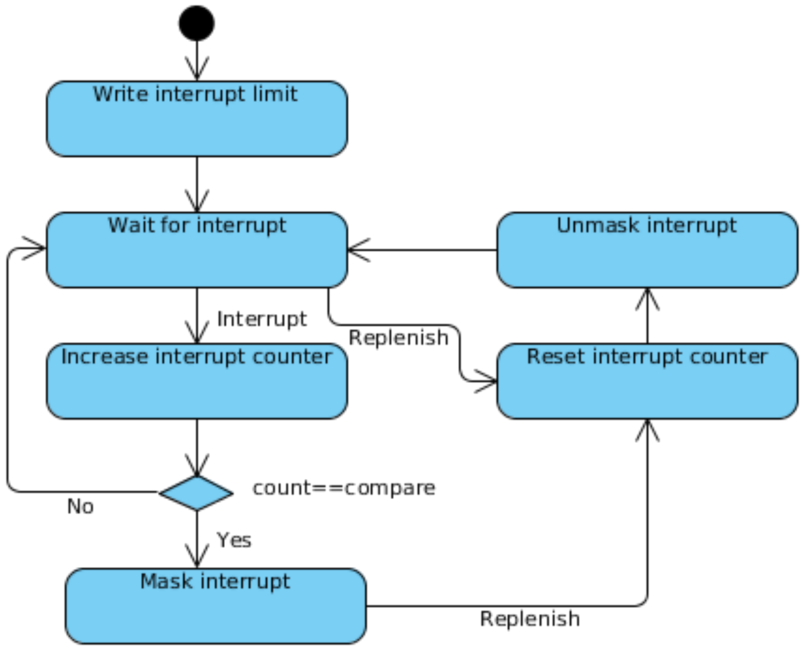



Figure 3.5: Counting interrupts

The number of interrupt counters inside the TMU will vary based on the processor defines, hence the number of interrupt counters will not be known until the processor is compiled. To implement the correct number of interrupt counters the VerilogHDL `for` loop is used. As opposed to a for-loop in a normal programming language, a for-loop in

HDL will generate the required logic for the statement inside the loop. Listing 3.6 shows the implementation of the interrupt counters and masking of interrupts. To ensure each arrived interrupt is only counted once, the counter for each line will only be incremented on a positive edge of the interrupt signal. Decoding the masked interrupts is done by comparing the current value of the interrupt count registers with its limit, and writing a logic `1` to the mask register if the line should be masked as shown in Listing 3.7. The masked interrupts is then placed on the output of the TMU, as shown in Listing 3.7.

Listing 3.6: Implementation of interrupt count

```
reg    [31:0]intr_count   [0:`OR1200_PIC_INTS-1];
integer      pics;
always @(posedge clk or `OR1200_RST_EVENT rst) begin
  if (rst == `OR1200_RST_VALUE) begin
    for (pics = 0; pics < `OR1200_PIC_INTS; pics = pics + 1) begin
      intr_count[pics] <= 32'b0;
    end
  end else if (replenish == 1'b1) begin
    for (pics = 0; pics < `OR1200_PIC_INTS; pics = pics + 1) begin
      intr_count[pics] <= 32'b0;
    end
  end else begin
    for (pics = 0; pics < `OR1200_PIC_INTS; pics = pics + 1) begin
      if (intr_edge[pics] == 1'b1 && sr[`OR1200_TMU_SR_CI]) begin
        intr_count[pics] <= intr_count[pics] + 1;
      end else begin
        intr_count[pics] <= intr_count[pics];
      end
    end
    pics = 0;
  end
end
```

Listing 3.7: Masking interrupts

```
integer mask;
always @(posedge clk or `OR1200_RST_EVENT rst) begin
  if (rst == `OR1200_RST_VALUE) begin
    pic_mask <= `OR1200_PIC_INTS'b0;
    mask <= 0;
  end else begin
    for (mask = 0; mask < `OR1200_PIC_INTS; mask = mask + 1) begin
      if ((intr_count[mask] >= intr_compare[mask])
        && (intr_compare[mask] != 0) && sr[`OR1200_TMU_SR_CI]) begin
        pic_mask[mask] <= 1'b1;
      end else begin
        pic_mask[mask] <= 1'b0;
      end
    end
  end
end
assign pic_ints_masked = pic_ints & ~pic_mask;
```

## 3.5 TMU testbench

To ensure correct behaviour of the TMU, it had to be tested before it could be integrated into the processor. The testbench for the TMU will test all the different operations of the TMU, both one at the time and in parallel. Since the construction of a testbench that tests every possible variation of inputs is not feasible, the testbench for the TMU will check the most common operations and possible errors. Table 3.5 shows all the test performed in this testbench.

The tests are run in the order indicated by Table 3.5, and report the result of each case. If the TMU does not behave correctly on all the tests, it can not be integrated into the processor. Since these tests are meant to aid in the development of the TMU, it will stop when a test fail. Making it easier to examine the simulation data, locate the error, and fixing the problem.

Table 3.5: Tests performed by the TMU testbench

| **Name** | **Test case** | **Excpected result** |
|---|---|---|
| Simple | Write *compare* register and start *count*. | The module should start counting with *count_ invalid* set. An exception should be generated when *count* reaches the *compare* value |
| Simple2 | Write *count* and *compare*, then start counting. The written *count* is not zero. | The module should start counting from the written *count* value, and generate an exception when *count* reaches the *compare* value |
| Zero | Set *count* and *compare* to zero then start the counter | The module should not count or generate an exception |
| No exception | Write *count* and *compare*, disable exception generation and start counting. | The module should not generate any exceptions |
| Read SR | Write *count* and *compare*, start counting and read the status register during operation | The module should continue to count when SR is read, and generate an exception when *count* reaches the *compare* value. The R-bit should be set |
| Test freeze | Write *count* and *compare*, start counting and set the *ex_freeze* signal during operation | The module should continue to count when *ex_freeze* is set, and generate an exception when *count* reaches the *compare* value |
| Freeze stop | Write *count* and *compare*, disable counting during freeze and start counting. Set *ex_freeze* several times during operation | The module should not increment the counter when *ex_freeze* is set, and generate an exception when *count* reaches the *compare* value |

*Continues on next page*

Table 3.5 – *Continued from previous page*

| **Test name** | **Test case** | **Excpected result** |
|---|---|---|
| Test reset | Write *count* and *compare* and start counting. Reset the TMU during operation | The module should not continue after reset and all registers should be set to the default value |
| Start stop | Write *count* and *compare* and start counting. Stop counting after some time, and then continue counting | The module should continue counting from the value it had when it was stopped and generate an exception when *count* reaches the *compare* value |
| Read count | Write *count* and *compare* and start counting. Stop counting after some time, and read the *count* register | The output values from SPR should match the value of *count* when the module is stopped |
| Random | Apply random stimuli to *spr_addr*, *spr_write* and *spr_data* | Nothing should happen |
| Count random | Write *count* and *compare* and start counting. Apply random stimuli to *spr_addr*, *spr_write* and *spr_data* during operation | The module should generate an exception when *count* reaches the *compare* value |
| Exception count | Write *compare* and *count* register then start counting. During operation, set *except_started* | The module should continue counting |
| No exception count | Write *compare* and *count* register, and disable the count during exceptions then start counting. During operation, set *except_started* | The module should stop counting when *except_started* is set |
| Replenish | Write replenishment *compare* | Module should start incrementing *rep_count* and restart the timer when *rep_count* reaches the *rep_compare* value |
| No masking | Write replenish and an interrupt *compare* register, set the interrupts fewer times than the limit | The interrupt should not be masked and counter should reset at the replenishment limit |
| Masking | Write replenish and an interrupt *compare* register. Signal more interrupts than the limit | The interrupt should be masked at limit and unmasked at the replenishment limit |

*Continues on next page*

Table 3.5 – *Continued from previous page*

| **Test name** | **Test case** | **Excpected result** |
|---|---|---|
| Disable masking | Write values into replenish, all the interrupt *compare* registers, and disable the interrupt counters. Apply random stimuli on the interrupt lines | The TMU should not count or mask any interrupts |
| Random masking | Write values into replenish and all the interrupt *compare* registers. Apply random stimuli on the interrupt input lines | The TMU should only mask the interrupts that reaches its limit, and unmask all interrupts at replenishment |
| Full | Write values to replenishment and all the interrupt *compare* registers. Load values into *count* and *compare* and start the TMU. Apply random stimuli on the interrupt lines. | The TMU should count execution time and mask interrupts that reaches its limit. When *count* reaches *compare* an exception should be generated. |

## 3.6 Results of TMU testbench

A simulation of the TMU testbench shows that it behaves according to the specifications and hence passes all tests described in Table 3.5. Since the output of the testbench is to large to include in the report, only parts of the full test is presented here, the rest can be found in appendix B in *tmu-bench*. This section displays some of the key events of the full TMU test from the testbench. For each event the figures are limited to the relevant signals and time frame. The full waveform of this test can be found in appendix B in *tmu-bench/tmu-full.fst*.

### Loading the replenishment counter

The full test first loads values into the replenishment register. Loading and starting the counter is shown in Figure 3.6. In this figure the hexadecimal value 0x20 is loaded into compare, and the counter is started on the next positive edge of the clock. The points marked in the figure are:

1. Replenishment value is loaded
2. Replenishment counter starts

### Masking interrupts

When an interrupt reaches its limit, it is masked until the beginning of the next replenishment period. Figure 3.7 shows that interrupt line 18 is masked the sixth time it arrives. In this figure *pic_ints* is the input line to the TMU and *pic_ints_masked* is the output. The *intr_edge* indicates the arrival of an interrupt, and *pic_mask* indicates if the interrupt is masked. The point marked in the figure is:

1. Interrupt on line 18 is masked

### Replenishment of the interrupt counter

During operation the interrupt counter budget is reset when the replenishment counter reaches its limit, in this case 0x20. Figure 3.8 shows that the counter is reset when the replenishment limit is reached, and that the interrupt mask is cleared, enabling all the interrupts. The points marked in the figure are:

1. The replenishment counter reaches its limit
2. All interrupts are unmasked

### Loading task counter values

Loading values into *compare* and *count*, and then starting the TMU is displayed in Figure 3.9. Both the 64-bit values are loaded in two operations, with the high value first. The hexadecimal value 0x10 is loaded into *count*, and the TMU continues to count from this value when it is started. The points marked in the figure are:

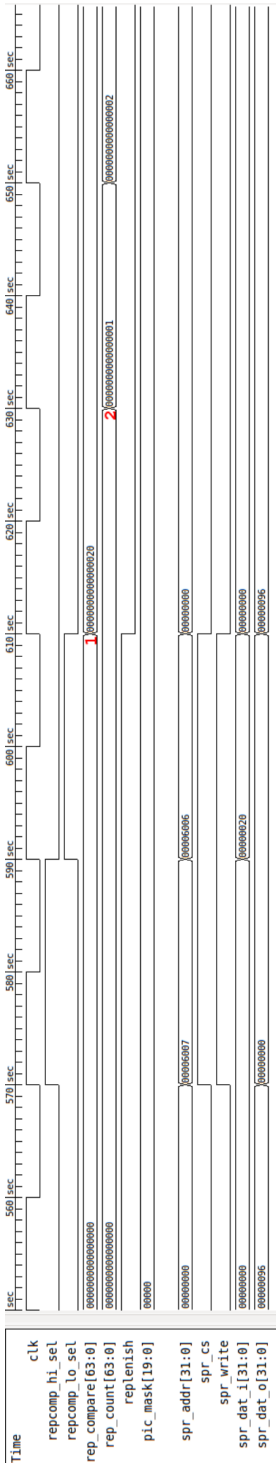



Figure 3.6: Loading a value into replenishment register

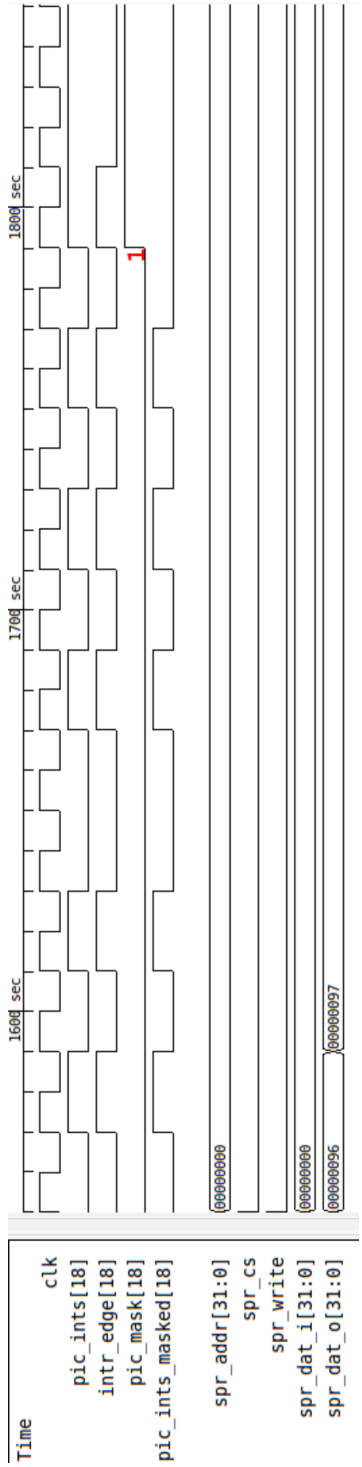



Figure 3.7: Masking an interrupt

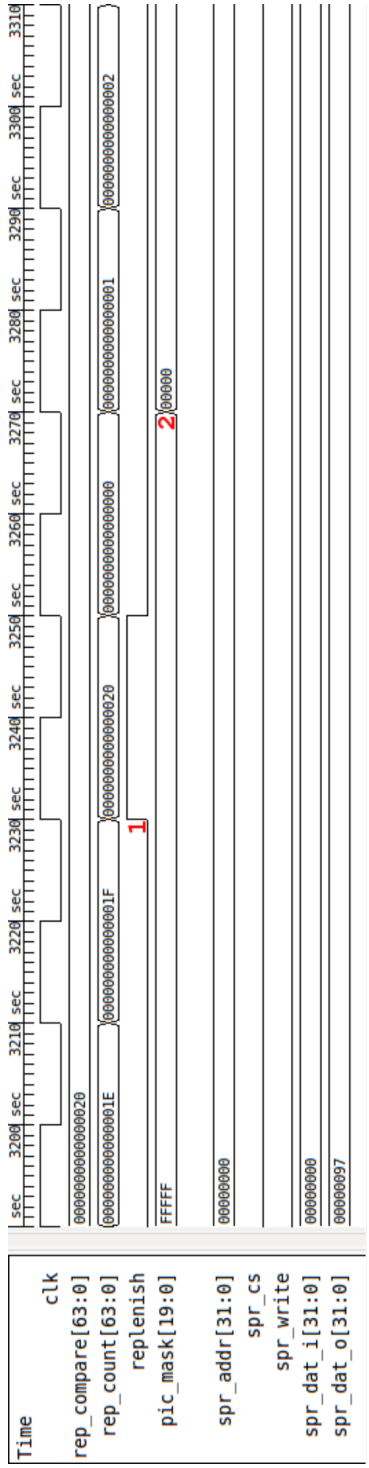



Figure 3.8: Replenishment of the interrupt counter

1. High part of *compare* is selected
2. Low part of *compare* is selected
3. High part of *count* is selected
4. Low part of *count* is selected
5. *Control* is selected

**Exception generation**

When the counter value reached the *compare* value, an exception was generated and the counter was stopped. This is shown in Figure 3.10, where the signal *intr* is the output from the TMU to the exception module. The exception was cleared by writing zero to the *count* register. The points marked in the figure are:

1. Exception signal is asserted
2. Count value is cleared

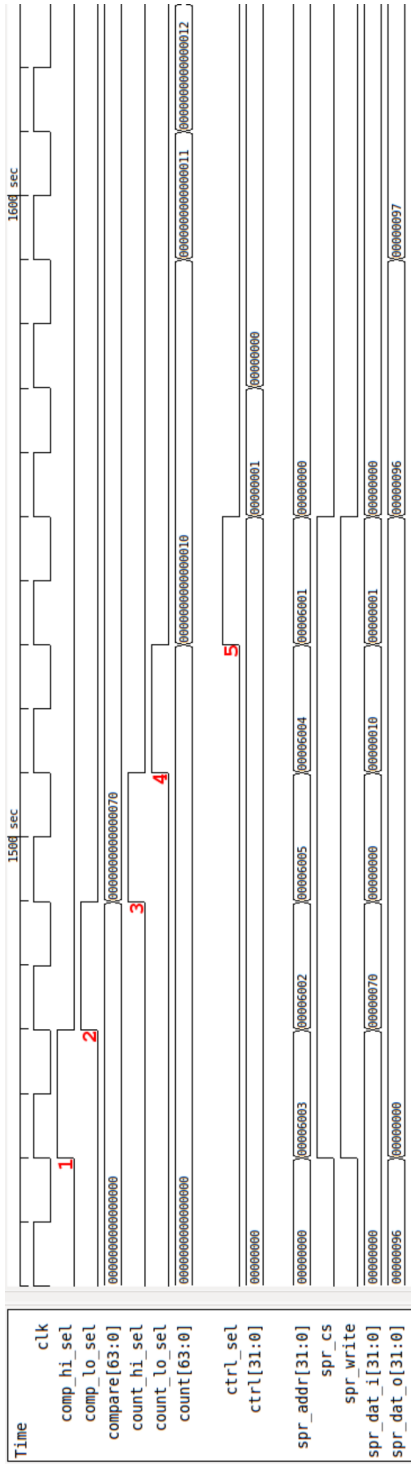



Figure 3.9: Loading values for task execution time counting

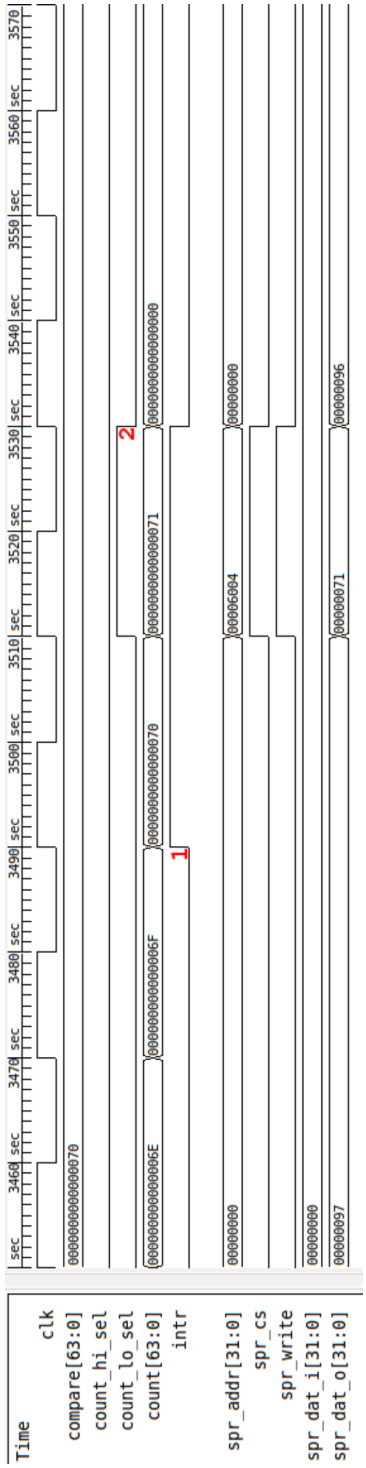



Figure 3.10: Generating the exception

### 3.6.1 Discussion of the results

By studying the results and waveforms of the testbench, the TMU seems to behave according to the specification. Although there are many case variants and different values that have not been tested, the correct execution of the tested cases proves the correctness of the module well enough for testing as a part of the entire system. To ensure that all synchronous logic functions properly and does not generate latches, there is a delay of one clock cycle from when an event occurs to the time the reaction arrives. This is regarded to be acceptable and preferred over using only asynchronous logic in the TMU, which could eliminate this delay.

Chapter 4

# Integrating the TMU in OR1200

*This chapter presents the process of integrating the TMU into the OR1200 processor. The first section is a description of the necessary changes to the existing modules of the processor, and the instantiation of the TMU module. The second section describes the set-up of the additional modules of the system used for testing the functionality. The last sections presents the results of the system tests with and without the TMU as well as the result of the tests designed to verify the behaviour of the TMU as part of the system.*

## 4.1 Integration into the OR1200 processor

After the behaviour of the TMU was verified, it could be integrated into the OR1200 processor. To do this, several of the modules of the OR1200 had to be modified. This section goes through the modifications necessary in order to integrate the TMU into the processor.

### 4.1.1 Modifications to the SPRS module

The SPRS module **or1200_sprs** needed an input line for data from the TMU, and logic to pass this data on to other modules. All the logic for writing to the TMU SPR group was already in place. To handle reading from the TMU the TMU SPR group was added to forward data from the TMU to the internal MUX. A bit in the supervision register(SR) was added to enable and disable the handling of the TMU exception in the same manner as with tick-timer and interrupts. The SPRS module needed logic to write to the bit in the supervision register that is used to enable and disable handling of the TMU exception. This bit can be set by a write to the supervision register via the *l.mtspr* instruction, automatically set to zero when an exception is started or restored from exception status register during a return-from-exception instruction.

Register 4.1 shows the new outline of the supervision register, the width of the part of this register that is used inside the processor had to be increased with one bit. The logic for writing to the TMU exceptions bit in SR is shown in Listing 4.1

Listing 4.1: Writing TMU exception bit in SR

```
assign to_sr[`OR1200_SR_CEE]
     = (except_started) ? {1'b0} :
       (branch_op == `OR1200_BRANCHOP_RFE) ?
        esr[`OR1200_SR_CEE] : (spr_we && sr_sel) ?
        spr_dat_o[`OR1200_SR_CEE] :
        sr[`OR1200_SR_CEE];
```

Register 4.1: OR1200 MODIFIED SR (32)

| CID | | | Reserved | | CEE | SUMRA | FO | EPH | DSX | OVE | OV | CY | F | CE | LEE | IME | DME | ICE | DCE | IEE | TEE | SM | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | | 28 | 28 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0 | x | 0 | 0 | x X | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | Default |

| | |
|---|---|
| **SM** | Supervision mode |
| **TEE** | Tick Timer Exception Enable |
| **IEE** | Interrupt Exception Enable |
| **DCE** | Data Cache Enable |
| **ICE** | Instruction Cache Enable |
| **DME** | Data MMU Enable |
| **IME** | Instruction MMU Enable |
| **LEE** | Little Endian Enable |
| **CE** | CID Enable |
| **F** | Conditional branch flag |
| **CY** | Carry flag |
| **OV** | Overflow flag |
| **OVE** | Overflow flag Exception |
| **DSX** | Delay Slot Exception |
| **EPH** | Exception prefix high |
| **FO** | Fixed One |
| **SUMRA** | SPRS User Mode Access |
| **CEE** | TMU Exception Enable |
| **CID** | Context ID, not implemented |

### 4.1.2 Modifications to exception module

The exception module **or1200_except** needed an input line for the TMU exception signal, and logic to process it. The signals *except_trig* and *except_stop* needed one additional bit each to indicate if the TMU exception is triggered or stopped. The order in which the TMU signal appear in these signals also indicates the priority of the TMU exception. The TMU signal was added to the end of *except_trig* and the beginning of *except_stop*, giving it the lowest priority.

To be able to load and store the bit in the supervision register associated with the TMU exception. And to ensure that handling the TMU exception is enabled at the right time, some logic was added to decode and delay the enabling of this exception. This is done in the same manner as with the interrupt- and tick-exceptions. The conditions for when a TMU exception will be treated as pending to be executed is shown in Listing 4.2. This means the exception will be handled under any of these circumstances:

- CEE bit in the supervision register is set
- CEE bit will be set in the next clock cycle
- Identified program counter is valid
- Delay from *l.rfe* is completed
- Processor is not freezed
- Exception branch is not taken
- Exception is not delayed
- Exception will not be disable in the next clock cycle

Listing 4.2: Decoding TMU exception

```
assign tmu_pending = sig_tmu
            & (sr[`OR1200_SR_CEE] | (sr_we & to_sr[`OR1200_SR_CEE]) )
            & id_pc_val & delayed_cee[2] & ~ex_freeze
            & ~ex_branch_taken & ~ex_dslot
            & ~(sr_we & ~to_sr[`OR1200_SR_CEE]);
```

The main decoding of exceptions is done in the FSM of the exception module. In the idle state a switch-case for the TMU exception was added to set the correct exception type and Exception Program Counter Register (EPCR) shown in Listing 4.3. The exception type indicates which memory address is loaded when an exception starts, and the TMU was set to be exception 0x0F, which was the first unimplemented exception vector of the OR1200. Placing the TMU at 0x0F also eliminates the need to increase the width of *except_type* and increasing the reserved memory for exception vectors.

Listing 4.3: Decoding TMU exception

```
`ifdef OR1200_EXCEPT_TMU
        15'b000_0000_0000_0001: begin
          except_type <= `OR1200_EXCEPT_TMU;
          epcr  <= id_pc;
          dsx <= ex_dslot;
        end
`endif
```

### 4.1.3 Modifications to the configuration module

The configuration module **or1200_cfgr** is responsible for setting the correct values of the version- (VR), unit present- (UPR) and configuration registers. The TMU position in UPR was set to the first available bit, and the configuration module was set to write this into the UPR at startup. Register 4.2 shows the new outline of the UPR.

Register 4.2: OR1200 MODIFIED UPR (32)

| Custom units | | | Reserved | | | TMUP | FPP | TTP | PICP | PMP | PCUP | DUP | MP | IMP | DMP | ICP | DCP | UP | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 31 | | 24 | 23 | | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 0 | x | X | 0 | x | X | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | Default |

| | |
|---|---|
| **UP** | UPR present |
| **DCP** | Data cache present |
| **ICP** | Instruction cache present |
| **DMP** | Data MMU present |
| **IMP** | Instruction MMU present |
| **MP** | MAC present |
| **DUP** | Debug unit present |
| **PCUP** | Preformance counters present |
| **PMP** | Power management present |
| **PICP** | PIC present |
| **TTP** | Tick timer present |
| **FPP** | Floating point unit present |
| **TMUP** | TMU present |

### 4.1.4 Modifications to top level modules

The central processing unit's top level module **or1200_cpu** will instantiate the TMU and needed wires for all TMU signals. Inputs for interrupts and output for the masked interrupts also had to be added to the module declaration. Listing 4.4 show the necessary additions to the CPU top module to make use of the TMU.

Listing 4.4: Additionds to **0r1200_cpu**

```
//Reroute of interrupt lines
input [`OR1200_PIC_INTS-1:0] pic_ints;
output [`OR1200_PIC_INTS-1:0] pic_ints_masked;
wire [31:0] spr_dat_tmu;
wire [14:0] except_trig;
wire [14:0] except_stop;
wire except_tmu;
or1200_tmu or1200_tmu (
  .clk(clk),
  .rst(rst),

  .spr_addr(spr_addr),
  .spr_dat_i(spr_dat_cpu),
  .spr_cs(spr_cs[`OR1200_SPR_GROUP_TMU]),
  .spr_write(spr_we),
  .spr_dat_o(spr_dat_tmu),

  //exception interface
  .intr(except_tmu),
  .except_started(except_started),

  //interrupt interface
  .pic_ints(pic_ints),
  .pic_ints_masked(pic_ints_masked),

  .ex_freeze(ex_freeze)
);
```

The OR1200 top level module **or1200_top** only had to reroute the interrupts signals from the input via the TMU to the interrupt controller.

### 4.1.5 Additions to the configuration file

The file **or1200_defines.v** contains the parameters used by all the modules in the processor. The defines needed by the TMU was placed here, and the parameters that had to be changed in order to include the TMU were modified. Table 4.1 has overview of all the parameters affected by the adding of the TMU and new parameters needed by other modules than the TMU itself.

## 4.2 Setting up the complete system

To be able to test and use the processor, it needed support from some other modules. Much of the work on developing the system used in this thesis was done as a separate project found in [6]. The system setup remains quite similar except for some modifications. In this thesis the instruction and data bus is divided and controlled by separate arbiters, and support for simulated memory is added. Two interrupt generators was connected to the data-bus, for the testing of interrupt masking. Figure 4.1 shows how the system is structured.

All the components used in this system were developed by the OpenCores community,

Table 4.1: Paramters for the TMU

| Name | Old Value | New value |
|---|---|---|
| OR1200_SR_WIDTH | 16 | 17 |
| OR1200_SR_CEE | - | 17 |
| OR1200_UPR_TMUP_BITS | - | 12 |
| OR1200_UPR_RES1_BITS | 23:12 | 32:13 |
| OR1200_UPR_TMUP | - | 1 |
| OR1200_DU_DSR_TMU | - | 14 |
| OR1200_DU_DRR_TMU | - | 14 |
| OR1200_EXCEPT_TMU | - | OR1200_EXCEPT_WIDTH'hf |
| OR1200_SPR_GROUP_TMU | - | 12 |

and can be downloaded for free from the internet. The complete system can also be found in appendix B in *hardware/orsoc*

## 4.3 Setting up simulation

Throughout the work of this thesis open-source tools for compiling and simulating hardware was used, Icarus Verilog for compilation and VVP for simulation. Because the simulated hardware is a computer system, actual software was used as input to the simulation. This meant that a simulation script had to handle both the compilation of hardware and software for OpenRISC.

This was done with a makefile which compiled the processor to an executable simulation model, compiled and converted the software to hexadecimal instructions and ran the simulation. The input program was loaded by the simulated memory when the simulation started. Figure 4.2 shows the steps in compiling and running the simulation. It is possible to specify a single test or run all the available test consecutively. The makefile used in this thesis is based on the simulation makefile from OrpSoC[1], with some modifications. The support for other compilers than Icarus Verilog is removed, the conversion for user specified ELF files to hexadecimal files is included and some rules for compiling and simulating FreeRTOS are added. The full description of how to set up and use the simulation framework can be found in Appendix C.

Instead of compiling one of the software tests, a pre-compiled ELF file can be specified. This file is then converted to hexadecimal format by the script and used as input in the simulation.

The output from the simulation was stored in a separate folder, and an execution log containing the completed instructions and the content of all the general purpose registers was created for each test. A waveform file for each test is generated only if it is specified.

[1]OrpSoC is a OR1200 based system on chip, developed by the OpenCore community

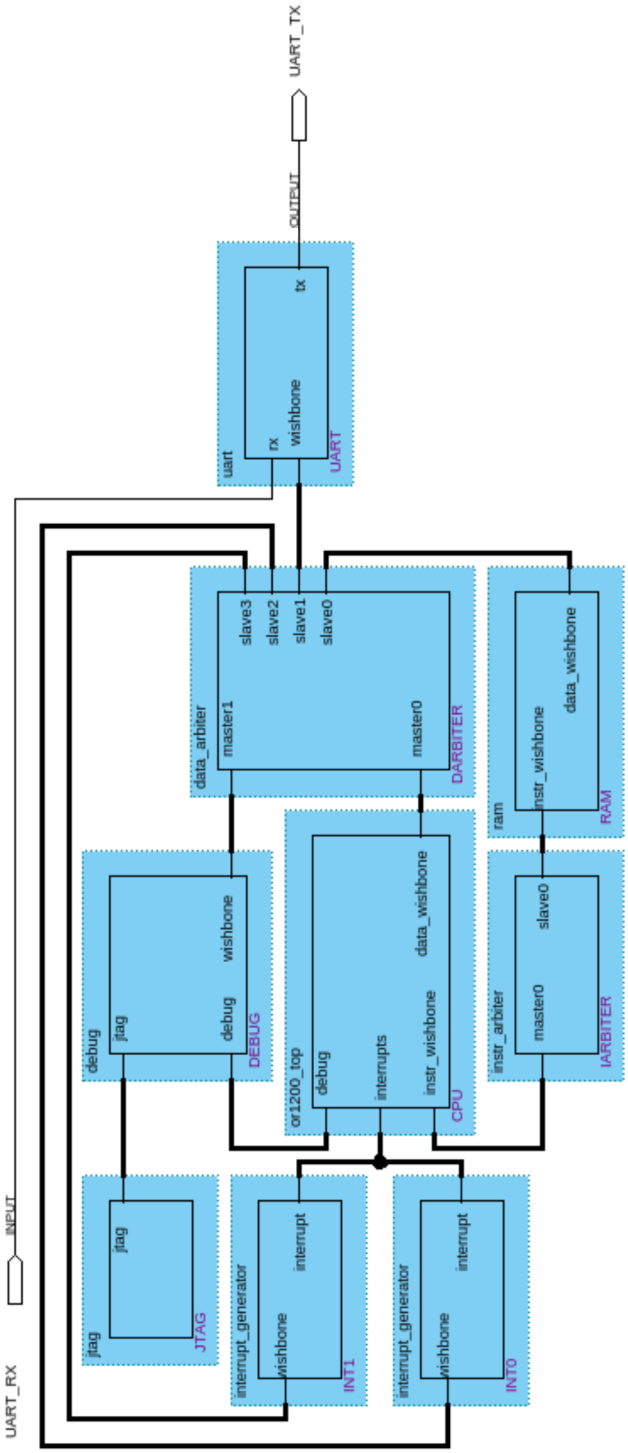



Figure 4.1: The complete system

## 4.4 OR1200 tests

To verify the behaviour of the OR1200 processor and the connected components in the system, the tests from OrpSoC was performed. Table 4.2 shows an overview of the standard tests performed by the OrpSoC testbench. The tests for OR1200 were written in either C or assembly and was compiled and loaded by the memory module during simulation.

Table 4.2: Tests performed by the SoC testbench

| Name | Test case | Language |
|---|---|---|
| or1200-simple | Check if the main loop is reached and the exit mechanism is functioning | C |
| or1200-basic | Basic test of the instruction set | ASM |
| or1200-cbasic | Test the basic C functionality | C |
| or1200-dctest | Test the data cache | C |
| or1200-float | Test the floating point unit in hardware or software | C |
| or1200-mmu | Test the MMU | C |
| or1200-except | Test handling of exceptions, reset, bus error, alignment, illegal, d-tlb miss. | ASM |
| or1200-mac | Test the MAC unit | ASM |
| or1200-ext | Test zero and sign extension instructions | ASM |
| or1200-cy | Test setting of carry bit | ASM |
| or1200-ov | Test setting the overflow bit | ASM |
| or1200-sf | Test the comparison instructions | ASM |
| or1200-dsx | Test delay-slot exception bit generation | ASM |
| or1200-dsxinsn | Test delay-slot exception bit on instruction fetch related instructions | ASM |
| or1200-ffl1 | Test the find first and find last `1` instructions | ASM |
| or1200-linkregtest | Test behaviour of link register in jump-and-link instructions | ASM |
| or1200-tick | Test the tick timer | ASM |
| or1200-ticksyscall | Test the tick timer and syscall exceptions simultaneous | ASM |
| uart-simple | Test the UART by printing a string | C |

When the TMU was added some additional tests were added, the tests written to test the TMU functionality is shown in Table 4.3. To test the masking of interrupts two simple interrupt generators were added to the system via the data bus. *or1200-test* in

Table 4.3 will run the following tests from Table 3.5:

- Simple
- Simple2
- Read SR
- Start stop
- Read count

The test counts the number of generated interrupts and reports if the test is successful.

*intgen-simple* will test masking of interrupts by loading a value into the replenishment compare register and the two interrupt count registers that corresponds to the connected interrupt lines. To make sure that interrupts are masked, the limit for each interrupt is set to be low relative to the replenishment period. The test will exit when a fixed number of interrupts are handled, and report if it was successful. *intgen-tmu* is a combination of *or1200-tmu* and *intgen-simple* designed to test both functions of the TMU simultaneously.

Table 4.3: Tests performed by the SoC testbench for the TMU

| **Name** | **Test case** | **Language** |
|---|---|---|
| or1200-tmu | Test basic task time counting and exception generation | ASM |
| intgen-simple | Test basic interrupt generation and masking | ASM |
| inten-tmu | Test interrupt generation and masking while counting task execution time | ASM |
| tmu-full | Three running tasks that has a 10% chance of using to much time. Two interrupt sources that generate interrupts at a random interval. | C |

### 4.4.1 Full TMU test

The test *tmu-full* is the most extensive test of the TMU. It consists of three tasks which has a 10% chance of entering a spin lock inside its critical section. During operation interrupts from two different sources are handled, and the interrupt service routine for each source resets the interrupt generator to a random time. The tick timer is used to switch to the next task, and all the tasks are run for one tick period. All the configuration parameters of this test can be found in Table 4.4. The limits and ticks per second was set high to limit the simulation time.

## 4.5 Results of the OR1200 tests

All the tests described in Table 4.2 reported success when they were run both with and without the TMU added to the processor. The results of these tests with the TMU included in the system can be found in appendix B in *or1200-tests*.

Table 4.4: Parameters for the full TMU test

| **Parameter** | **Value** |
|---|---|
| Clock frequency | 50 MHz |
| Ticks/sec | 500 |
| Tick period | 100000 |
| Interrupt replenishment | 400000 |
| Task1 limit | 200000 |
| Task2 limit | 150000 |
| Task3 limit | 250000 |
| Interrupt0 limit | 5 |
| Interrupt1 limit | 4 |

The tests designed to check the functionality of the TMU as a part of the processor described in Table 4.3 were also able to complete successfully. The results of these tests can be found in Appendix B in *or1200-tmu-tests*. The results of the *tmu-full* test is found in Appendix B in *or1200-tmu-full*.

Only the key events of the *tmu-full* test is displayed here. In each event the involved signals along with the current program counter and current instruction is displayed. Table 4.5 shows the value of the current program counter, whether or not the TMU has generated an exception and the current interrupt mask. These values were sampled each time a change occurred in the TMU *intr* signal or *pic_ mask* during the simulation.

By comparing the program counters at the times the TMU exception was raised to the disassembly file for this program, the faulty tasks can be identified. The disassembly is found in appendix B in *or1200-tmu-full/tmu-full.dis* and the waveform file in *or1200-tmu-full/tmu-full.fst*

The waveform for this test is shown in Figure 4.3, where 4.3(a) shows the first half and Figure 4.3(b) shows the second half of the test. The following events are marked in the figures 4.3(a) and 4.3(b):

1. First interrupt arrives, source is line 4
2. Interrupt on line 3 is blocked by task 2
3. Task 2 reaches its limit and TMU generates the first exception
4. Interrupts on line 3 arrives for the sixth time, but it is masked
5. Replenish interrupt budget, interrupt 3 is unmasked, but it is blocked by task 1
6. Task 1 reaches its limit and TMU generates the second exception
7. Interrupts are blocked by task 3
8. Task 3 reaches its limit and TMU generates the third exception

### 4.5.1 Discussion of the results

The results of the standard OR1200 tests is only an indication of whether or not the OR1200 and its connected modules are functioning correctly. The successful execution of

Table 4.5: TMU full test events

| Time | PC | TMU exce | PIC mask |
|---|---|---|---|
| 423570 | 0x1de4 | 1 | 00000 |
| 5403410 | 0x1234 | 1 | 00000 |
| 7787030 | 0x11f8 | 0 | 00008 |
| 9485230 | 0x12c0 | 0 | 00018 |
| 12410990 | 0x1170 | 0 | 00000 |
| 14401850 | 0x1174 | 1 | 00000 |
| 19427170 | 0x12f0 | 1 | 00000 |
| 24148590 | 0x114c | 0 | 00010 |

these tests before adding the TMU logic serves as a reference point for how the system should operate. When the TMU is added to the system, the standard tests yielded the same result, indicating that the TMU has not had any unintended effects on the normal operation of the processor.

The correct execution of the tests for the TMU written in software yielded the same result in regards to the TMU as the tests in Table 3.5, indicating that the TMU behaves as specified even as a part of a larger system. The tests also show that the processor executes the correct exception vector when the TMU exception arrives, and that the added bit to the processors supervision register decides if the processor core will acknowledge the TMU exception or not.

## 4.6 Using the TMU

This section is meant as a guideline for anyone wanting to use the functionality provided by the TMU.

### 4.6.1 Task time counting

The values required for counting task execution time is *count* and *compare*. When starting to count the execution time for a new task the user must make sure that a value is written to the *compare* register, and then to the *count* register if a task should start without a full budget. To be able to write these values, the counting must first be stopped. When the values are loaded, counting is started by writing to the *START*-bit in *control*. If no value is loaded into *count*, this register should be set to zero by writing to the *RESTART*-bit in *control*.

During task execution time counting, the TMU is either stopped by writing to the *STOP*-bit in *control*, or when an exception is generated because the limit is reached. If an exception is generated, this is cleared by writing zero to the *count* register, or by setting *RESTART* or *CLEAR* in the *control* register. If a task is stopped before the task limit is reached the value in *count* can be read until a new value is written.

When switching between tasks, the user must make sure that a valid value is written to *compare* before the timer is started, to prevent the new task from using the previous task limit.

If the TMU is configured to count during exceptions, *CE*-bit in *SR* is set, it must be ensured that the TMU is stopped before loading new values.

### 4.6.2 Counting interrupts

To start counting interrupts a value must first be loaded into the *rep_ compare* register, and the *CID*-bit in *SR* must be set. This will start the interrupt replenishment counter. The limits for each interrupt line should be written to its corresponding *intr_ compare* register. All interrupts that exceeds its limit will be masked out before reaching the interrupt controller. The programmable interrupt controller remains unchanged. To stop masking of a specific interrupt, zero should be written to the *intr_ compare* register of that line. To stop counting and masking of all interrupts, zero should be written to the *rep_ compare* register.

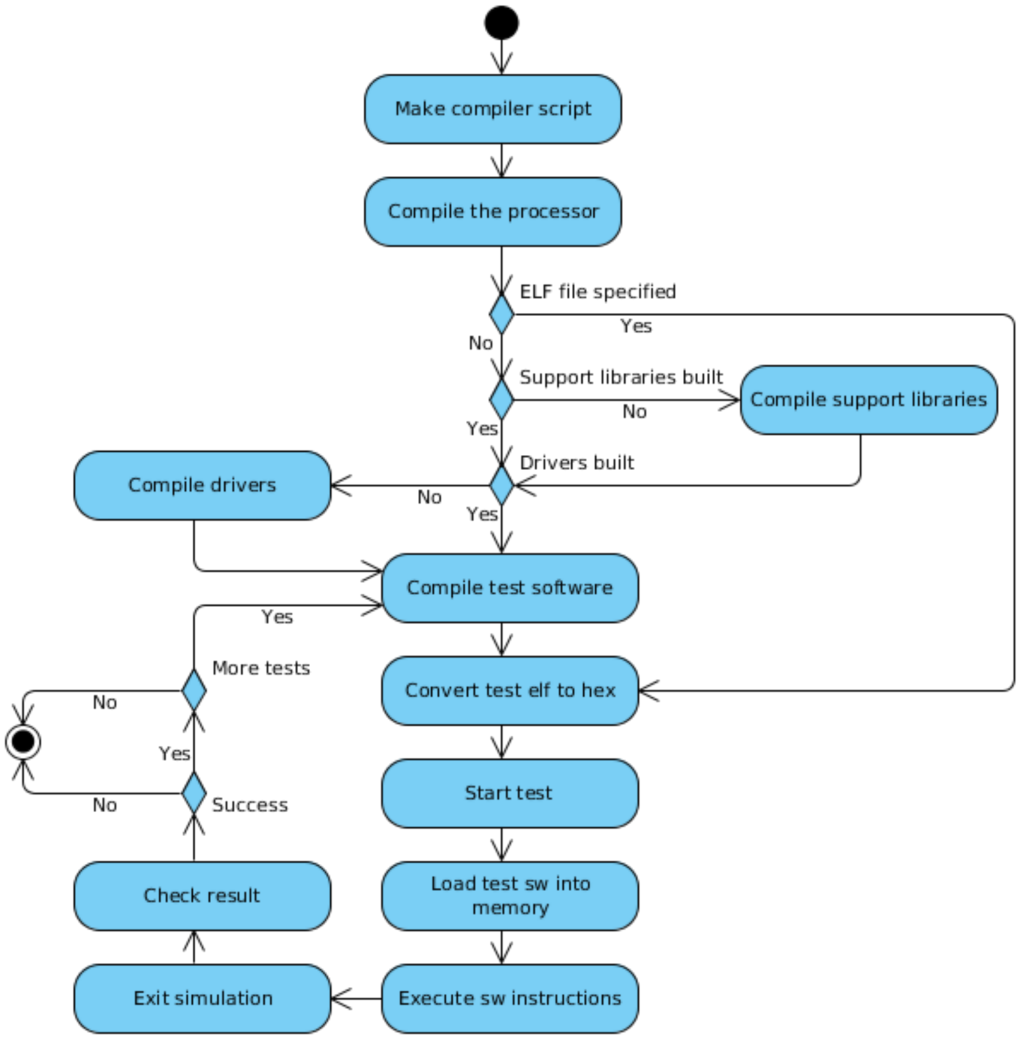
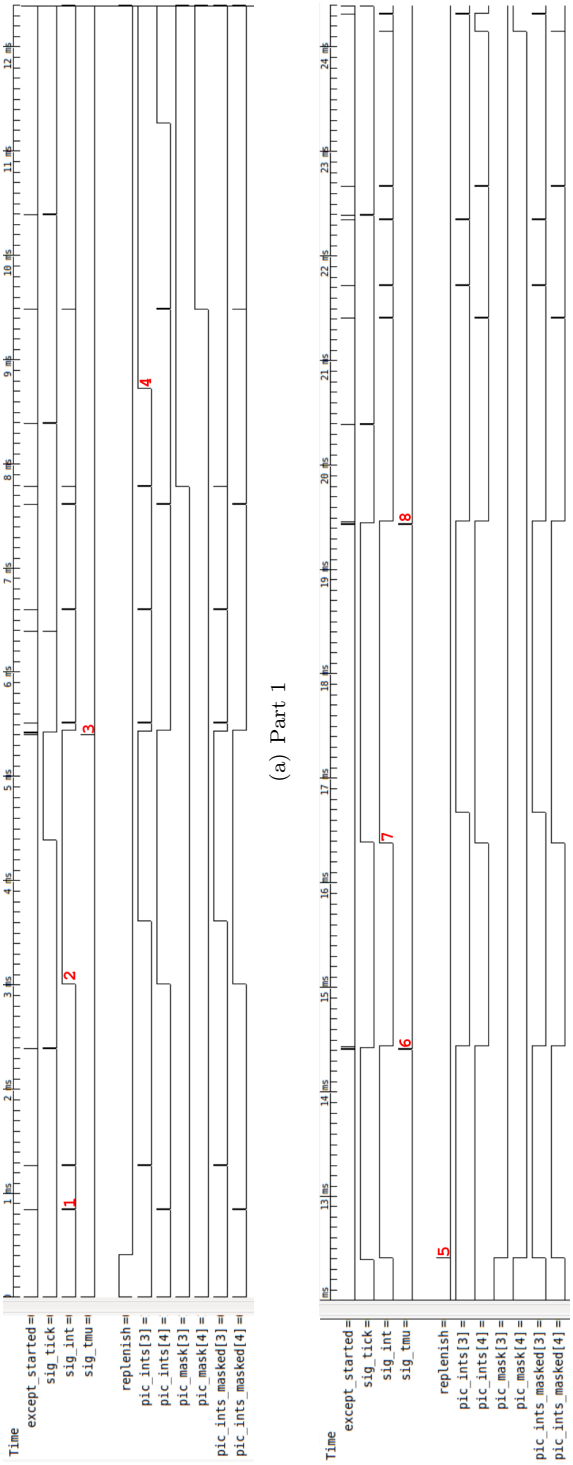



Figure 4.2: Makefile simulation flow




(a) Part 1

(b) Part 2

Figure 4.3: Key events from the TMU full test

# Chapter 5

# Or1ksim

*The following chapter starts by describing the relevant behaviour and mechanics within Or1ksim. It then continues to describe the modifications performed in this simulator. A driver for the TMU is then presented. The chapter proceeds with a section on verification of the TMU module and its driver in Or1ksim. The chapter ends with a short discussion about the modifications and results. Most information in this chapter is gathered from examining the source code. Citations are given if the information is found elsewhere.*

## 5.1 Or1ksim description

Or1ksim is an instruction set simulator for the OpenRISC 1000 instruction set architecture.

### 5.1.1 Downloading, installing and running

Or1ksim can be downloaded from the OpenRISC repository on github [10]. The version used in this thesis is from the commit hash: cae154d3fa13882b846030e5d2242fda1fe70604. The tool is installed by simply running the configure script, followed by make and make install.

Running Or1ksim is done by calling the *or32-elf-sim* command from shell. Refer to '–help' for the different options available for Or1ksim. '-i' enables the interactive prompt in the simulator, this allows for access to information regarding modules and memory.

### 5.1.2 Modules

Or1ksim is divided into several modules. Each module represents a similar hardware module and are seperated in different directories. Each module can be customized by changing variables in **sim.cfg**, a optional setup file for the simulation, included by giving the '-f'-option to *or32-elf-sim*. Modules are compiled via automake scripts in the Or1ksim source folder.

### 5.1.3 Running Or1ksim as debug server

Or1ksim can be used with remote debugging through a remote serial protocol (RSP) server for GDB. This allows debugging of software compiled for the OR1200 processor. Step one and tree in the explanation of Or1ksim behaviour is the handling of this connection.

### 5.1.4 Orksim structures

A selection of variables and structures are collected in Table 5.1

Table 5.1: A selection of variables and structures in Or1ksim

| File | Struct | | Variable | Description |
|---|---|---|---|---|
| sim-config.h | runtime | sim | cycles | Number of executed cycles |
| | | | mem_cycles | Number of memory cycles. |
| | | cpu | instructions | Number of instructions executed |
| | config | pic | | Configuration of Programmable Interrupt Controller |
| execute.h | cpu_state | | sprs[] | SPR register |

The **config**-struct holds configuration data for non-peripheral modules. These modules are mostly configurable via **sim.cfg**.

### 5.1.5 Or1ksim behavior

Upon calling the simulator from shell Or1ksim starts in the **_main**-function in **support.c**, which sends it to **main** in **toplevel.c**. **main** initializes all the different modules and configures Or1ksim for simulation. After Or1ksim is ready, **main** passes control to **exec_main**.

During the configuration process, Or1ksim reads **sim.cfg** and configures itself to the given specifications. **sim.cfg** is processed by **reg_config_secs** which calls the configuration functions to modules configurable via **sim.cfg**. Configuration functions have names like **reg_<module-name>_sec**, e.g. **reg_pic_sec** for the Programmable Interrupt Controller (PIC). Default values are set by **init_defconfig**, or module specific init functions if available. **sim_reset** is called before initializing the simulation, this function calls reset functions in some of the other modules e.g. **pic** and **tick**.

Or1ksim measures time in cycles, Figure 2.4. Normally one instruction is executed per cycle, the exceptions are reading and writing to memory, which has been observed, during simulation, to take two or three cycles, but no more.

**exec_main** is the main loop in Or1ksim and the following describes a simplified version, Figure 5.1:

1. If debug is enabled, handle debug commands as long as the CPU is stalled.

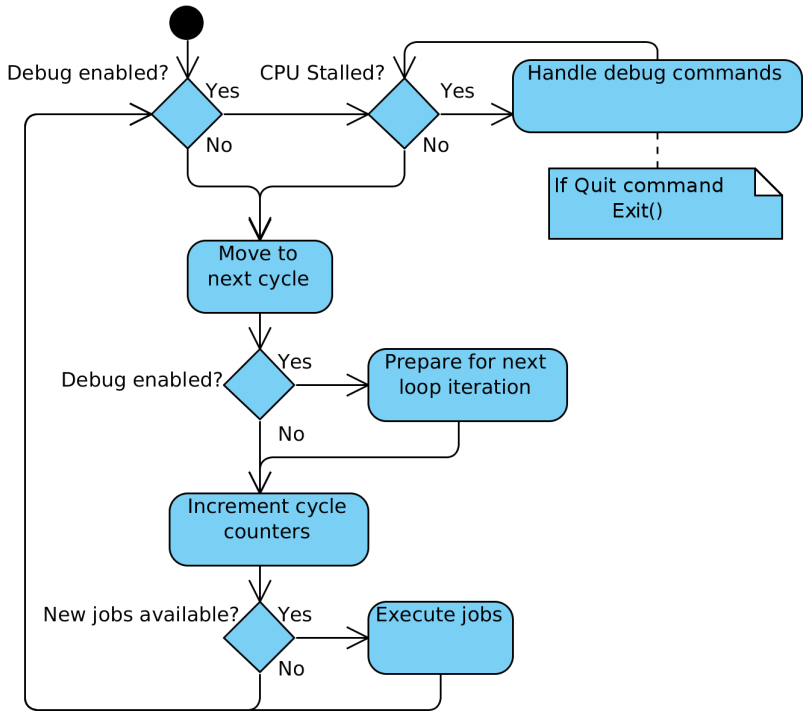



Figure 5.1: **exec_main**, from **execute.c**

2. Update the program counter and execute the next instruction.
3. If debug is enabled and if GDB is requesting single-step debugging, stall the CPU for next loop iteration and store a trap exception in the **rsp**-struct.
4. Increment the cycle counters.
5. If there are unblocked simulator jobs, like the tick timer, execute these.

Step two and four are where the actual execution of the instruction cycle takes place.

In step five **exec_main** allows the different modules to get runtime. This time is not included in the cycle count, but it is necessary to allow modules like tick timer to get runtime so it can issue its tick exception. Jobs scheduled to be executed here are from modules which do not necessarily need to run every cycle. As an example the tick timer is suppose to generate an interrupt every X clock cycles. This is done by scheduling a job at that cycle count in the future. This allows Or1ksim to reduce the overhead associated with trying to mimic concurrent hardware with sequential software.

### 5.1.6 Exceptions in Or1ksim

Exceptions in Or1ksim are issued through **except_handle**, which takes the two arguments: *except*, the exception vector for the exception, and *ea* the effective address of

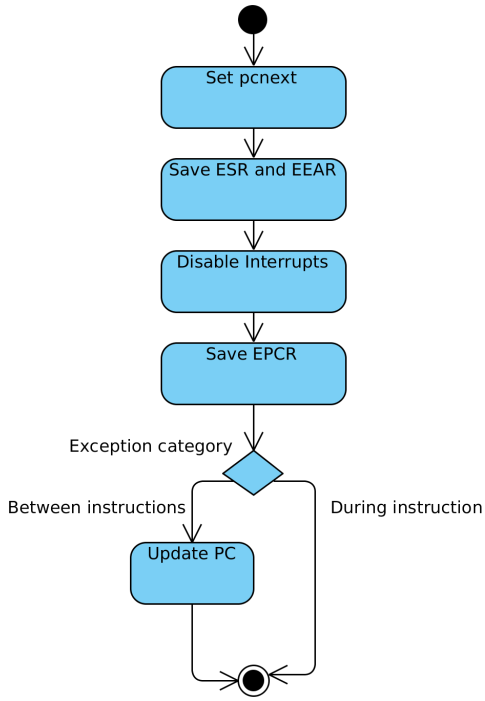



Figure 5.2: **except_handle**, from **except.c**

the instruction that generated the exception. The function behavior is described below and shown in Figure 5.2.

1. Set *pcnext* to the address of the exception.
2. Save the Supervision Register(SR) to Exception Status Register(ESR) and effective address to Exception Effective Address Register (EEAR) located in Special Purpose Registers(SPR).
3. Disable interrupts.
4. Save the program counter with necessary adjustment in Exception Program Counter Register(EPCR).
5. If the exceptions is in the *between-instructions*-category, move *pcnext* into the PC.

Necessary adjustments to the program counter can be subtracting four if *cpu_state.delay_insn* is set, adding four if its the *SYSCALL*-exception or saving *ea* instead if it is an *ITLBMISS-/IPF*-exception. When a *between-instruction*-exception occurs the program counter is already pointing to the next instruction to be executed. Without the check in step four updating PC would not happen until after the next instruction has been executed, if this next instruction is a jump it would cause serious problems, as stated in **except.c**.

## 5.2 Or1ksim changes

### 5.2.1 Exception handling

Support for TMU exceptions are added to **except_handle**. An exception vector *EXCEPT_TMU* is added and classified as a *between*-instructions exception, Figure 5.2. A check on the count during exception bit is included to allow the exception handle to stop the TMU counter if the bit is low. Supervision register(SR) is modified with the addition of Count Exception Enable *CEE*, this is set low when an exception is raised.

### 5.2.2 SPR

The changes made to **mtspr** in **sprs.c** are listed below, no changes to **mfspr** was necessary.

- Protection is added to the status register, replenishment counter and interrupt counters, so that these can not be written to.
- Protection is added to count and compare (high/low), so these can not be written to while the TMU is running.
- Count invalid is set during a write to count, and cleared when compare is written.

### 5.2.3 Programmable interrupt controller

In addition to adding an interrupt filter check at the start of the **report_interrupt**, described later, two additional changes were made.

**Ignored interrupts** The Programmable Interrupt Controller (PIC) prints out a message stating that it had ignored an interrupt. This indicates that the rate of interrupts on one line is to large, it occurs when an interrupt arrives and the corresponding bit is already set in *PICSR*. To measure the amount of issued, but ignored interrupts a series of SPR registers was added. This allows software to read the measured amount. The registers base is available at *0x4803*, *0x3* offset from the PIC group base, *0x4800*. There is one register for each line. Each register is offset from the base according to the line number. The print was disabled to improve simulator performance because large amount of calls to **printf** has an impact on simulation time.

**Simultaneous interrupts** The second change involved running a test with two interrupt sources, at seemingly random times the program would enter an infinite loop, looping on the complete external interrupt routine in FreeRTOS. This problem seemed like a race condition, triggered by the arrival of two interrupts within the same instruction cycle. After examining the **report_interrupt**-function the most likely suspect was when the function schedules a new interrupt. The first attempt to fix this was removing the call to the exception from the scheduling queue prior to issuing another, because there is no reason to signal this exception twice. The interrupt handler will read *PICSR* to check the raised interrupts, so scheduling an exception is just a way of signalling that an interrupt has been raised. This worked, so no effort was made to discover where exactly

Table 5.2: Configurable variables for the TMU in the **config**-struct

| Name | Description |
|---|---|
| enabled | Indicates whether or not the TMU is enabled. |
| status | Holds the reset value for the status register. |
| use_tt | Indicates if the task timer is enabled. |
| use_if | Indicates if the interrupt filter is enabled. |

Table 5.3: Configurable variables for the TMU in **sim.cfg**

| Name | Description |
|---|---|
| enabled | 1 or 0, default: 0 |
| status | Sets the default status of the module, default: *0x2* |
| task_timer | Enable/disable the task timer, default 0 |
| int_filter | Enable/disable the interrupt filter, default 0 |

the simulator fails, or if this is the best solution, both of these tasks fall outside the scope of this thesis and as shown in later chapters the module behaves as expected.

### 5.2.4 Time management module

The TMU module in Or1ksim is designed to the specifications given in Sections 3.3.

The TMU module is not scheduled via the Or1ksim scheduler. Instead there is a call to **tmu_main** each cycle of **exec_main**. This call includes the cycle count value, *runtime.sim.mem_cycles*, which is also added to *runtime.sim.cycles*.

The following are added to Or1ksim's configuration and setup functionality.

- **tmu**-section is added to **sim.cfg**, Table 5.3 .
- **reg_tmu_sec** is added to **reg_config_secs** in **sim-config.c**.
- **tmu**-struct, Table 5.2, is added to **config**-struct in **sim-config.h**.
- Default values are added to **init_defconfig**.
- **tmu_reset** is added to **sim_reset** in **toplevel-support.c**.

A configuring section for the TMU is added to the **sim.cfg**. **reg_tmu_sec** was created and added to **reg_config_sec**, this function will set the values in *config.tmu* according to the TMU section in **sim.cfg**. The default values are set in **init_defconfig** and the **tmu_reset** function is included in the system reset.

**tmu_status** is registered to the information function **sim_cmd_info**. This allows the user to check the TMU registers by giving the command 'info reg' to interactive prompt in the simulator.

All TMU-functions callable from outside of the module are protected by the enable parameters set up in *config.tmu*.

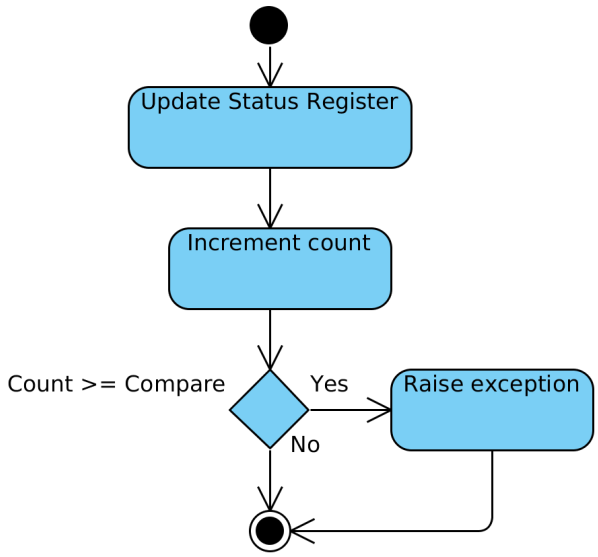



Figure 5.3: **tmu_main**, from **tmu.c**

The module is added to the makefiles by searching for the tick timer file names, and adding the TMU files accordingly, all while also referring to an automake tutorial. Due to the low amount of places that needed changing in the makefiles, it was easier to brute force this rather than learning all the ins and outs of automake.

The TMU module can be viewed as two separate modules and are explained as such, however, they use some of the same functions.

**Part one, task timer:** This part of the TMU is responsible for measuring the execution time of the current task. If the task exceed its given budget the task timer will raise a TMU exception. Task timer functionality is shown in Figure 5.3.

**tmu_main** calls **tmu_update_status**, **tmu_increment_count** and **tmu_compare**,in this order.

1. **tmu_update_status** checks the control register and applies the requested changes to the status register.
2. **tmu_increment_count** checks if the TMU is running, if so it adds *runtime.sim.mem_cycles* to the count value.
3. After checking if the task timer is running and TMU-exceptions are enabled in the processor SR and TMU SR. **tmu_compare** checks if the count value has exceeded compare, if so an exception is scheduled.

Exceptions are generated by scheduling **tmu_raise_except** in the current instruction cycle. **tmu_raise_except** calls **except_handle** with the arguments *EXCEPT_TMU*(*0xF00*) and the current contents of *ea*.

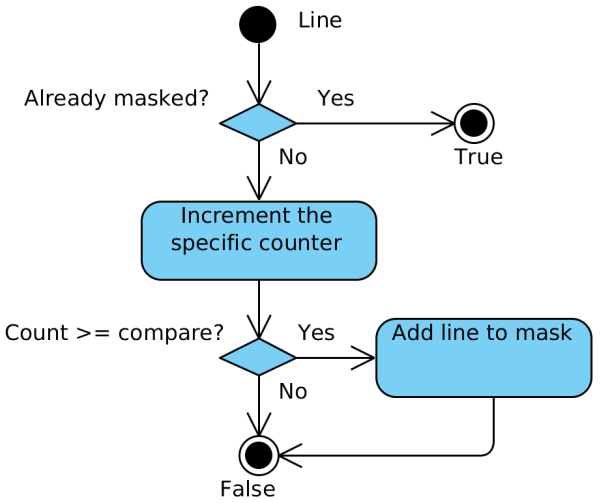



Figure 5.4: **tmu_int_filter**, from **tmu.c**

Another solution is to schedule an exception at a time equal to compare in the future, like the tick timer does. The reason for not choosing this implementation is that it requires a significantly more complex, outspread and intrusive implementation. For example, during the context switch upon reading the count value, this value needs to be calculated from within the **mfsr**-routine and updating the status register would have to be handled in **mtspr**. It means loosing cycle by cycle control over the TMU functions and values, which comes in handy when debugging the module.

**Part two, interrupt filter:** The main task of this part is to filter out interrupts that issues requests at an abnormal level. This is done by using an interrupt filter which ignores interrupts when they deplete their budget and the same count/compare functions from task timer to replenish the interrupt budgets.

As mentioned in Section 5.2.3, a call to the interrupt filter is added to the start of the **repport_interrupt**-function. This function dismisses interrupts raised on ignored lines. Returning *true* indicates the interrupt is filtered out, the function returns *false* otherwise. The function is explained below and shown in Figure 5.4, it is called with an integer representing which interrupt line is reported:

1. If count interrupts are disabled return *FALSE*.
2. Check if this interrupt is masked, return *TRUE* if it is.
3. If count interrupt is enabled, increment the interrupt counter.
4. Check if this interrupt lines counter has exceeded its limit. If true add the line to the mask.
5. Return false.

**tmu_increment_count** adds the same cycle count to replenishment count as the

task timer part of the function does. The addition is only performed if the interrupt filter is enabled and the replenishment compare value is higher than zero.

When the replenishment count exceed the replenishment compare value, **tmu_compare** sets the replenishment counter, filter mask and interrupt line counters to zero. The function also reasserts all the interrupts that was masked out. All actions are only performed if the interrupt filter is enabled and the compare value is above zero. If the count interrupt bit in SR is low, the compare function will unmask all interrupts.

Reasserting the interrupts are done to mimic how a hardware module would keep its interrupt signal high even if it is masked. So when the filter is disabled, it would appear to the processor like a new interrupt. If the interrupts are not reasserted by this function they would be lost, since interrupts in Or1ksim can be thought of as if their signal is passed through an edge-detector.

### 5.2.5 Interrupt generator

To properly test the TMU module, a separate module was created to generate interrupts. This module is accessible from software via the bus at address *0xA0L*.

Writing to the base address, which is directed to *interrupt0*'s time-out register, will schedule an **intgen0**-job at a time in the future equal to the written value. It also sets the *timeout*-variable and the *active*-flag in the module. Writing to *0xA0L+0x4* will clear the *timeout*-variable, the *active*-flag and the bit in *PICSR*, corresponding to the interrupt line. **intgen0** reports an interrupt to line 3, and **intgen1** reports on line 4. **intgen1** is written to like **intgen0**, but with an offset of *0x1000000*.

Reading from the time-out addresses returns the time-out written, and reading from the clearing addresses returns whether or not this interrupt is active. The reset function, **intgen_reset**, is added to the *sim_reset* routine. This will upon Or1ksim startup reset the interrupt generator, and print out the status of the module. **intgen_sec_start** sets the default values shown in Table 5.4.

During testing it was discovered that another version of the interrupt generator was required. This version will generate interrupt and schedule itself to generate another interrupt at a randomly selected cycle in the future. This loop is triggered by writing to the time-out of address of the module, this sets the time-out for the first interrupt. After this initial time-out, the succeeding interrupts are scheduled with a time-out set by using the **rand**-function in **stdlib**. There is no way to stop this interrupt loop when it is triggered. A stopping function was not needed for the interrupt tests. This could easily be implemented by adding an additional case statement in the write function. When written to, this case statement should issue a call to **SCHED_FIND_REMOVE** which can remove a scheduled job, thus breaking the loop. This functionality was not needed because an uncontrollable interrupt source is wanted to properly test the TMU's interrupt filter.

The two versions can be changed between by setting version in **sim.cfg** to one or two. The maximum limit for random values can be set by changing the *RAND_MAX*-definition in **intgen.c**.

Table 5.4: Configurable variables in **sim.cfg**

| Name | Description |
|---|---|
| Enabled | 1 or 0, default: 0 |
| BaseAddress | Sets the base address for the module, default: *0xA0000000* |
| Size | Size of the memory allocated to this module, default: *0x1000008* |
| Version | 1 or 2, Choosing between the version, default: 1 |

#### Creating the module

The module based on the **generic** module in **generic.c/h**,this is an example module included in the Or1ksim files. Non-essential functions were deleted and the remaining functions were re-written to act as a interrupt generator. Using the module registration facilities allows this module to easily be modified in **sim.cfg**, Table 5.4 shows these.

To add compilation support, the same approach as with the TMU was taken, adding the intgen files in the makefiles wherever the other peripheral modules were mentioned.

## 5.3 TMU driver

To easily control the TMU module a set of functions is implemented to provide an interface for the user. These functions can be found in **tmu.c/h** in the **arch** subfolder of the FreeRTOS folder. Table 5.5 lists the driver function names and effects. As of now the TMU in Or1ksim does not provide the option to select whether or not to count during freezes. The driver functions regarding this bit will only have an effect on SR.

## 5.4 Verifying Or1ksim

The GNU debugger was used during the development of Or1ksim. Since FreeRTOS was already ported and was working on Or1ksim, testing of the final implementation could be done via FreeRTOS. First the driver functions were tested, then these drivers were used to test the task timer and interrupt filter behaviour. These test are run in FreeRTOS without the TMU functionality.

The print out from the complete test can be viewed in Listings D.1 in Appendix D.1. The listings in this section is an excerpt from the listing in appendix. How to run this test is explained in the tutorial in Appendix C.5.

### 5.4.1 TMU driver

All the functions in the driver was called and the effect on the TMU was verified to be as intended. **tmu_driver_test** in **tmutest.c/h** calls all the drivers and checks their effect. The functions have been tested to verify that they work as intended, but not for all possible combinations and orders. Table 5.6 shows special test cases outside of verifying the behaviour from Table 5.5

Table 5.5: Functions available in the TMU driver and description

| **Function** | **Description** |
|---|---|
| tmu_start | Starts the TMU counter. |
| tmu_stop | Stops the TMU counter. |
| tmu_restart | Restarts the TMU counter. |
| tmu_clear | Clears count and compare. Counter is stopped. |
| tmu_enable_exception | Set the exceptions enabled bit in TMU SR. |
| tmu_disable_exception | Clear the exceptions enabled bit in TMU SR. |
| tmu_enable_count_exception | Set the count exceptions enabled bit in TMU SR. |
| tmu_disable_count_exception | Clear the count exceptions enabled bit in TMU SR. |
| tmu_enable_count_freeze | Set the count freeze enabled bit in TMU SR. |
| tmu_disable_count_freeze | Clear the count freeze enabled bit in TMU SR. |
| tmu_enable_count_interrupt | Set the count interrupt enabled bit in TMU SR. |
| tmu_disable_count_interrupt | Clear count interrupt enabled bit in TMU SR. |
| tmu_set_control(*val*) | Sets the control register to the unsigned long *val* given as parameter. |
| tmu_get_status | Returns the contents of the Status Register as unsigned long. |
| tmu_set_compare(*val*) | Sets the 64 bit compare register to the given unsigned long long parameter *val*. |
| tmu_get_compare | Returns the contents of compare as an unsigned long long value. |
| tmu_get_count | Returns the contents of count as an unsigned long long value. |
| tmu_set_interrupt_rep_period(*val*) | Sets the replenishment period for the interrupt counters to the unsigned long long parameter *val*. |
| tmu_get_interrupt_rep_period | Returns the interrupt replenishment period as an unsigned long long. |
| tmu_set_interrupt_compare(*line*, *val*) | Sets the arrival limit, *val*, for the interrupt, *line*. |
| tmu_get_interrupt_compare(*line*) | Returns the limit for the requeste interrupt, *line*, as an unsigned long. |
| tmu_get_interrupt_count(*line*) | Returns the count value for the requested interrupt, *line*, as an unsigned long. |

Table 5.6: TMU driver special cases tests

| Test Case | Special condition | Expected result |
|---|---|---|
| All initial values. | Check if all values are initiated correctly. | All values are correct at startup. |
| Starting when started. | Make sure the counter does not reset. | Counter continues without change. |
| Stopping when stopped. | Make sure the counter remains stopped. | Counter remains stopped. |
| Writing compare while running. | Make sure compare can not be written to without stopping. | Compare is not updated. |
| Writing count while running. | Make sure count can not be written to without stopping. | Count is not updated. |
| Writing to replenishment count. | Make sure replenishment count can not be written to. | Replenishment count is not updated. |
| Writing to interrupt counters. | Make sure interrupt counters can not be written to. | Interrupt counters are not changed. |

Listing 5.1: Results from the TMU driver test on Or1ksim

```
tmu_driver_test returns with 0 errors
```

All the test cases passed, and the test returns with zero errors, printout from the test is shown in Listings 5.1

### 5.4.2 TMU behavior

The two parts of the TMU is tested separately, the tests are described in the following paragraphs.

**Task timer test** Task timer test is set up like a state machine, Table 5.7. States two to five sets up and initiates different tests. State zero is the idle state where the function waits on results from the system. When a test returns, state zero calls **tmu_clear** and restores the TMU so it is ready for the next test. State one exits the test and returns the number of errors.

Listing 5.2: Results from the TMU task timer test on Or1ksim

```
TMU:: Scheduling exception, Compare = 400000
res2_except
TMU:: Scheduling exception, Compare = 4294967297
res2_except
TMU:: Scheduling exception, Compare = 1200000
res2_except
tt_behavior_test returns with 0 errors
```

Listings 5.2 shows an excerpt of the printout from the complete test. It shows that the total test program returned with zero errors and that the expected exceptions were

Table 5.7: Task timer tests

| # | **Test Case** | **Test Behaviour** | **Expected result** |
|---|---|---|---|
| 2 | Short compare | Start timer with short compare, 400 000 | Exception is raised quickly. |
| 3 | Long compare | Start timer with long (33-bit) compare | Exception is raised. |
| 4 | Exception disabled in TMU | Disable exception in TMU SR, short compare, check vs. tick timer | Tick timer reaches limit before exception is raised. |
| 5 | Exception disabled in SR | Disable exception in SR, short compare, check vs. tick timer | Tick timer reaches limit before exception is raised. |
| 6&7 | Extending compare | Starting the TMU with 600 000 as compare value, state six, after one tick set compare value to 1 200 000, state seven. | Exception generated in state seven, and not state six, state six ends after approx 500 000[1] cycles. |

produced. The absence of generated exceptions from state four, five and six shows that those states completes as expected. State seven produces an exception with the compare value 1 200 000. Without the TMU functionality in FreeRTOS the *0xF00*-exception calls the **res2_except** function which prints out *res2_except*. During this test **res2_except** also calls **tmutest_exception**, the exception handling function for this test.

Additionally the print out from Or1ksim shows that the compare and count values are not flipped, so MSB and LSB remains the same in both FreeRTOS and Or1ksim. Compare value set was *0x100000001* which equals the printed integer: 4 294 967 297.

**Interrupt filter tests** Table 5.8 shows the explanation of different states in this test. It starts by testing the filter and shows that the TMU reasserts the interrupt. The second state shows how the filter reacts under pressure from multiple sources spamming interrupt requests.

The test stops after the first interrupts in the fifth replenishment period. During replenishment period one and two, state zero is active. It is clear from the results that this test works as expected. Interrupt line tree is ignored and the interrupts stops, when the budget is replenished the interrupt gets reasserted by the TMU and the loop continues until it is once again ignored, Listings 5.3. During state one the first interrupt triggered the filter, but passed. This interrupt disabled the interrupt counter and unmasked all interrupt lines. The next four interrupts passes the filter without being counted. The fifth interrupt is counted and the mask is set. When the sixth interrupt arrives it is ignored. This is shown in Listings 5.4

State two is active in the fourth replenishment period, and from the results it is clear that the interrupt filter ignores interrupts as they deplete their budget, Listing 5.5

Table 5.8: Interrupt filter tests

| # | Test Case | Test Behaviour | Expected result |
|---|---|---|---|
| 0 | Reasserting | While loop which requests a new interrupt for each arrival | Interrupts stops when exceeding the limit(2), and continues in the next replenishment period. |
| 1 | Enable/disable count interrupts | While loop which requests a new interrupt for each arrival. Upon the first interrupt, disable count interrupts, re-enable upon the fifth. | After the first interrupt, four interrupts will pass the mask, the fifth interrupt will be counted, and the sixth will be ignored. |
| 2 | Ignoring multiple | While loop which requests two interrupts every other tick period | Interrupts will be repeatedly ignored when they exceed their limit(2 and 3). |

Listing 5.3: Results from the TMU interrupt filter test on Or1ksim, state zero

```
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 0 Pass:: 3 | # 1
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 1 Pass:: 3 | # 2
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 2 Ignored 3 | # 1

TMU:: Replenish ---------------
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 0 Pass:: 3 | # 3
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 1 Pass:: 3 | # 4
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 2 Ignored 3 | # 2
```

Listing 5.4: Results from the TMU interrupt filter test on Or1ksim, state one

```
TMU:: Replenish ---------------
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 0 Pass:: 3 | # 5
TMU:: Filter:: Count interrupts are disabled :: Int 3
TMU:: Filter:: Count interrupts are disabled :: Int 3
TMU:: Filter:: Count interrupts are disabled :: Int 3
TMU:: Filter:: Count interrupts are disabled :: Int 3
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 1 Pass:: 3 | # 6
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 2 Ignored 3 | # 3
```

Listing 5.5: Results from the TMU interrupt filter test on Or1ksim, state two

```
TMU:: Replenish ----------
TMU:: Filter:: Checking line 3
    Budget = 2    Count = 0 Pass:: 3 | # 7
TMU:: Filter:: Checking line 4
    Budget = 3    Count = 0 Pass:: 4 | # 1
TMU:: Filter:: Checking line 3
    Budget = 2    Count = 1 Pass:: 3 | # 8
TMU:: Filter:: Checking line 4
    Budget = 3    Count = 1 Pass:: 4 | # 2
TMU:: Filter:: Checking line 3
    Budget = 2    Count = 2 Ignored 3 | # 4
TMU:: Filter:: Checking line 4
    Budget = 3    Count = 2 Pass:: 4 | # 3
TMU:: Filter:: Checking line 3
    Budget = 2    Count = 2 Ignored 3 | # 5
TMU:: Filter:: Checking line 4
    Budget = 3    Count = 3 Ignored 4 | # 1
TMU:: Filter:: Checking line 3
    Budget = 2    Count = 2 Ignored 3 | # 6
TMU:: Filter:: Checking line 4
    Budget = 3    Count = 3 Ignored 4 | # 2
TMU:: Filter:: Checking line 3
    Budget = 2    Count = 2 Ignored 3 | # 7
```

## 5.5 Discussion

Testing shows that the Or1ksim TMU module works as expected. All the driver function produce the expected reaction in the TMU module. The TMU generates exceptions correctly and it is possibly to assign compare while a task is executing.

The driver provides an easy to use interface for controlling the TMU.

Or1ksim provides a good base for developing applications that uses the TMU. The possibility to debug by using GDB on both the simulator and the application provides a great way to locate and correct errors. The simulator will also work as a reference point for tests run on the FPGA.

Chapter 6

# FreeRTOS

*This chapter starts out by describing the normal operation of FreeRTOS, the information here is mostly gathered from examinig the source code. Appropriate citations are given when the information is taken from elsewhere. After a description of the different functionalities, this chapter continues on to describing the modifications made to accomodate the Time Managemen Unit. The chapter proceeds with a test of the modified FreeRTOS, and finishes with a short discussion of the modifications and test results.*

## 6.1 FreeRTOS description

### 6.1.1 Introduction

FreeRTOS is a small and simple real-time operating system developed for embedded applications. The FreeRTOS website highlights [2]:

- Pre-emptive scheduling option.
- Co-operative scheduling option.
- 6k to 10k ROM footprint.
- Mutexes and semaphore support.
- Very efficient software timers.

The FreeRTOS version used in this thesis is 6.1.1. It is already ported and tested for the OpenRISC 1000 instruction set architecture simulator, Or1ksim, and can be aquired via SVN checking out [5], the FreeRTOS used in this thesis is from 29.05.2014.

FreeRTOS was chosen because it is a simple RTOS with a small memory requirement at its bare minimum. The authors have previous experience with this RTOS through previous courses, as stated in the prelimenary project for this theses [6].

### 6.1.2 Memory layout

An example of the memory layout of FreeRTOS is shown in Figure 6.1, numbers are collected from a map file after compiling FreeRTOS. The placement of the different

sections are defined in the linker configuration file i Appendix B in *link.ld*.

### 6.1.3 Naming conventions in FreeRTOS

Tabel 6.1 lists the naming convention used in FreeRTOS.

Table 6.1: Naming conventions in FreeRTOS

| Prefix | Meaning |
|---|---|
| c | variable of type *char* |
| s | variable of type *short* |
| l | variable of type *long* |
| f | variable of type *float* |
| d | variable of type *double* |
| e | enumerated variables |
| x | struct |
| v | void |
| p[s] | pointer to [short] |
| u[s] | unsigned variable of type [short] |
| prv | private functions |
| v[Task]Create | void function in **task.c** |
| API functions are prefixed with their return variable type | |

### 6.1.4 Task

The executing entity in FreeRTOS is called task instead of process. Only one task is allowed to execute at a time, this task is chosen by the scheduler [2]. Tasks have no knowledge of the scheduler activity, thus it is the job of the scheduler to ensure memory integrity when swapping contexts on the processor. This is achieved by each task having its own stack, where all information related to this task is stored. A task operates within its own context with no coincidental dependencies to other tasks or the scheduler.

In FreeRTOS a task should not return, but instead be deleted by itself or other tasks[2]. This does not immediately free the resources previously allocated to the task, instead the task is moved to the exit state. This tells the scheduler that it should ignore this task. Deallocating the resources is normally performed in the idle-task, as this has the lowest priority this might take some time.

#### Task states

Tasks in FreeRTOS can exist in the following states: Suspended, blocked, ready and running [2] in addition adding the state exit is useful for understanding.

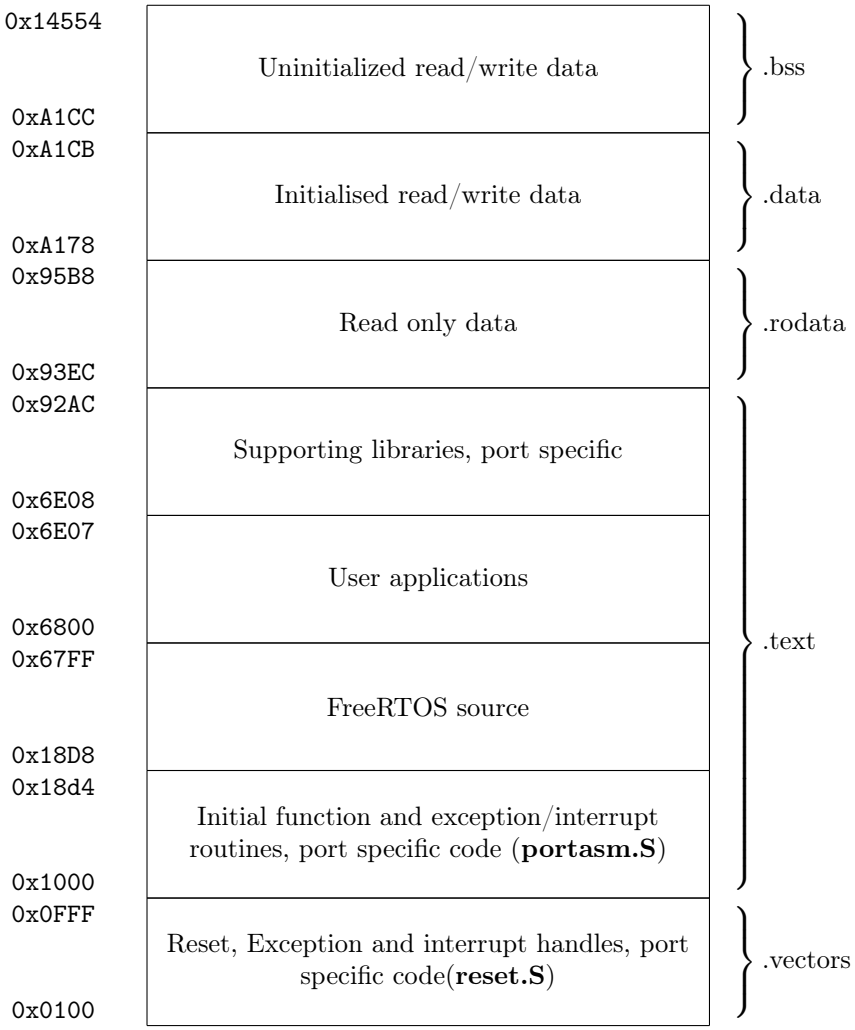



Figure 6.1: Example of FreeRTOS memory layout

Table 6.2: Excerpt of TCB from `task.h`

| Struct member | Description |
|---|---|
| pxTopOfStack | Points to the location of the last item placed on the tasks stack |
| xGenericListItem | List item used to place the TCB in the ready, blocked and terminate lists |
| xEventListItem | List item used to place the TCB in event lists. |
| uxPriority | The priority of the task where 0 is the lowest priority. |
| pxStack | Points to the start of the stack. |
| pcTaskName | Descriptive name given to the task when created. |

**Ready:**
Representing that a task is ready to enter the running state. New tasks are immediately placed in the ready state.

**Running:**
Execution state, *pxCurrentTCB* points to the task in this state.

**Blocked:**
The task is waiting for some event. When the event is triggered the task moves to the ready state.

**Suspended:**
A task can only reach this state by suspending itself, calling **vTaskSuspend**.

**Exit:**
Task placed on the **xTasksWaitingTermination** list after calling **vTaskDelete**.

When a task is initialized it is placed in one of the ready lists. Upon initialization one ready lists are created per available priority level. Internally the ready lists is organized by the tasks priority, highest through lowest priority. **xTaskWaitingTermination**-list is used by the idle-task to fulfil its duties as garbage collector.

### Task structures

The process control block is called task control block(TCB) in FreeRTOS, Table 6.2 shows the minimum necessary members, derived from `task.h`.

The global pointer **pxCurentTCB** points to the task currently in the running state.

Figure 6.2 shows how the context of a task is saved on the task stack in FreeRTOS. The internal ordering of the registers are set to accommodate for cache performance. When loading and saving the context the memory is accessed in a successive order, this decreases the chance of cache misses, since it is common to collect blocks from memory[8, p.78].

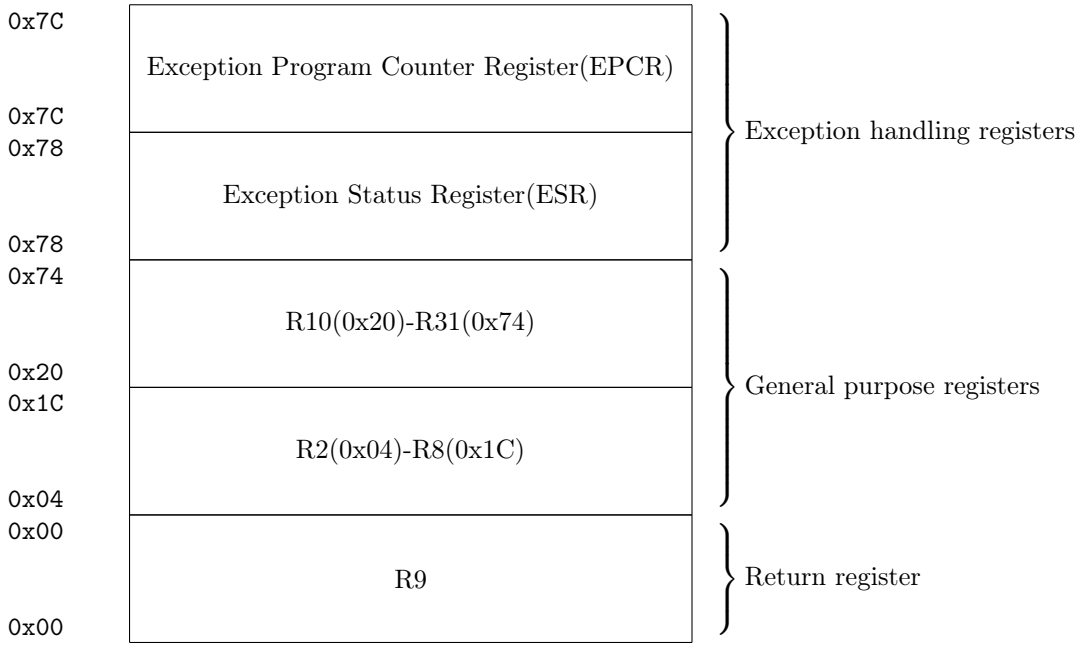



Figure 6.2: Context layout on the stack, addresses are offset from **pxTopOfStack**, derived from **portmacro.c**

### Task creation

User tasks are created from **main** and the only operating system task, the idle-task, is created in **vTaskStartScheduler**. From **vTaskCreate** the user task is created through the following steps:

1. **vTaskCreate** is a compiler macro which calls **vTaskGenericCreate**, shown in Figure 6.3. In addition to the arguments passed by **vTaskCreate**, shown in Table 6.3, are some parameters used when the processor requires MPU wrappers.

2. **vTaskGenericCreate** starts by allocating room in memory for the new task, calling **prvAllocateTCBAndStack**. The amount of allocated memory equals the size of TCB plus the requested stack size.

3. If the previous step was successful, **vTaskGenericCreate** sets the variables in the TCB, calling **prvInitialiseTCBVariables**.

4. Next it sets up the stack according to what the operating system expects when loading a new task onto the processer. This is done by **prvPortInitialiseStack**. Now the task is set up and ready to enter *running*.

5. **vTaskGenericCreate** checks if this is the first task created, by checking if *pxCurrentTCB* is already set. If it is the first, *pxCurrentTCB* is set to this task,

Table 6.3: Parameters for **vTaskCreate**, from **task.h**

| Parameter | Description |
|---|---|
| pvTaskCode | Pointer to the task entry function. |
| pcName | A name for the task. |
| usStackDepth | Number of variables the stack can hold. |
| pvParameters | Pointer to task specific parameters. |
| uxPriority | Priority for the task. |
| pvCreatedTask | Pointer to the TCB created for this task. |

and the state lists will be initialized by calling **prvInitialiseTaskLists**.

6. If *pxCurrentTCB* is set and the scheduler is not running, the priorities are checked and *pxCurrentTCB* is set to the highest prioritised task.

7. The task is added to the appropriate *ready*-list, depending on its priority.

8. Lastly **vTaskGenericCreate** does a check to see if the new task has a higher priority than the *pxCurrentTCB*-task, if so *pxCurrentTCB* is moved and a context switch is initiated with the **portYEILD_WITHIN_API**-macro.

**pxPortInitialiseStack** is responsible for writing the initial stack to memory, including setting the EPCR and ESR which is loaded into PC and SR when execution of this task start.

### 6.1.5 Exceptions and interrupts

Interrupt and Exception handling follows the procedure explained in Section 2.1.4. The service starts at the interrupts/exceptions vector defined in reset.S, these are located in the .vectors memory area, starting at *0x0100* and stretching to *0x0FFF*. This is were the processor jumps after confirming there has been an exception. The different functions are organized in memory by using the *.org <address>* flag, which assigns the first line of the succeeding code to the specified address. These architecture specific assembly functions sends the program into **portasm.S**, after setting some arguments, if necessary. As an example, the path of an external interrupt is explained below and shown in Figure 6.4

1. External interrupt source signals the processor and the exception handler at address 0x800 is executed.

2. The exception handle issues a *l.j vPortExtIntHandler*. **vPortExtIntHandler** is located in **portasm.S**.

3. This starts out by saving the context, with the **portSAVE_CONTEXT**, before jumping to **int_main**.

4. **int_main** collects the programable interrupts controllers status register(PICSR).

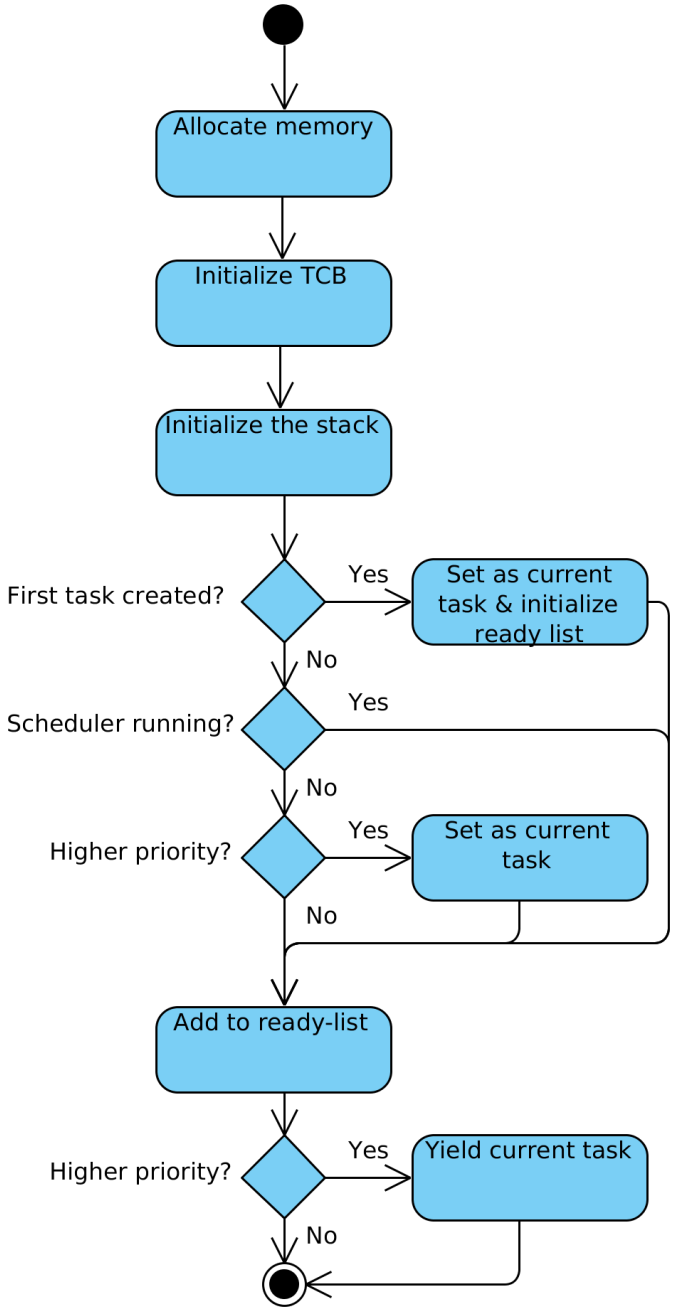



Figure 6.3: Task Create

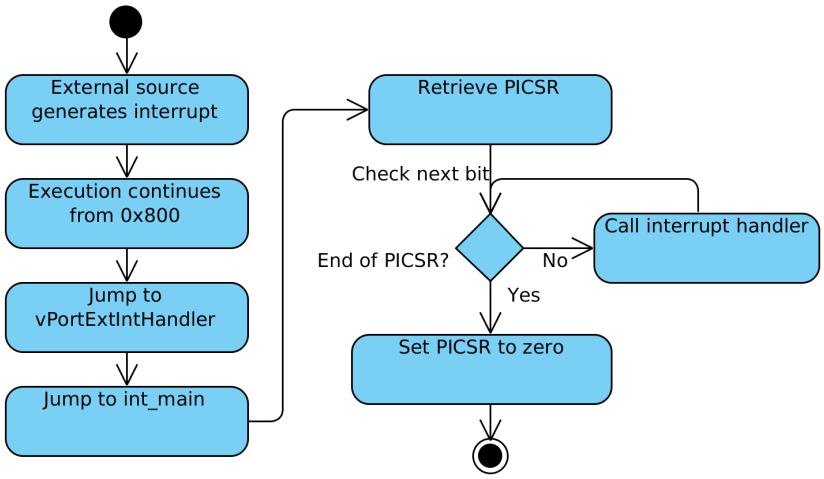



Figure 6.4: Path of the external interrupt in FreeRTOS

5. **int_main** iterates through every bit in this register looking for a high interrupt. When finding one **int_main** realeases control to the corresponding interrupt routine.

6. This loop continues to all the bits have been checked. Before returning **int_main** sets PICSR to zero, knowing non-handled interrupts will be re-asserted.

7. After **int_main** returns to **vPortExtIntHandler**, **vTaskSwitchContext** is called, this function moves *pxCurrentTCB* to the highest prioritized task in the ready queue.

8. Lastly the routine will restore the context of the task pointed to **pxCurentTCB** by issuing the **portRESTORE_CONTEXT**-macro.

This is mostly the software part of the general operating system interrupt routine. The values saved from *EPCR* and *ESR*, in **portSAVE_CONTEXT**-macro, are updated by hardware to the value at the time of interrupt prior to updating the program counter to *0x800*, as with the general handling.

### 6.1.6 Scheduler

In FreeRTOS there is no task with the scheduler responsibility. All scheduling associated functions are called either by a user task or by exception/interrupt handlers. To make it easier to understand, the scheduler will be portrayed as a single entity as if it was a process.

FreeRTOS has two primary scheduling modes, pre-emptive and co-operative. If it is in co-operative all tasks must yield to release processor control, using the **portYIELD**-macro, because **vTaskSwitchContext** is disabled in all task switching

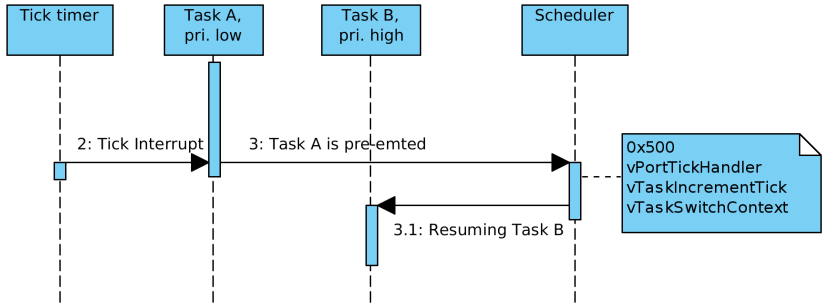



Figure 6.5: Task A is running, gets interrupted, Task B gets control

routines. The modes are set in **FreeRTOSConfig.h** by the define variable *configUSE_CO_ROUTINES*. In this thesis the scheduler is set to use preemtion, meaning that the scheduler can force a context switch.

Periodically a hardware timer will signal an interrupt to the processes, this interrupt is called the *tick-timer*. The period can be set in **FreeRTOSConfig.h**. This interrupt is handled by the **vPortTickHandler** in **portasm.c**. In short, the function stores current context, increments tick count, manages state-lists, moves *pxCurrentTCB* to highest prioritized task and finally restores the new context. This is the active part of the scheduler, all other functions are called passively from either this routine, the idle-task or user tasks.

Figure 6.5 shows normal execution with two tasks. Task A is running with low priority, while task B(high priority) is blocked waiting for the next tick. At the end of the time slice, the tick interrupt is generated. This pre-empts A and gives the scheduler control over the processor. The scheduler increases tick count and discovers task B has been unblocked and puts B into the ready queue. *pxCurrentTCB* will be set to task B in **vTaskSwitchContext**, because B has a higher priority. **portRESTORE_CONTEXT** will load task B onto the processor and resume the execution of B. This policy is called fixed priority pre-emptiv scheduling. Processes with equal priority will get runtime according to a round-robin policy.

### 6.1.7 Context switch

The term refers to switching processor control between two tasks. The context switch in FreeRTOS is primarily done by the **portSAVE_CONTEXT**-, **portRESTORE_CONTEXT**-macros and **vTaskSwitchContext**. The three functions are explained below, in the order of execution during a context switch.

**portSAVE_CONTEXT-macro** , Listing 6.1:

1. Make room in the stack for the context.
2. Early save some registers, to be used as clobber register[1]

[1] Clobber registers are a term used for overwriteable registers.

3. Save the Supervision Register(SR) and Program Counter(PC) at the time of the exception, respectively from ESR and EPCR.
4. Save the content of the remaining registers.
5. Refresh the **pxTopOfStack**-pointer in the TCB.

Listing 6.1: **portSAVE_CONTEXT**-macro

```
.macro portSAVE_CONTEXT
  .global pxCurrentTCB
  # make rooms in stack
  l.addi  r1, r1, -STACKFRAME_SIZE
  # early save r3-r5, these are clobber register
  l.sw  0x08(r1), r3
  l.sw  0x0C(r1), r4
  l.sw  0x10(r1), r5
  # save SPR_ESR_BASE(0), SPR_EPCR_BASE(0)
  l.mfspr r3, r0, SPR_ESR_BASE
  l.mfspr r4, r0, SPR_EPCR_BASE
  l.sw  0x78(r1), r3
  l.sw  0x7C(r1), r4
  # Save Context
  l.sw  0x00(r1), r9
  l.sw  0x04(r1), r2
  l.sw  0x14(r1), r6
  l.sw  0x18(r1), r7
  l.sw  0x1C(r1), r8
  l.sw  0x20(r1), r10
  l.sw  0x24(r1), r11
    .
    .
    .

  l.sw  0x74(r1), r31
  # Save the top of stack in TCB
  l.movhi r3, hi(pxCurrentTCB)
  l.ori r3, r3, lo(pxCurrentTCB)
  l.lwz r3, 0x0(r3)
  l.sw  0x0(r3), r1
  # restore clobber register
  l.lwz r3, 0x08(r1)
  l.lwz r4, 0x0C(r1)
  l.lwz r5, 0x10(r1)
.endm
```

**vTaskSwitchContext** , Figure 6.6:

1. If the scheduler is suspended, switching the task will not be allowed.
2. Check if there is a stack overflow for the current task.
3. Iterate through the prioritized ready lists to find the highest prioritized non-empty list.
4. Move *pxCurrentTCB* to the next entry in this list.

**portRESTORE_CONTEXT** , Listing 6.2:

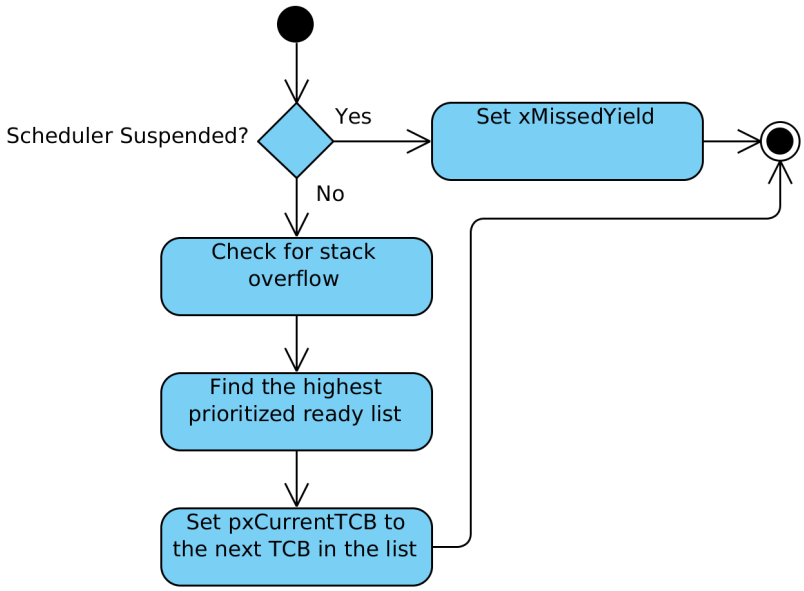



Figure 6.6: **vTaskSwitchContext**

1. Load the stack pointer from *pxCurrentTCB*
2. Restore the non-clobber registers.
3. Load the previous PC and SR into SPR.
4. Restore clobber registers.
5. Move the stackpointer.
6. Issue *l.rfe*(return from exception). This loads the PC and SR from SPR and resumes exception from this point.

Listing 6.2: **portRESTORE_CONTEXT**-macro

```
.macro portRESTORE_CONTEXT
  l.movhi r3, hi(pxCurrentTCB)
  l.ori r3, r3, lo(pxCurrentTCB)
  l.lwz r3, 0x0(r3)
  l.lwz r1, 0x0(r3)
  # restore context
  l.lwz r9,  0x00(r1)
  l.lwz r2,  0x04(r1)
  l.lwz r6,  0x14(r1)
  l.lwz r7,  0x18(r1)
  l.lwz r8,  0x1C(r1)
  l.lwz r10, 0x20(r1)
  l.lwz r11, 0x24(r1)
    .
    .
    .

  l.lwz r31, 0x74(r1)
  # restore SPR_ESR_BASE(0), SPR_EPCR_BASE(0)
  l.lwz r3,  0x78(r1)
  l.lwz r4,  0x7C(r1)
  l.mtspr r0,  r3, SPR_ESR_BASE
  l.mtspr r0,  r4, SPR_EPCR_BASE
  # restore clobber register
  l.lwz r3,  0x08(r1)
  l.lwz r4,  0x0C(r1)
  l.lwz r5,  0x10(r1)
  l.addi  r1,  r1, STACKFRAME_SIZE
  # move
  l.rfe
  l.nop
.endm
```

## 6.2 FreeRTOS modifications

### 6.2.1 Context layout

Figure 6.7 shows how the TMU registers are placed in the context stack. The addresses are chosen to remain cache friendly while they are written back and forth from memory. Addresses *0x00-0x7C* are kept unchanged, Figure 6.2.

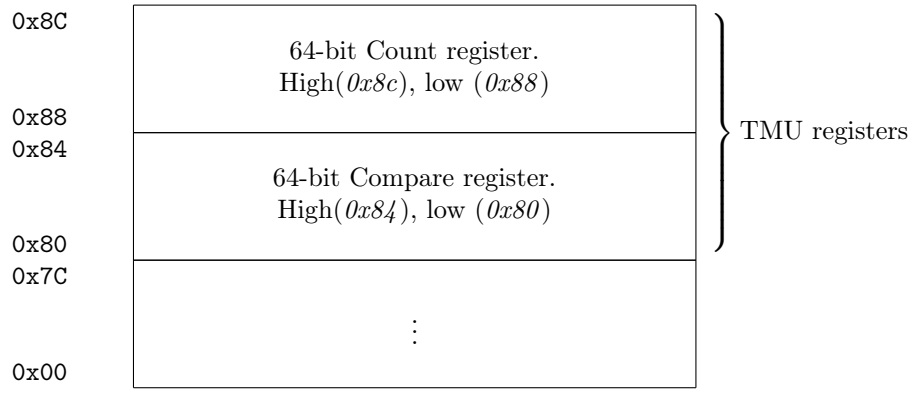



Figure 6.7: Context layout on the stack, with the addition of the TMU registers. Addresses are offset from **pxTopOfStack**, from **portmacro.c**

The context size was set to 144, after the addition of four extra registers. The red-zone was also increased accordingly, in **portmacro.h**. After this change **vPortMiscIntHandler** was no longer able to locate its parameter. Before the change the function stored the parameter on the stack with an offset which accounted for the stack pointer shift in the context switch. Now that the stack pointer was moved 288 instead of 256, the offset for the parameter had to be set to 292 instead of 260. This change was made in **reset.S**

### 6.2.2 xTMUStruct

For easy setup of the TMU a struct named **xTMUStruct** was created, prepended with the *x* to follow FreeRTOS standards, Table 6.4 shows the members, Listing 6.3 shows the struct as c-code. The structure is forward declared in **portable.h** so **pxPortInitialiseStack** would have access to it and at the same time avoid a header declaring error. The error is raised when breaking a circular include raised by trying to include **task.h** in **portable.h**. The compiler is then unable to include **FreeRTOS.h** before **task.h**, even when specifically including **FreeRTOS.h** before all **task.h**.

Listing 6.3: **xTMUStruct**

```
struct xTMUStruct{
  pdEXCEPTION_CODE     vExceptionHandle;
  unsigned long long   ullCompare;
  void                 *pvParameters;      /* if NULL: Set by      */
};                                         /*  vTaskGenericCreate  */
```

**vExceptionHandle** is of type *pdEXCEPTION_CODE*. This type was added to **projdefs.h** to be used as a function pointer for the exception handle. This function pointer can take a void pointer parameter, which allows passing parameters to the exception handle.

Table 6.4: **xTMUStruct**

| Name | Description |
|---|---|
| vExceptionHandle | Function pointer to the user exception routine. |
| ullCompare | Compare value for the task. |
| pvParameters | Parameters set via vTaskCreateTMU. |

Listing 6.4: **pdExceptionCode**

```
typedef void (*pdEXCEPTION_CODE)(void *);
```

Compare is used to initialise the compare value on the task stack. Count is not included in the struct because it is always initialized to zero, since a task always starts with a full budget.

*pvParameters* is used to pass arguments to the exception handle, either specific exception parameters may be passed, or the same **pvParamters** sent to the task. The latter only happens if the TMU-struct with *pvParameters* as NULL is passed to **vTaskGenericCreate**. Having this as a pointer allows for sharing private data between the task and exception function. It is up to the user to add the necessary protection for this data.

### 6.2.3 Task control block

The decleration of *pxCurrentTCB* was moved from **task.c** to **task.h**, and renamed to *xCurrentTCB* since it is no longer private. Moving this handle forces a move of the task control block (TCB) struct, which was also moved to **task.h**, a forward deceleration did not suffice. Moving this handle was necessary to allow access to the task control block currently in the running state. A pointer could be set to point to the same as *xCurrentTCB*, but that would only increase memory size and the pointer would still give write access to the data. If *xCurrenTCB* was written to or the pointer was being manipulated this solution would be preferred, but since the TMU functionality only requires read access, the hazard of using the *xCurrentTCB* pointer is dismissed. In addition to moving the TCB, a new member was added, a pointer to an **xTMUStruct**-struct. To connect the currently running tasks control block, *xCurrentTCB*, with the information required by the TMU.

### 6.2.4 Task creation

Following the already existing task creating system in FreeRTOS, the macro **vTaskCreateTMU** is declared in **task.h**. This function translates into **vTaskGenericCreate**, as described in Section 6.1.4, this function initialises a task. To accommodate the TMU an additional argument was added to **vTaskGenericCreate**, *xTMUStruxt * xTMU*. With this new parameter, **xTMUStruct** can be passed to the task creating process. As a part of the *Initialise TCB*-state in Figure 6.3, the TMU struct is attached to the

**xTMUStruct**-pointer in the TCB. During this state **vTaskGenericCreate** also checks if pvParameters equals NULL, if it does the *pvParameters* sent to the task are attached to the TMU struct. The rest of the function remain unchanged.

**pxPortInitialiseStack** is modified to accept a pointer to a TMU struct, from which it extracts compare and splits this value between the high and low compare addresses. The function also sets the TMU exception bit in the ESR-word on stack and writes zero to the count high/low-address. The reason for sending the task pointer instead of the compare value itself, is to open for future initialisation values, like writing the initial control register or setting a initial count value above zero. This is not done at this point because there is no special circumstance modes in the TMU that would benefit this.

### 6.2.5 TMU exception handling

In the OpenRISC 1000 architecture specification the 15th exception vector is unused, *0xF00*. The exception handle at this address in **reset.S** was changed to work as a starting point for the TMU exception routine, shown in Listings 6.5. The existing tick timer handle was used as an example, since this is the closest categorized exception. Both are *in-between*- and *synchronous*-exceptions.

Listing 6.5: **_except_f00**

```
.org 0xf00
_except_f00:
  l.nop
  l.j    vPortTMUExceptionHandler
  l.nop
```

The handle passes control along to the **vPortTMUExceptionHandler**, shown in Listing 6.6.

Listing 6.6: **vPortTMUExceptionHandler**

```
.text
.global vPortTMUExceptionHandler
.type vPortTMUExceptionHandler, %function
vPortTMUExceptionHandler:
  portRESTART_TMU
  portSAVE_CONTEXT

  l.jal tmu_except
  l.nop

.if configUSE_PREEMPTION == 0
  # do nothing
.else
  l.jal vTaskSwitchContext
  l.nop
.endif

  portRESTORE_CONTEXT
.size vPortTMUExceptionHandler, .-vPortTMUExceptionHandler
```

This function starts by restarting the TMU then it saves the register content using a modified version of the *portSAVE_CONTEXT*-macro.

The **portRESTART_TMU** macro was used to reset the count value after an exception is raised. Writing to count works at this point because the module stops when it reaches a TMU-exception and using the value from **r0** is safe because it is always zero. This action will clear the exception flag from the TMU module. Clearing the count at this time is not a problem because no other module/program depends on the count value. If the user is using it to measure time, the compare value can be used instead of the count value at this point.

Listing 6.7: **portRESTART_TMU**

```
.macro portRESTART_TMU
  l.mtspr r0, r0, SPR_TMU_CNT_L
  l.mtspr r0, r0, SPR_TMU_CNT_H
.endm
```

To aid in the context switch the following macros were created: **portStart_TMU** in Listings 6.8 and **portSTOP_TMU** shown in Listings 6.9.

Listing 6.8: **portSTART_TMU**, starts the TMU

```
.macro portSTART_TMU
  l.ori r3, r0, SPR_TMU_CTRL_START
  l.mtspr r0, r3, SPR_TMU_CTRL
.endm
```

Listing 6.9: **portSTOP_TMU**, stops the TMU

```
.macro portSTOP_TMU
  l.ori r3, r0, SPR_TMU_CTRL_STOP
  l.mtspr r0, r3, SPR_TMU_CTRL
.endm
```

When using these macros great care had to be taken to call them at the right time cause they do not protect the current content of **r3**.

After resetting the counter, **vPortTMUExceptionHandler** calls the **portSAVE_CONTEXT**-macro. This macro was modified to include automatic control of the TMU. The first change to the **portSAVE_CONTEXT**-macro was to have it stop the TMU with the stop-macro, because the context switch should not be counted on the tasks budget. This happens after making room in the stack and saving **r3**.

It is not done sooner because of the previously stated reasons of not protecting the contents of **r3**. Next after saving **r4** and **r5** it saves the essential data from TMU, *count*- and *compare*-values. They are saved before *EPCR* and *ESR* so that the save context routine is still able to provide good cache performance. Continuing from *ESR* the macro remains unchanged.

Listing 6.10: **portSAVE_CONTEXT**

```
.macro portSAVE_CONTEXT
.macro portSAVE_CONTEXT
  .global xCurrentTCB
  # make rooms in stack
  l.addi  r1, r1, -STACKFRAME_SIZE
  # early save r3-r5, these are clobber register
  l.sw  0x08(r1), r3
  portSTOP_TMU
  l.sw  0x0C(r1), r4
  l.sw  0x10(r1), r5

  #SPR_TMU_CMP
  l.mfspr r4, r0, SPR_TMU_CNT_H
  l.mfspr r3, r0, SPR_TMU_CNT_L
  l.sw 0x8C(r1), r4
  l.sw 0x88(r1), r3
  #SPR_TMU_CNT
  l.mfspr r4, r0, SPR_TMU_CMP_H
  l.mfspr r3, r0, SPR_TMU_CMP_L
  l.sw 0x84(r1), r4
  l.sw 0x80(r1), r3

  # save SPR_ESR_BASE(0), SPR_EPCR_BASE(0)
  l.mfspr r3, r0, SPR_ESR_BASE
  l.mfspr r4, r0, SPR_EPCR_BASE
  l.sw  0x78(r1), r3
  l.sw  0x7C(r1), r4

  # Save Context
  l.sw  0x00(r1), r9
  l.sw  0x04(r1), r2
  l.sw  0x14(r1), r6
  l.sw  0x18(r1), r7
  l.sw  0x1C(r1), r8
  l.sw  0x20(r1), r10
       .
       .
       .

  l.sw  0x74(r1), r31
  # Save the top of stack in TCB
  l.movhi r3, hi(xCurrentTCB)
  l.ori r3, r3, lo(xCurrentTCB)
  l.lwz r3, 0x0(r3)
  l.sw  0x0(r3), r1
  # restore clobber register
  l.lwz r3, 0x08(r1)
  l.lwz r4, 0x0C(r1)
  l.lwz r5, 0x10(r1)
.endm
```

First the modified **portRESTORE_CONTEXT**-macro makes sure the TMU is stopped, because compare/count can not be written while the TMU is running. After this the macro gradually unloads the stack, by increasing memory addresses. When it gets

to the point where count and compare is saved these values are sent to SPR, before starting the TMU, restoring the last clobber register, moving the stack register and returning from exception.

Listing 6.11: **portRESTORE_CONTEXT**

```
.macro portRESTORE_CONTEXT
  portSTOP_TMU
  l.movhi r3, hi(xCurrentTCB)
  l.ori r3, r3, lo(xCurrentTCB)
  l.lwz r3, 0x0(r3)
  l.lwz r1, 0x0(r3)

  # restore context
  l.lwz r9,  0x00(r1)
  l.lwz r2,  0x04(r1)
  l.lwz r6,  0x14(r1)
  l.lwz r7,  0x18(r1)
  l.lwz r8,  0x1C(r1)
  l.lwz r10, 0x20(r1)
      .
      .
      .

  l.lwz r31, 0x74(r1)
  # restore SPR_ESR_BASE(0), SPR_EPCR_BASE(0)
  l.lwz r3,  0x78(r1)
  l.lwz r4,  0x7C(r1)
  l.mtspr r0,  r3, SPR_ESR_BASE
  l.mtspr r0,  r4, SPR_EPCR_BASE
  #SPR_TMU_CMP
  l.lwz   r3, 0x80(r1)
  l.lwz   r4, 0x84(r1)
  l.mtspr r0, r3, SPR_TMU_CMP_L
  l.mtspr r0, r4, SPR_TMU_CMP_H
  #SPR_TMU_CNT
  l.lwz   r3, 0x88(r1)
  l.lwz   r4, 0x8c(r1)
  l.mtspr r0, r3, SPR_TMU_CNT_L
  l.mtspr r0, r4, SPR_TMU_CNT_H

  # restore clobber register
  l.lwz r4,  0x0C(r1)
  l.lwz r5,  0x10(r1)

  portSTART_TMU
  # restore clobber register
  l.lwz r3,  0x08(r1)
  l.addi  r1,  r1, STACKFRAME_SIZE
  l.rfe
  l.nop
.endm
```

Besides the context macros, **vPortTMUExceptionHandle** also calls **tmu_except** and **vTaskSwitchContext**, **vTaskSwitchContext** remains unchanged.

**tmu_except** is the function which calls the user exception routine. This uses the function pointer set up in **xTMUStruct**. First it checks if the function pointer is not

*NULL*. If it is not, it calls this function via the pointer and sends along the parameters stored in **xTMUStruct**. If the pointer was *NULL* and the function did not perform this check. Then the program would jump to *0x0000* where it loops upon its own address. In other words, the contents at *0x0000* is *0x0000*, and this instruction is decoded into a jump to the same address. And as with other jump instruction there is a delay of one instruction. So the program counter would alternate between *0x0000* and *0x0004*.

Listing 6.12: **tmu_except**

```
/*TMU interrupt handler */
void tmu_except(void){
  if (xCurrentTCB->xTMU->vExceptionHandle != NULL )
  {
    xCurrentTCB->xTMU->vExceptionHandle(xCurrentTCB->xTMU->pvParameters);
  }
}
```

From **tmu_except** the jump is made to the user function. At this point the user has access to the same *pvParameters* as the task, and if used correctly the exception handle can use this shared memory to correct errors from the task.

### 6.2.6 Critical sections

The *CEE* is added to **vPortEnableInterrupts**/**vPortDisableInterrupts**, this disables the handling of the TMU exception in the processor when **vTaskEnterCritical** is called, and enables when **vTaskExitCritical**.

The addition of this was to avoid a situation where the TMU did not properly stop before entering the scheduler functions, which frequently uses these functions when manipulating state lists. If the scheduler gets interrupted by the TMU, the scheduler could loose tasks. Listings 6.13 shows an example in **vTaskResumeAll** at line 1097 in **task.c**. If the TMU exception is raised after the task is removed from the **xPendingReadyList**, but before it is added to the ready list. This results in the task not being connected to any of the state list and possibly lost. Another example is in **vTaskPlaceOnEventList**.

Listing 6.13: Excerpt from **vTaskResumeAll**, line 1097-1099 in **task.c**

```
1097:    vListRemove( &( pxTCB->xEventListItem ) );
1098:    vListRemove( &( pxTCB->xGenericListItem ) );
1099:    prvAddTaskToReadyQueue( pxTCB );
```

To disable the tick timer and external interrupts while keeping TMU exceptions enabled, the function **vPortDisableInterruptsAndTick** and **vPortDisableInterruptsAndTick** are created. They disable/enable external interrupts and the tick timer. They do not support nesting critical sections.

## 6.3 Setting up FreeRTOS to use the TMU

To use the TMU in FreeRTOS the follow these simple steps.

1. Create a **xTMUStruct**: *struct xTMUStruct xTMU{ };*.

2. Add the function name of the function which should run upon a TMU exception, to the first argument of the struct: {*excFunc, ...*}.
3. Set a compare value as the second argument: {*excFunc, comp, ..*}.
4. Set specific parameters as the third argument, or *NULL* if the same parameters as the task should be passed. {*excFunc, comp, NULL* }.
5. Pass the address of this struct to **vTaskCreateTMU** as the last parameter. The other arguments remain the same as in **vTaskCreate**

## 6.4 Verifying FreeRTOS

During the modifications FreeRTOS was debugged via RSP on Or1ksim. When trying to verify the correct behaviour of the modified context switch a fatal flaw was discovered. The problem was that the general purpose registers was not consistent across context switches. The effect of the error is that the program was sent to a random location in the code when the the program counter reached a jump instruction to a address indicated by the value in the corrupted register. More often than not to a location initialized to zero or into an align exception loop. If the location was initialized to zero the PC then alternate between the current instruction address and the next. The align exception loop was caused by the value of the stack pointer being corrupted and when the align exception tried to read from the stack, this read was issued on a non aligned memory address, thus starting the align loop. After correcting this the context switch performed as expected.

FreeRTOS was used as the platform for testing the TMU behaviour on Or1ksim, so the following test will focus on verifying the changes made to FreeRTOS.

**FreeRTOS test** The test was set up with two tasks alternating between execution. This was to verify that the count and compare values are kept correctly between context switches. The task behavior:

1. Enter critical section and print count and compare to *UART*. Prepend the string with "Start". Exit critical.
2. Busy wait.
3. Enter critical. If no exception occurred, print count and compare to *UART*.Prepend the string with "End". Exit Critical.
4. Yield.

Protecting the uart-printing from exceptions ensures that the strings are printed correctly. It will also demonstrates that exceptions raised during critical sections will be handled immediately upon exiting that critical section. The busy wait is a for-loop which served as a point where exceptions can be issued, while helping to run up the count and also avoid to many switches back and forth.

This test also generates TMU exceptions to verify that the exception handler was set up correctly. And that the appropriate registers were set up correctly at initialization. After two exceptions the exception handler will kill task one. And after task two has generated four exceptions task two will also be killed by the exception handler. After the

two tasks are deleted the idle-task gets control of the processor. The idle-task stops the simulation by calling **ftIdleHook**[2], which was added to the idle-hook function in **main.c**

Using only one exception handler showed that the exception handler correctly got the same *pvParameters* as the task.

**Results** Listings 6.14 shows that count and compare are written correctly, so that task two's runtime got counted on task two's count value. The jump between End and Start count values was because of the time it takes to print to *UART* between retrieving the count values from the TMU. It also showed that the TMU exception is raised when task one exits critical, count is 80 000 more than compare.

Listing 6.14: Results from the FreeRTOS test on Or1ksim

```
Start    Task 2, Compare = 2000000, Count =  58
End      Task 2, Compare = 2000000, Count =  510841
Start    Task 1, Compare = 1000000, Count =  58
End      Task 1, Compare = 1000000, Count =  510301
Start    Task 2, Compare = 2000000, Count =  549322
End      Task 2, Compare = 2000000, Count =  1080397
Start    Task 1, Compare = 1000000, Count =  548794
End      Task 1, Compare = 1000000, Count =  1079346
1-exception
```

Listings 6.15 shows the next part of the simulation were task one gets deleted in the exception handler. From that point Task two is the only task remaining. The simulation ends when task two generates its fourth exception.

Listing 6.15: Results from the FreeRTOS test on Or1ksim

```
Start    Task 2, Compare = 2000000, Count =  1123674
End      Task 2, Compare = 2000000, Count =  1659600
Start    Task 1, Compare = 1000000, Count =  53
End      Task 1, Compare = 1000000, Count =  510301
Start    Task 2, Compare = 2000000, Count =  1703090
2-exception
Start    Task 1, Compare = 1000000, Count =  548789
End      Task 1, Compare = 1000000, Count =  1079337
1-exception
Start    Task 2, Compare = 2000000, Count =  50279
End      Task 2, Compare = 2000000, Count =  576029
Start    Task 2, Compare = 2000000, Count =  614502
End      Task 2, Compare = 2000000, Count =  1145557
Start    Task 2, Compare = 2000000, Count =  1188932
End      Task 2, Compare = 2000000, Count =  1724841
Start    Task 2, Compare = 2000000, Count =  1768344
2-exception
```

The complete output of this test can be seen in Appendix D.2, Listings D.2. How to run this test is explained in the tutorial in Appendix C.5.

[2]ft-prepending signifies that this is a part of FreeRTOS test, it is unrelated to FreeRTOS naming conventions

## 6.5 Discussion

### xTMUStruct

At first it was decided to include a task handle to allow an exception handler to delete its parent task. But after discovering that **vTaskDelete** will delete the calling task when given the parameter *NULL*, the handle was unnecessary. However, it is an important part of what the TMU exceptions can be used for.

**vTaskDelete** can be viewed as a large function, large enough to be undesirable to run from within an exception. An alternative solution to this would be for the user set a global flag for tasks that should be deleted, and a separate task managing this. But there is one significant drawback, if the malfunctioning task has a higher priority than the managing task. The managing task would not get scheduled if the faulting task is stuck in a spin-lock. Another solution would be giving the managing task the highest priority and periodically invoke it. This would still lock the processor until the next time the managing task is scheduled. And in some cases, like a task eating up memory, waiting for the managing task is not an option.

### Context switch changes

**Start- and stop-macros** The decision of implementing the start- and stop-macros without protecting r3 was based on the the goal of adding the least amount of overhead. The protection could easily be added, but that would add two additional instructions to each macro, up to a total of six addition instructions in a complete context switch. And since all of these instructions are memory accesses the worst case additional delay caused by the bus is $6 * 4096 = 24\,570$ clock cycles. The bus access has a time-out which triggers when a twelve bit wide register reaches *0xFFF*(4096).

Considering that achieving worst case response time is rare and the total number of instructions in a complete context switch, adding six more instruction does not seem like a significant amount, but the goal is adding the TMU with minimum overhead. Another option would be to let the compiler determine which register to use. This could be done with inline assembly in *C*, but would require a *l.jal* to the function and a jump to return. This is also two additional instructions, and the compiler might decide that the best approach is to temporary save a register, and by that it adds the delay from the other solution as well as the jumping instructions. For these reasons the best solution was handling the macros with care.

**Restore context race condition** Starting the TMU three instructions prior to returning from the exception can potentially be hazardous. If a task gets preemted with less time in its budget than it takes to execute the last *l.lwz*- and *l.addi*-instructions. Then the next time the task gets runtime its counter would start at **portSTART_TMU**. This will produce a TMU exception between **portSTART_TMU** and the point where *PC* and *SR* is restored via *l.rfe*. If not for the *CEE*-bit in *SR* the processor would start handling the exception from this point. Saving *ESR*, *EPCR* and the context. Most of the context would be correct except r3 and/or r1, depending on where the exception is raised. This would most likely end up in an align exception, if r1 or r3 contains a non-aligned

value, or a loop around *l.rfe* and the *EPCR* from the *l.lwz*- or *l.addi*-instruction. But since SR contains a bit which enables/disables the handling of these exceptions, this race condition is eliminated.

**Complete solution** The **xTMUStruct** along with **vTaskCreateTMU**-macro provides an easy interface for initialising the TMU and connecting user written exception handlers to the TMU exceptions. The context macroes swap in and out count and compare values correctly, and the user does not need concern her/him-self with starting/stopping the counter.

Passing values to functions are mostly done with generic solutions, like sending **xTMUStruct**. This is to make further modifications as easy as possible. For example, if someone in the future has a need for setting an initial count value. This can be done by adding an *unsigned long long* to the **xTMUStruct** and then adding this value to the correct address in **pxPortInitialiseStack**.

By changing the exception handle at *0xF00* and adding a new path for the TMU exception helps separate the TMU exception functionality from the rest of the system, making errors in exception handling less likely. This is a more direct solution for TMU exceptions, which means it provides a quicker response time for the user written exception function. Compared to calling the user function from **res2_except**, which requires FreeRTOS to decode the exception vector in **misc_int_handler**.

Stopping the TMU counter at the start of both the **portSAVE_CONTEXT**-macro and the **portRESTORE_CONTEXT**-macro is added to ensure that the TMU is stopped before the context switch macro tries to write to it. This is not really necessary if the count during exception bit is not set in the TMU status register. If the two stop macros were removed it would save four instructions each context switch. The total number of instructions during a context switch is significantly higher, approximately 100 instructions plus the **vTaskSwitchContext**-function and all its calls. They also serve as an extra precaution to make sure the TMU is stopped correctly. Considering the total amount of instructions this is an acceptable trade-off. Should the user want to change this, it can be done by removing calls to the the macro in the context macros in **portasm.S**.

The FPP scheduling policy in FreeRTOS does not directly benefit from the TMU, other than the fact that it might be called at a higher rate. But the simplicity of FreeRTOS made making the necessary modifications easier and FreeRTOS provided a good base to create programs for verification. Through applications, FreeRTOS is still able to help produce results that proves the behaviour of the TMU and what effects it can have in more complex systems.

Chapter 7

# Testing the full system

## 7.1 Equipment

The FPGA development kit used to run the full system is a Terasic Cyclone V GX starter kit with an Altera Cyclone V FPGA, seen in Figure 7.1. This board was chosen because it was available, had enough embedded memory for FreeRTOS and because it uses the Altera USBBlaster I for JTAG communication[16]. As opposed to the Altera's USBBlaster II, version I is supported by the JTAG bridge software used to communicate with the system debugger. Some of the most relevant features of the Cyclone V GX starter kit is:

- 77 000 programmable logic elements
- 4884 Kbits embedded memory
- 18 LEDs
- 4 de-bounced push buttons
- 1 CPU reset push button

## 7.2 Resource usage

Some relevant number from the Quartus compilation report are examined to determine the cost of adding the TMU in terms physical resources. Table 7.1 shows the resource usage of the TMU for the full system and for the standalone unit. In this table the full system values includes both the resources from the TMU and other parts of the system. The number of adaptive logic modules (ALM) is the total number of blocks needed and the number of registers shows how many of the ALMs that are used as registers.

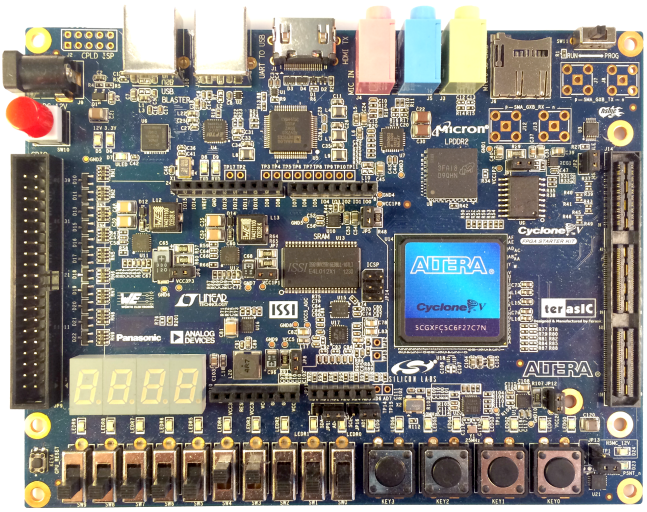



Figure 7.1: Cyclone V GX starter kit

Table 7.1: TMU resource usage

| | **Parameter** | **Value** |
|---|---|---|
| Full system without TMU | Adaptive logic modules | 3 296 |
| | Registers | 2 609 |
| Full system with TMU | Adaptive logic modules | 3 842 |
| | Registers | 3 160 |
| Increase | Adaptive logic modules | 16.57% |
| | Registers | 21.12% |
| Only TMU | Logic cells combinationals | 520 |
| | Logic cells registers | 497 |

## 7.3 TMU functionality test

### 7.3.1 Test setup

The functionality test was written using FreeRTOS and contains three running tasks and two interrupt sources. It was constructed to be dependent on the correct operation of the TMU in order to complete its execution, and to show how the TMU provides security in systems that contains tasks with critical sections which disables external interrupts and tick handling.

The test has three running tasks, which each has a 10% chance of failing in its critical section. If a task fails, it will enter an infinite loop ignoring both ticks and external interrupts. If a task does not fail, it will be suspended until the next tick. Each of the tasks increments a counter every time it is started and the number of executions for each task will be reported when the test is completed. When a task reaches its limit and the TMU exception is handled, a separate task exception counter is incremented to indicate how many times a task has failed.

To test the interrupt masking functionality, the two interrupt generators will get a low budget for the number of interrupt arrivals. The replenishment budget will be set to a much higher value than the tick period to make sure that interrupts are masked. When running on the FPGA, the interrupt line of each of the generators is mapped to a LED, which will light up when an interrupt is pending on that source. To make it possible to observe this, the replenishment period is set much lower than it would be in a real real-time system. Each time an interrupt is handled, a counter associated with the interrupt source will be incremented, to indicate how many times this interrupt has arrived. The time-out for an interrupt generator will be written by the interrupt handler, and is set to a random value. The maximum time-out value for the interrupt generators are tested with different values to examine the effects of fewer interrupts on the system.

Table 7.2 shows all the configuration parameters of the test. The limit for the tasks and replenishment of interrupts in Table 7.2 is given in clock cycles. The cycles per tick is found by Equation 7.1.

$$Cycles\,per\,tick : t_t = \frac{f_c}{f_t} \tag{7.1}$$

Where $t_t$ is the number of cycles per tick, $f_c$ is the clock frequency and $f_t$ is the tick frequency.

During the execution of the test, no information will be printed. This is done because printing to UART is a very time consuming operation. Instead the execution statistics are printed when the test is completed.

The functionality test was executed on Or1ksim and the FPGA. Four cases were examined, both with and without a critical section guarding the spin lock from external/tick interrupts.

- Interrupt time-out limit of 65 535
- Interrupt time-out limit of 65 535, with seeding of the random number generator
- Interrupt time-out limit of 268 435 455

Table 7.2: Test parameters

| **Paramter** | **Value** |
|---|---|
| Clock frequency[MHz] | 50 |
| Ticks to run | 20 000 |
| Tick frequency[Hz] | 100 |
| Cycles per tick | 500 000 |
| Replenishment limit | 5 000 000 |
| Interrupt 0 limit | 2 |
| Interrupt 1 limit | 2 |
| Task 1 limit | 1 000 000 |
| Task 2 limit | 1 500 000 |
| Task 3 limit | 2 500 000 |
| Interrupt timer max | 65 535<br>268 435 455 |

- Interrupts disabled

The test is also performed on the system, but without the TMU. This is achieved by compiling and synthesizing the processor without defining **OR1200_TMU_IMPLEMENTED** in *or1200_defines.v*. Since the test would never complete with tasks that have a critical section, this case is omitted when the TMU is removed from the system.

### 7.3.2 Results

#### Expected results

Running the test with the TMU enabled and a low maximum time-out for the interrupt generators, it is expected that the tasks will get more runtime than the interrupt handlers. The number of generated TMU exceptions should be ten percent per task. By allowing two interrupts per replenishment period for each source, the expected number of interrupts are about 4 000 interrupts being handled for each source, $20\,000/2 * 2 = 4\,000$.

When running the test without the TMU, the tasks are expected to get stuck in the spin lock. This test is to show how a system without the TMU solution will react under the same circumstances. The expected amount of interrupts when no TMU is used are calculated using Equations 7.1-7.3. Using the parameters from Table 7.2 the expected number of interrupts would be interrupt $E_I = 305\,180$ per source.

$$Expected\,timeout : E_{to} = \frac{Maximum\,timeout}{2} \tag{7.2}$$

$$Expected\,interrupts : N_I = \frac{\Delta t_t}{E_{to}} * N_t \tag{7.3}$$

### Simulation on Icarus Verilog

When the test is run on simulated hardware using Icarus Verilog, the tick count had to be decreased because the simulation time is very long. It takes approximately five hours to simulate 100 ticks on a Linux system with an Intel Xeon CPU clocked at 2.66 GHZ. The number of ticks to run is lowered to 200 for this simulation. The simulation results from Icarus Verilog is included as verification of the correct operation of the TMU and FreeRTOS, and can be seen in Table 7.3.

Table 7.3: Icarus verilog result, 200 ticks

| | **Rounds** | **Exceptions** | **%** |
|---|---|---|---|
| Task 1 | 167 | 16 | 9.6 |
| Task 2 | 165 | 19 | 11.5 |
| Task 3 | 164 | 21 | 12.8 |
| Interrupt 0 | 54 | | |
| Interrupt 1 | 55 | | |

### Or1ksim

The result from running the tests on Or1ksim is shown in Table 7.4, 7.5 and 7.6.

Table 7.4: Or1ksim test results, with TMU and critical section

| **Interrupt time-out** | **Parameter** | **Value** | | | | |
|---|---|---|---|---|---|---|
| | | Task 1 | Task 2 | Task 3 | Interrupt 0 | Interrupt 1 |
| 65535 | Rounds | 16 655 | 16 612 | 16 661 | 4 630 | 4 647 |
| | Exceptions | 1 681 | 1 696 | 1 639 | | |
| | % | 10.09 | 10.21 | 9.84 | | |
| 65 535 seeded | Rounds | 16 663 | 16 696 | 16 690 | 4 649 | 4 667 |
| | Exceptions | 1 694 | 1 677 | 1 632 | | |
| | % | 10.11 | 9.91 | 10.41 | | |
| 268 435 455 | Rounds | 16 611 | 16 618 | 16 594 | 139 | 92 |
| | Exceptions | 1 652 | 1 680 | 1 696 | | |
| | % | 9.95 | 10.11 | 10.22 | | |
| No interrupts | Rounds | 16 597 | 16 605 | 16 597 | 0 | 0 |
| | Exceptions | 1 662 | 1 653 | 1 715 | | |
| | % | 10.01 | 9.95 | 10.33 | | |

Table 7.5: Or1ksim test results, with TMU, without critical section

| **Interrupt time-out** | **Parameter** | **Value** | | | | |
|---|---|---|---|---|---|---|
| | | Task 1 | Task 2 | Task 3 | Interrupt 0 | Interrupt 1 |
| 65 535 | Rounds | 14 410 | 15 032 | 14 252 | 3 928 | 3 937 |
| | Exceptions | 1 457 | 1 489 | 1 484 | | |
| | % | 10.11 | 9.91 | 10.41 | | |
| 65 535 seeded | Rounds | 14 559 | 15 085 | 14 350 | 3 932 | 3 944 |
| | Exceptions | 1 408 | 1 528 | 1 448 | | |
| | % | 9.67 | 10.13 | 10.09 | | |
| 268 435 455 | Rounds | 14 321 | 14 955 | 14 296 | 72 | 183 |
| | Exceptions | 1 459 | 1 515 | 1 402 | | |
| | % | 10.19 | 10.13 | 9.81 | | |
| No interrupts | Rounds | 14 408 | 14 931 | 14 173 | 0 | 0 |
| | Exceptions | 1 388 | 1 517 | 1 471 | | |
| | % | 9.63 | 10.16 | 10.38 | | |

Based on the results in Table 7.4 and 7.5 a second test was created with Or1ksim. Here some minor modifications were done to the interrupt generator to make it send out interrupts without the need for receiving a new time-out value through the bus. The time-out is generated as in test one, by taking a random number between 0 and 65 535. The results are shown in Table 7.7. Test one was also run on the same set-up for reference, Table 7.9 The ignored interrupt number are collected from the counter explained in Section 5.2.3.

Moving the write back of *PICSR* closer to reading it, during **int_main**, does not break the loop for both interrupts during the initial main test, Table 7.8. The same behavior can be seen when increasing the minimum time-out to 5000, Table 7.10

### FPGA execution

During the execution of the test on FPGA the LEDs, assigned to the interrupts lines, were observed to be on for long periods, indicating blocked interrupts. The results of the test execution on FPGA is found in Table 7.11, 7.12 and 7.13. Where Table 7.11 shows the execution with TMU included in the processor and where each task has a critical section. Table 7.12 shows the results when the TMU is included, but the tasks does not have a critical section, and Table 7.13 shows the results when the TMU is not included in the processor and the tasks have no critical section.

Table 7.6: Or1ksim test results, without TMU and critical section

| **Interrupt time-out** | **Parameter** | **Value** | | | | |
|---|---|---|---|---|---|---|
| | | Task 1 | Task 2 | Task 3 | Interrupt 0 | Interrupt 1 |
| 65 535 | Rounds | 1 | 4 | 6 | 18 | 166 |
| | Exceptions | 0 | 0 | 0 | | |
| | % | 0 | 0 | 0 | | |
| 65 535 seeded | Rounds | 10 | 4 | 26 | 14 | 313 |
| | Exceptions | 0 | 0 | 0 | | |
| | % | 0 | 0 | 0 | | |
| 268 435 455 | Rounds | 9 | 1 | 2 | 70 | 73 |
| | Exceptions | 0 | 0 | 0 | | |
| | % | | 0 | 0 | | |
| No interrupts | Rounds | 9 | 1 | 2 | 0 | 0 |
| | Exceptions | 0 | 0 | 0 | | |
| | % | 0 | 0 | 0 | | |

Table 7.7: Or1ksim result, second round: TMU disabled, short interrupt period, 20 000 ticks

| | **Rounds** | **Exceptions** |
|---|---|---|
| Task 1 | 2 | |
| Task 2 | 9 | |
| Task 3 | 1 | |
| Interrupt 0 | 293 137 | |
| Interrupt 1 | 293 235 | |
| Ignored Int0 | 1 690 | |
| Ignored Int1 | 4 652 | |

Table 7.8: Or1ksim result: TMU disabled, short interrupt period, *PICSR* write back moved, 20 000 ticks

| | **Rounds** | **Exceptions** | **%** |
|---|---|---|---|
| Task 1 | 1 | | |
| Task 2 | 11 | | |
| Task 3 | 13 | | |
| Interrupt 0 | 297 254 | | |
| Interrupt 1 | 358 | | |
| Ignored Int0 | 0 | | |
| Ignored Int1 | 0 | | |

Table 7.9: Or1ksim result, second round: TMU enabled, short interrupt period, 20 000 ticks

| | **Rounds** | **Exceptions** | **%** |
|---|---|---|---|
| Task 1 | 16 611 | 1 732 | 10.4 |
| Task 2 | 16 590 | 1 640 | 9.9 |
| Task 3 | 16 604 | 1 658 | 10 |
| Interrupt 0 | 3 814 | | |
| Interrupt 1 | 3 818 | | |
| Ignored Int0 | 869 | | |
| Ignored Int1 | 913 | | |

Table 7.10: Or1ksim test results, without TMU and critical section, longer minimal interrupt time-out

| **Interrupt time-out** | **Parameter** | **Value** | | | | |
|---|---|---|---|---|---|---|
| | | Task 1 | Task 2 | Task 3 | Interrupt 0 | Interrupt 1 |
| 65 535 | Rounds | 1 | 5 | 3 | 258 862 | 4 |
| | Exceptions | 0 | 0 | 0 | | |
| | % | 0 | 0 | 0 | | |

Table 7.11: FPGA test results, with TMU and critical section

| **Interrupt time-out** | **Parameter** | **Value** | | | | |
|---|---|---|---|---|---|---|
| | | Task 1 | Task 2 | Task 3 | Interrupt 0 | Interrupt 1 |
| 65 535 | Rounds | 16 633 | 16 594 | 16 633 | 5 540 | 5 450 |
| | Exceptions | 1 670 | 1 675 | 1 705 | | |
| | % | 10.04 | 10.09 | 10.25 | | |
| 65 535 seeded | Rounds | 16 645 | 16 655 | 16 645 | 5 497 | 5 405 |
| | Exceptions | 1 678 | 1 621 | 1 669 | | |
| | % | 10.08 | 9.73 | 10.03 | | |
| 268 435 455 | Rounds | 16 682 | 16 649 | 16 671 | 85 | 80 |
| | Exceptions | 1 661 | 1 671 | 1 679 | | |
| | % | 9.96 | 10.04 | 10.07 | | |
| No interrupts | Rounds | 16 642 | 16 646 | 16 657 | 0 | 0 |
| | Exceptions | 1 679 | 1 654 | 1 680 | | |
| | % | 10.09 | 9.94 | 10.09 | | |

Table 7.12: FPGA test results, with TMU, without critical section

| **Interrupt time-out** | **Parameter** | **Value** | | | | |
|---|---|---|---|---|---|---|
| | | Task 1 | Task 2 | Task 3 | Interrupt 0 | Interrupt 1 |
| 65 535 | Rounds | 14 491 | 15 117 | 14 307 | 4 750 | 4 683 |
| | Exceptions | 1 440 | 1 475 | 1 483 | | |
| | % | 9.94 | 9.76 | 10.37 | | |
| 65 535 seeded | Rounds | 14 501 | 15 008 | 14 285 | 4 672 | 4 649 |
| | Exceptions | 1 414 | 1 574 | 1 446 | | |
| | % | 9.75 | 10.49 | 10.12 | | |
| 268 435 455 | Rounds | 14 391 | 15 007 | 14 271 | 78 | 82 |
| | Exceptions | 1 426 | 1 491 | 1 452 | | |
| | % | 9.91 | 9.94 | 10.17 | | |
| No interrupts | Rounds | 14 364 | 14 987 | 14 288 | 0 | 0 |
| | Exceptions | 1 443 | 1 509 | 1 421 | | |
| | % | 10.05 | 10.07 | 9.95 | | |

Table 7.13: FPGA test results, without TMU and critical section

| **Interrupt time-out** | **Parameter** | **Value** | | | | |
|---|---|---|---|---|---|---|
| | | Task 1 | Task 2 | Task 3 | Interrupt 0 | Interrupt 1 |
| 65 535 | Rounds | 1 | 6 | 10 | 23 894 | 20 910 |
| | Exceptions | 0 | 0 | 0 | | |
| | % | 0 | 0 | 0 | | |
| 65 535 seeded | Rounds | 16 | 44 | 14 | 59 802 | 21 637 |
| | Exceptions | 0 | 0 | 0 | | |
| | % | 0 | 0 | 0 | | |
| 268 435 455 | Rounds | 16 | 5 | 18 | 74 | 67 |
| | Exceptions | 0 | 0 | 0 | | |
| | % | | 0 | 0 | | |
| No interrupts | Rounds | 16 | 5 | 18 | 0 | 0 |
| | Exceptions | 0 | 0 | 0 | | |
| | % | 0 | 0 | 0 | | |

Table 7.14: FPGA test results, without TMU and critical section, longer minimal interrupt time-out

| **Interrupt time-out** | **Parameter** | **Value** | | | | |
|---|---|---|---|---|---|---|
| | | Task 1 | Task 2 | Task 3 | Interrupt 0 | Interrupt 1 |
| 65 535 | Rounds | 8 | 6 | 34 | 231 315 | 230 333 |
| | Exceptions | 0 | 0 | 0 | | |
| | % | 0 | 0 | 0 | | |

Table 7.15: Overhead test parameters

| Parameter | Value |
|---|---|
| Clock frequency[MHz] | 50 |
| Ticks to run | 20 000 |
| Tick frequency[Hz] | 100 |
| Cycles per tick | 500 000 |

## 7.4 Overhead

### 7.4.1 Testing TMU overhead

A simple test was devised to test the overhead introduced by using the TMU in a context switch. The modified context switch has an additional 22 instructions to accommodate the TMU. The test is a single running task which increments a value in a loop for a fixed number of ticks, and reports this number when the test is done. The overhead can be calculated from the reported numbers by running the test with and without the TMU. Table 7.15 shows the parameters for the test. During this test the TMU will not generate any exceptions.

The executing loop was analysed to determine how many clock cycles it took to execute. To make this calculation easier, both instruction- and data-cache were disabled during the execution of this test. Disabling caches also has the benefit of the resulted overhead representing the absolute worst case, when instructions and data has to be loaded from memory for each instruction. through simulation the fetch of an instruction was found to be four cycles, and the memory access time for read/write instructions was found to be four cycles. The execution time for the different instructions is given in Table 2.1.

The assembly code for the execution loop is shown in Listing 7.1 and is compiled from the C-code in Listing 7.2. The analysis of each instructions execution time is shown in Table 7.16.

Table 7.16: Execution time analysis for the overhead loop

| Instruction | I-Bus cyc. | D-Bus cyc. | Ex-cyc. | Total |
|---|---|---|---|---|
| l.lwz r2,0x4(r1) | 4 | 4 | 1 | 9 |
| l.addi r2,r2,0xffffffff | 4 | 0 | 1 | 5 |
| l.movhi r4,0x1 | 4 | 0 | 1 | 5 |
| l.ori r4,r4,0x6cb8 | 4 | 0 | 1 | 5 |
| l.slli r3,r2,0x2 | 4 | 0 | 1 | 5 |
| l.add r3,r4,r3 | 4 | 0 | 1 | 5 |
| l.lwz r3,0x0(r3) | 4 | 4 | 1 | 9 |
| l.addi r3,r3,0x1 | 4 | 0 | 1 | 5 |
| l.movhi r4,0x1 | 4 | 0 | 1 | 5 |
| l.ori r4,r4,0x6cb8 | 4 | 0 | 1 | 5 |
| l.slli r2,r2,0x2 | 4 | 0 | 1 | 5 |
| l.add r2,r4,r2 | 4 | 0 | 1 | 5 |
| l.sw 0x0(r2),r3 | 4 | 4 | 1 | 9 |
| l.j 6fdc | 4 | 0 | 1 | 5 |
| l.nop | 4 | 0 | 1 | 5 |
| Total cycle count | | | | 87 |

Listing 7.1: Overhead test function loop assembly

```
l.lwz r2,0x4(r1)
l.addi r2,r2,0xffffffff
l.movhi r4,0x1
l.ori r4,r4,0x6cb8
l.slli r3,r2,0x2
l.add r3,r4,r3
l.lwz r3,0x0(r3)
l.addi r3,r3,0x1
l.movhi r4,0x1
l.ori r4,r4,0x6cb8
l.slli r2,r2,0x2
l.add r2,r4,r2
l.sw 0x0(r2),r3
l.j 6fdc <ohTaskFunction+0x14>
l.nop
```

Listing 7.2: Overhead test function loop C

```
while (1) {
  ohTaskCount++;
}
```

Some key values are calculated based on the total number of instructions in the loop. The average difference in value per tick from Equation 7.4 and the number of overhead

clock cycles per tick from Equation 7.5.

$$diff_{tick} = \frac{Count_{NoTMU} - Count_{TMU}}{N_{Ticks}} \tag{7.4}$$

Where $diff_{tick}$ is the difference in loop execution per tick, $N_{ticks}$ is the number of ticks and the $Count$ values are the value of the counter when execution is complete.

$$cycle_{overhead} = diff_{tick} \cdot cycles_{loop} \tag{7.5}$$

Where $cycle_{overhead}$ is cycles added for each tick and $cycles_{loop}$ is the number of cycles for one iteration of the counter loop.

### 7.4.2 Overhead test results

Table 7.17 shows the results of the overhead test for the system. In the table the difference per tick is calculated according to Equation 7.4 and delayed cycles per tick is calculated according to Equation 7.5.

Table 7.17: Overhead test results

| **Parameter** | **Value** |
|---|---|
| Count no TMU | 115 934 630 |
| Count TMU | 115 904 472 |
| Difference/tick | 1.51 |
| Delayed cycles/tick | 131.19 |

The test is also executed with instruction- and data-caches included in the system, but in this case it is impossible to calculate the number of cycles used by the counting loop. Table 7.18 shows the results of the overhead test when caches are used.

Table 7.18: Overhead test results

| **Parameter** | **Value** |
|---|---|
| Count no TMU | 587 168 191 |
| Count TMU | 587 105 985 |
| Difference/tick | 3.11 |

## 7.5 Processor utilization

To show how the TMU can help in processor utilization the following test is created. This test models the behaviour of a simple task which usually finishes quickly, but in some

cases it will get overworked and not finish in time, e.g. a calculating task gets input that are too large or otherwise requires to much work. An upper limit can be set by using the TMU, this will preempt the task if it reaches this limit, allowing the task to reset itself and allow more important work to be done.

To verify that the TMU can consistently preempt tasks when they deplete their budget two tasks are created, each will fail at a given failure rate, this is implemented by using a counter because using **rand** makes calculating WCET a lot harder.

The test behaviour is shown in Figure 7.2, **dualtask.c** contains the source code for this test.

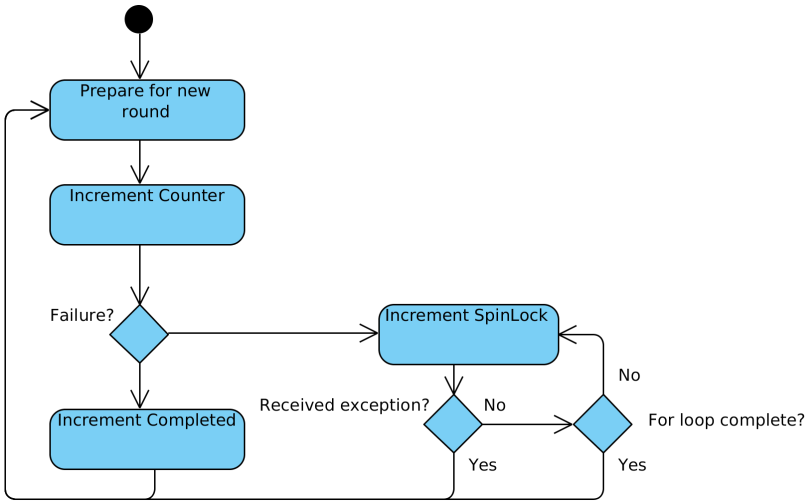



Figure 7.2: Processor utilization test

These task first prepares for the new round by resetting the TMU counter and setting the TMU exception help flag to zero, then they increment the failure counter and checks if it is time for a failure. If the tasks are suppose to fail they enter a for-loop which increments the loop counter, up to 50 times. If the *ExceptionHelp*-flag is raised during the loop, it will exit and return to the start of the main loop. The completed counter is only incremented when a task does not fail.

### 7.5.1 Expected results

Calculating the WCET of the main work of the task is done by accumulating the number of cycles each instruction uses. The results are shown in Table 7.19, addresses are read from the disassembly file produced after compilation. The for-loop trap is excluded from the calculation.

During debugging the delay per instruction for Or1ksim was observed to be one cycle per non-load/store-instruction. Load instructions required one additional cycle, while store instructions needed two.

To calculate the execution time on the FPGA the values from the overhead test are used, Section 7.4.

Table 7.19: Instructions per functions. Derived from **rtosdemod.asm** in Appendif B

| **Function** | **Start** | **End** | **#Instructions** | **#Load** | **#Store** |
|---|---|---|---|---|---|
| Duaktask while-loop | 6B58 | 6CC8 | 93 | 15 | 7 |
| Failure trap | 6BD0 | 6C64 | 38 | 6 | 3 |
| tmu_restart | 7B20 | 7B58 | 15 | 2 | 2 |
| tmu_set_control | 7D18 | 7D4C | 14 | 3 | 3 |
| mtspr | 6FFC | 7028 | 12 | 3 | 3 |
| Total Instruction count one iteration | | **SUM** | 94 | 17 | 12 |

From the TMU's point of view the loop starts when **tmu_restart** writes *restart* to the control-register, and ends when **tmu_restart** writes *stop*. Therefore the compare value must be the sum of instruction executed between these two writes. Table 7.19 shows how many instructions each function has. **tmu_restart** calls **tmu_set_control** and **mtspr** twice each loop iteration. The first call to these function stops the TMU, the second starts it. Because this is done with the same functions, the instructions between these to writes equals to the instruction count for **mtspr** and **tmu_set_control** and the number of instructions between the two calls to **tmu_set_control** in **tmu_restart**, which is two.

When setting the compare value the starting offset, four instructions, should be considered. This is the number of instructions between issuing the *mtspr*-instruction which starts the TMU and the *l.nop* after *l.rfe*. But this additional delay is only related to context switches. Since the main loop in this test does not include a yield.

**Or1ksim** The expected execution time on Or1ksim is $94+17+12*2 = 135$ cycles. This is using the observed delay for load and store, which will vary depending on the workload for the bus at any given time.

The calculations indicate that the lowest amount of for-loop iterations should be with a compare value at 135. With values higher than 135 the results should show that the time spent in the for-loop increases upto 50 times the failure rate.

Below 135, exceptions will be raised every iteration. This will be the case until a compare value of 68 where the number of exceptions should be twice as many as the number of started tasks. This happens because the exception trigger has time to create multiple exceptions during the execution time of one round in the task. Time spent in the for-loop will not increase because the for-loop will be broken by an exception, and at one point the for-loop will break before incrementing the counter, because the flag will be set prior to evaluating it at the start of the for-loop.

The for-loop contains 32 instruction total, where six are load instructions and two are store instructions. Prior to the loop there are five normal instructions and two store instruction. This means that the for-loop needs $135+5+2*2+50*(32+6*1+2*2) = 2\,244$ cycles to complete without generating an exception.

**FPGA** The expected cycle count for the loop on the FPGA is 586, $94*5+17*4+12*4 = 586$, five cycles per instruction, 4 additional cycles for load/store. For the FPGA the loop counter should stop counting at around $586+5*5+2*4+50*(32*5+6*4+2*4) = 10\,219$ .The overall behaviour is expected to be the same as with Or1ksim.

### Results

All the results are shown in **dualtask.xls** in the dualtask folder in Appendix B. Sheet one is for Or1ksim, sheet two is for FPGA.

**Or1ksim** Figure 7.3 shows how many times a task started versus how many iterations the for-loop achieved plotted against the compare value. As the compare value decreases the iterations by the for-loop also decreases and the amount of task starts increases. The figure is based on the numbers from the tables in Appendix D.4.

The overall behaviour is as expected. The expected value of 135 does yield the lowest amount of time spent in the for-loop, but it gives a higher than requested exception rate. The lowest compare value that did not yield an exception rate above 1% was 300.

The expected rise in exceptions happened at a compare value of 134-135. The same increase happened again at 61-62, the magnitude was as expected, but the second rise happened at a slightly lower value than expected. From 2 332 and above the for-loop managed to finish within the time limit, and the task was restarted without the TMU raising any exceptions. This was expected to happen a earlier, at 2 244.

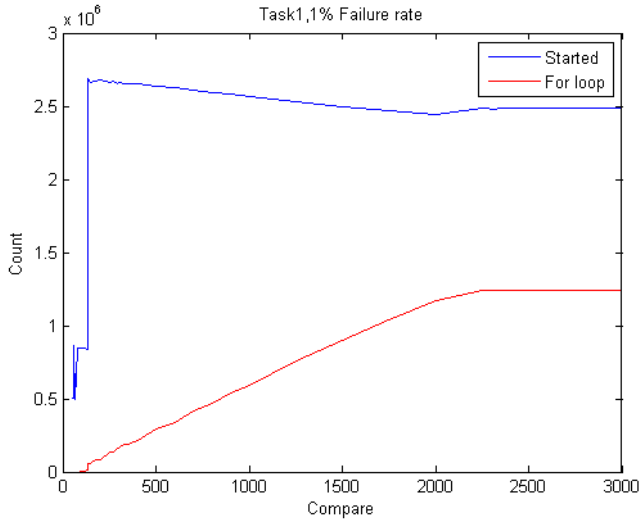
Comparison between the different failure rates show that the results and overall behaviour is the same, see Appendix D.3 for all the figures.

Table 7.20 shows the results from the test. By using the expected cycle count(135) as the compare value the processor utilization increased by 8.02% and the time spent doing the unwanted work was decreased by 89.15%. For a more conservative compare value set at twice the expected runtime, 270, the same numbers went slightly down, 7.41% and 89.27% respectively. With the lowest value which did not produce extra exceptions the utilisation was 7.26% better and it allowed 87.15% less time in the for-loop. All values are compared to running the test without the TMU enabled.

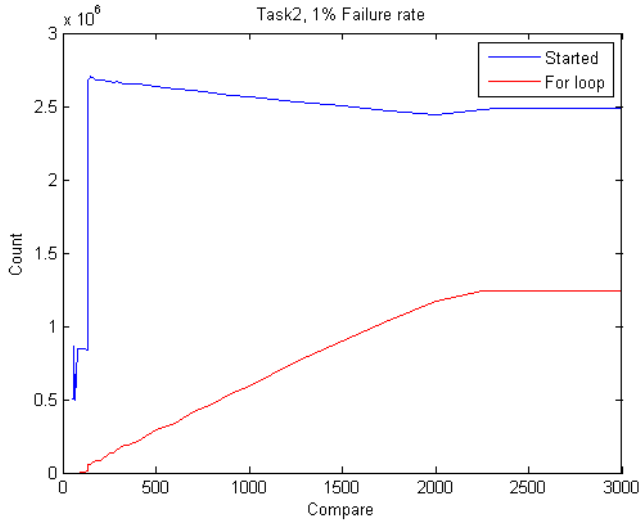
Table 7.20: Processor utilization results, Or1ksim

| **Compare** | **Task one** | | **Task two** | | **Average** | |
|---|---|---|---|---|---|---|
| | **Util.[%]** | **Loop [%]** | **Util.[%]** | **Loop[%]** | **Util.[%]** | **Loop[%]** |
| 135 | 8.26 | 95.69 | 7.77 | 95.69 | 8.02 | 95.69 |
| 270 | 7.55 | 89.27 | 7.27 | 89.27 | 7.41 | 89.27 |
| 300 | 7.26 | 87.15 | 7.14 | 87.15 | 7.2 | 87.15 |

**FPGA** For the FPGA the expected value of 586 yielded a low for-loop time, the difference down to the measured best value is negligible. The best result were with a compare value of 579. The exception rate was even more unstable on the FPGA than in




(a) Task one




(b) Task two

Figure 7.3: Task started versus time spent in the for-loop, failure rate 1%, Or1ksim

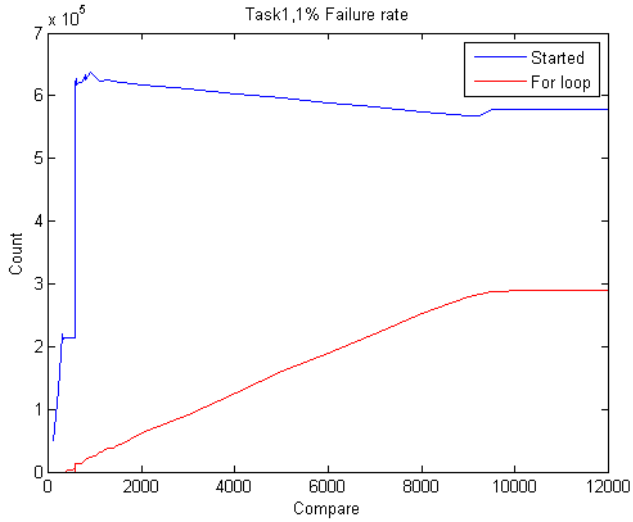
Or1ksim. The lowest compare value which finished with 1% exception rate was 1 400. Between a compare value of 9 500 and 10 000 the started to finish every iteration.

Figure 7.4 shows the results for FPGA for how many times the task started versus time spent in the for-loop. These results also show the drop in started tasks just below the best compare value.

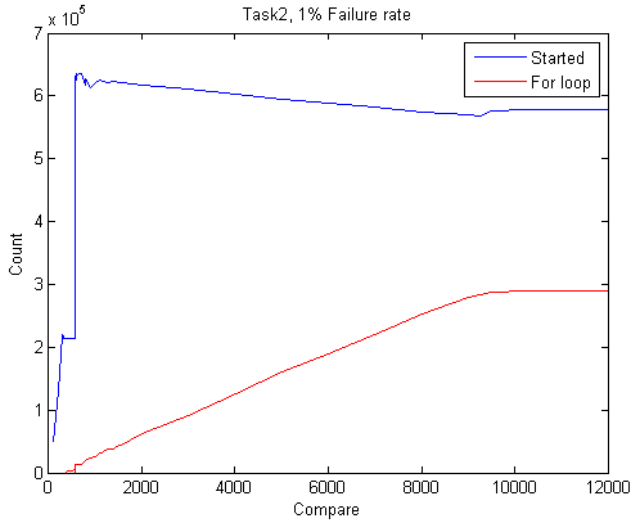
Table 7.21 shows the utilization results from the test run on the FPGA. This resulted in slightly better results, compared to running the test on Or1ksim.

Table 7.21: Processor utilization results, FPGA

| **Compare** | **Task one** | | **Task two** | | **Average** | |
|---|---|---|---|---|---|---|
| | **Util.[%]** | **Loop [%]** | **Util.[%]** | **Loop[%]** | **Util.[%]** | **Loop[%]** |
| 579 | 7.63 | 95.63 | 9.20 | 95.63 | 8.42 | 95.63 |
| 586 | 6.78 | 95.62 | 9.94 | 95.60 | 8.36 | 95.61 |
| 1 158 | 8.09 | 89.23 | 7.95 | 95.62 | 8.02 | 92.42 |
| 1 172 | 7.30 | 89.19 | 8.19 | 95.61 | 7.75 | 92.40 |
| 1 400 | 7.84 | 87.06 | 7.86 | 87.06 | 7.85 | 87.06 |




(a) Task one




(b) Task two

Figure 7.4: Task started versus time spent in the for-loop, failure rate 1%, FPGA

Chapter 8

# Discussion

## 8.1 TMU implementation and integration

Sections 3.4 and 4.1 shows how the TMU is designed and integrated into OR1200. The TMU is basically a simple counter module with the ability to generate exceptions and mask interrupts. Even though the operation of the TMU is simple in many ways, it could potentially lead to erroneous behaviour when it is integrated into a large and complex system. All the changes listed in Section 4.1 shows that this placement of the module is highly intrusive, and should be part of the initial design. Because of the flexible design of the OR1200 processor it was still possible to integrate an extra module, with relative ease through the use of the existing special-purpose register interface.

The results shown in Section 4.5 shows that the TMU is implemented successfully in OR1200, and that it is able to operate as a part of the system without any unintended side-effects. The overhead related to switching tasks is strongly related to the context switch implementation, but the lowest possible overhead would be seven instructions from where the top value of *compare* is loaded to the timer is started as shown in Figure 3.9. The overhead related to acknowledging the exception generated by the TMU is the time it takes to write to the *count* register, and is two instructions. Stopping the TMU and reading the *count* register can be done with six instructions. For the counting and masking of interrupts there will be no overhead once the limits for each interrupt line and the replenishment period is loaded.

From Table 7.1 one can see that adding the TMU to the system increases the total usage of resources by 16.57%. The large increase in area is because the TMU implements its registers in the FPGA's adaptive logic modules, whereas other large register modules, like the register file, uses megafunctions on the FPGA. Since the full system only uses 3 862 of the 29 080 available ALMs on the FPGA no effort has been put into reducing the resource usage of the TMU.

## 8.2 Functionality test

**Overall** The results produced by running FreeRTOS on the FPGA shown in Tables 7.11 and 7.12 indicates that the TMU is functioning properly inside the processor, and that FreeRTOS is able to use the functionality it provides. If the TMU had not been functioning correctly, the number of task iterations would be around ten and the exception counters would have been set to zero, as shown in Table 7.13.

Each of the three tasks are executed almost the same number of times, and the fail rate is almost 10%. The difference in the specified fail rate and the reported can most likely be attributed to the random number generator.

The reported number of interrupts is also as expected, and it is much lower than the best case scenario. If the TMU was not functioning and the random number generator returned the maximum value of 65 535 every time, the minimum number of interrupts would be approximately seven interrupts per generator per tick period. Resulting in a total of $7 * 2 * 20\,000 = 280\,000$ interrupts.

By disabling the TMU and critical section, the amount of interrupts were expected to be about 305 000 interrupts per source. Even though the number of interrupts is significantly higher in this case, Table 7.13, the system was not able to handle this rate of interrupts, resulting in interrupts being lost.

**Or1ksim** FreeRTOS and Or1ksim was not able to keep the interrupt loop alive during execution without the TMU active. This is most likely because interrupt zero is raised after *PICSR* is read during the handling of interrupt one. When interrupt one finishes execution, *PICSR* will be set to zero, in other words, deleting the interrupt request from interrupt zero. Table 7.8 shows the results of a test run where the write-back of *PICSR* is moved up in **int_main**, right after it is read from the PIC. During this execution the system is able to keep one interrupt source alive. When the minimum interrupt time-out is increased to 5 000, the system is still only able to keep one interrupt alive.

After these test results, some minor modifications was made to the interrupt generator to support the theories of why the interrupts stop during the first test runs. The solution was to move the interrupt setting loop entirely into Or1ksim. As explained in Section 5.2.5, when an interrupt is generated it will also schedule another at an arbitrary time in the future. This yielded the results in Table 7.7. This shows that the FreeRTOS running on Or1ksim can handle receiving the amount of interrupts, and it supports the theory of why it was not able to do this in the initial test. Table 7.9 is the initial test run with a short random interrupt time-out, critical section enabled around the spin lock, on Or1ksim with the interrupt generator generating the time-outs. Interestingly the amount of ignored interrupts equals the difference in handled interrupts. They are most likely from situations where the TMU tries to reassert the interrupt at the approximately same time that the interrupt generator signals a new interrupt. This table serves as a reference to show that the test results remain the same, regardless of where the interrupt time-out is generated.

**FPGA** The results of running FreeRTOS on an FPGA provides a good indication that the TMU is functioning according to the specifications. Because none of the internal

signals of the processor has been measured during the test execution, one cannot be sure that the results are not a false positive. But since the results are consistent during multiple executions, this seems unlikely. Since removing the TMU from the system yielded the expected result, the probability of a false positive decreases even more.

When the minimum time-out limit for the interrupt generators were increased to 5 000, the system was able to handle the received number of interrupts as shown in Table 7.14. This proves that there is a limit for the number of interrupts the system is able to handle without the TMU.

By comparing the values from executing the test on the instruction-set simulator and an FPGA, one can see that the values corresponds closely with each other. This serves as additional proof of the correct behaviour of the modified OR1200 core.

## 8.3 Overhead test results

From the results in Table 7.17 the overhead per tick is calculated to be 131 clock cycles, which includes both fetching data from memory and writing it to the TMU. With an average execution time of six cycles per added instruction this number is consistent with the theoretical overhead of $6 * 22 = 132$ cycles. The actual time of this overhead is dependent on system clock, and in the case of a $50\,MHz$ clock the overhead delay from the TMU is $2.62\,\mu s$. This value only shows the added overhead during a context switch and not the handling of the exception from the TMU.

All the read/write operations via the special-purpose register interface is done with a single instruction, which has a fixed execution time, and will not be subjected to delays caused by waiting on the bus. As opposed to previous implementations of a TMU connected through an external bus, the overhead of the internal implementation will remain constant. This will make the behaviour of the system more predictable, and provide easier scheduling calculations.

When caches are included in the system the number of iterations increases dramatically and the difference per tick increases. If one assumes that all instructions of the loop are present in the cache at all times meaning one cycle per instruction, then the loop will have a cycle count of 15.

The number of overhead cycles per tick is then $cycle_{overhead} = 3.11 \cdot 15 = 46.65$, which is much lower than when no caches are used.

It is unclear whether memory accesses for TMU data is included in the overhead in Stian Søvik's solution, which is 28 clock cycles. Based on the waveform diagram in [15, figure 11], the 28 clock cycles only include time spent reading and writing data to the TMU. Hence to get a comparable number, the time for all load/store operations is deducted from the overhead in Table 7.17. The number of cycles is then $6 \cdot 11 = 66$ cycles without cache and 11 cycles with cache.

## 8.4 Processor utilisation

The general behaviour is as expected. For Or1ksim the expected cycle count for the loop was correct. Below 135 the exception rate went up to 100%, as expected. The same

happened at 579 for the FPGA, this was expected at 586, the difference is negligible. Because this test pushes the compare value so close to the actual cycle count for the task there is a slight increase in exceptions as the limit approaches. The low magnitude of this implies that the system is fairly deterministic. The increase is most likely because of some rare events that cause an increase in the bus delay.

For certain values for the test run with 0.1% failure rate, the started counter is unexpectedly high. By comparing the results from task one and two it can be observed that the temporary oscillations of the two tasks started value are opposite of each other. This indicates that there is a difference in how many times each task was scheduled, although the exact reason for this is unknown. The same behaviour can be observed in the FPGA results, Figure 7.4.

As a reference point this could be compared to a monitor solution. A task which is called periodically, but in FreeRTOS such a task can only be called related to the tick period, tasks can only be unblocked during a tick timer exception. A task can not as of now have its delay extended by other tasks. This means that a monitor-task can not occur between ticks and it would have to be invoked every tick to check if a task is stuck in a loop or performing unnecessary work. The tick timer would have to be set as low as the period of a non failed task. All in all, this would lead to an extraordinary large amount of tick interrupts. Therefore this is not a comparable solution, although it is the closest one available in software.

This test shows that by using the TMU tasks can be guaranteed to finish within a certain time, either by actually finishing or by calling its exception handle. The processor is able to perform more actual work by setting an appropriate compare value.

## 8.5 Real-time effects

**Real-time improvements** The TMU contributes in four of the five characteristics of real-time operating systems explained in Section 2.1.1. The systems determinism, user control, reliability and fail-soft operation are improved. The TMU provides a way to guarantee the maximum execution time for a task. This ensures that no task will overrun its budget, even if the budget is less than the tick period. The interrupt filter part of the TMU can set a limit for incoming interrupts, ensuring that the processor will not get overwhelmed by interrupts. Compared to other solutions half of the bus delay is removed, since during a context switch the processor only has to access the memory via the bus, and not the TMU. This increases the determinism of the system because when the user assigns execution time for a set of tasks, the latest point of time when a task gets to start can be guaranteed as the sum of execution times for previous tasks added with the sum of maximum interrupt handling over the same period. Since the user has full control over task execution budgets, interrupt replenishment times and interrupt arrival limits, the system has increased user control by introducing a more hands on control of execution times. Software reliability is increased because of the possibility of TMU exceptions. When a task exceed its limit for execution time, the TMU exception will be raised. Meaning that if a task is locked in a spin- or dead-lock, the TMU exception function can be used to help that task recover, either by correcting the cause of the spin lock or, worst case, terminate tasks. Terminating individual tasks can be preferable to a

complete reboot, which would mean possible data loss. Before deleting tasks the TMU exception could store some vital data so that the task could be resumed from a recovery point. Depending on the user implementation of the TMU exception functions, their behaviour can be viewed as introducing a simple exception handling to FreeRTOS, either termination or resumption. This is one way the TMU can help to achieve decent fail-soft operation.

**Areas of use** The Fixed Priority Preemptive (FPP) scheduling policy used in FreeRTOS does not really benefit from a TMU in other aspects than the fact that the scheduler may possibly be invoked at a higher rate. Other scheduling policies can benefit greatly from TMU assistance. Out of all the scheduling policies described in [14], Rate Monotonic Scheduling(RMS) is the most obvious one to benefit from having TMU assistance. As explained in 2.1.2 RMS performs an analysis of the schedulability of a set of tasks prior to execution. Since this requires definitive numbers in regards to execution time and periodicity, the TMU can contribute in two ways. First, the TMU can guarantee an upper limit to a task's execution time. Secondly, interrupts will be easier to account for since asynchronous interrupts can be given a periodic maximum limit. This makes the analysis for RMS easier.

As seen in the TMU test performed on Or1ksim the compare value of a task can be changed during runtime. This implies that a task can dynamically change its compare value dependent on events during runtime. Meaning if a task generates unwanted TMU exceptions, it can increase its compare value, vice versa, if a task completes long before its budget it can decrease its compare value. For schedulers that perform long-term scheduling[1] a dynamic compare approach may be beneficial in the way that the scheduler will have an exact upper limit regarding required runtime for each of the already running tasks. This can not be tested on the current FreeRTOS version, because it does not perform long-term scheduling.

[1] resource analysis before admitting new tasks to the ready queue[14]

# Chapter 9

# Conclusion

This thesis has presented a working implementation of a Time Management Unit (TMU) for the OpenRISC based OR1200 processor. The TMU was designed and implemented to be an integrated part of the processor. By counting the execution time and the arrival of interrupts it can provide assistance to a operating system's scheduler. The ISA simulator Or1ksim has been modified to provide a reference for the behaviour of OR1200 with the TMU included, and the real-time operating system FreeRTOS was adapted use the functionality provided by the TMU.

Simulations and tests on an FPGA have shown that the TMU behaves according to its specifications without affecting the normal operation of the processor. Tests executed on both Or1ksim and on an FPGA show that FreeRTOS is able to use the functionality provided by the TMU. By counting task execution time, tests have proven that the TMU exceptions can help improve processor utilization and guarantee an upper limit for a tasks runtime. By counting the arrival rate of interrupts it provides a limit to how much time will be spent handling interrupts, and it protects the processor from faulty interrupt sources. Task and interrupt budgets are both set by the user, and can dynamically changed. The TMU exception routine can be used to implement fail-soft functions from software errors. With this the TMU adds increased determinism, reliability, user control and fail-soft operation.

By implementing the TMU inside a processor core, the overhead becomes fixed for all context switches, which is more stable and also provides higher predictability than previous implementation, where the overhead is dependent on the availability of the system bus.

The potential gains of having a TMU in a real-time system are considerable with regard to scheduling and security, this makes the TMU functionality described in this report a viable candidate for inclusion in other processor architectures and operating systems.

## 9.1 Further work

During the work with this project some options and ideas were discovered as possible improvements for the TMU, but they were not implemented.

- An option for connecting the replenishment budget for interrupt countering to the tick timer can be included, with an additional option for using a fixed number of tick periods as the replenishment limit.
- To remove the overhead related to starting and stopping the TMU, it could be configured to start counting automatically when the *count* and *compare* values are written. This is done by Forsman in his master thesis[4].
- A wall-clock timer could be included as described in [13]. Writing the limit for task execution, would then be calculated relative to this timer.

In addition some concepts with regard to the TMUs effect on scheduling policies remain unverified. The most promising areas are considered to be Rate Monotonic Scheduling and long-term scheduling.

# Bibliography


[1] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Pearson Education, 4 edition, 2009.

[2] FreeRTOS community. Freertos online reference. `www.freertos.org`, Feb 2014.

[3] D. L. Damjan Lampret et. al. Openrisc 1000 architecture manual, ver. 1.0. `http://opencores.org/websvn,filedetails?repname=openrisc&path=%2Fopenrisc%2Ftrunk%2Fdocs%2Fopenrisc-arch-1.0-rev0.pdf`, cited feb. 2014,, December 2012.

[4] Bjørn Forsman. En time managment unit (tmu) for sanntidssystemer. Master's thesis, NTNU, 2008.

[5] FreeRTOS. Freertos repository. `http://opencores.org/ocsvn/openrisc/openrisc/trunk/rtos/freertos-6.1.1`.

[6] Kyrre E. Gonsholt and Lars Ødegaard. Developing an openrisc system-on-chip. Technical report, NTNU, 2013.

[7] Kristoffer Gregertsen and Amund Skavhaug. Functional specification for a time management unit. *Vienna, Austria: SAFECOMP, 2010. 29th International Conference on Computer Safety, Reliability and Security*, 2010.

[8] David Patterson John Hennessy. *Computer Architecture A Quantitaive Approach*. Morgan Kaufmann, 2012.

[9] Damjan Lampret. Openrisc 1200 ip core specification, rev. 0.7. `http://opencores.org/ocsvn/openrisc/openrisc/tags/or1200/rel2/doc/`, cited feb. 2012, file must be downloaded as pdf, September 2001.

[10] OpenCores. Or1ksim repository. `https://github.com/openrisc/or1ksim`.

[11] OpenCores. Wishbone system-on-chip interconnection architecture. `http://cdn.opencores.org/downloads/wbspec_b4.pdf`, cited feb. 2014, 2010.

[12] Micheal L. Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Springer Science+Business Media, 4 edition, 2010.

[13] Håvard Skinnemoen and Amund Skavhaug. Hardware support for on-line execution time limiting of tasks in a low power environment. *Linz: Institute of System Science, Johannes Kepler University, EUROMICRO/DSD Work in Progress Session. ISBN: 3-902457-21-X*, 2003.

[14] William Stallings. *Operating Systems: Internals and Designs Principles.* Pearson Education, 7 edition, 2012.

[15] Stian Juul Søvik. Hardware implementation of a time management unit (tmu). Master's thesis, NTNU, 2010.

[16] Terasic Technologies. Cyclone v gx starter kit, user manual. `http://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=830&FID=17219f04ba333c8a2ee2066deab991e5`, cited, jun. 2014, April 2014.

Appendix A

# TMU verilog code

```
/*
* or1200_tmu.v
*
* Time management unit for OR1200
*
* Lars Ødegaard & Kyrre Gonsholt
* NTNU 2014
*
*/

`include "or1200_defines.v"

module or1200_tmu (
  //clk and rst
  clk, rst

  // SPR interface
  , spr_addr, spr_dat_i, spr_cs, spr_write, spr_dat_o

  //exception interface
  , intr, except_started

  //Interrupt interface
  , pic_ints, pic_ints_masked

  , ex_freeze

);

input         clk;
input         rst;
//
// SPR interface
//
input    [31:0]     spr_addr;
input    [31:0]     spr_dat_i;
```

```
input             spr_cs;
input             spr_write;
output   [31:0]     spr_dat_o;
//
// Exception interface
//
input             except_started;
output            intr;
//
//  Interrupt interface
//
input [`OR1200_PIC_INTS-1:0] pic_ints;
output  [`OR1200_PIC_INTS-1:0] pic_ints_masked;

input             ex_freeze;


`ifdef OR1200_TMU_IMPLEMENTED

//
//  TMU status register
//
reg    [31:0]     sr;

//
//  TMU ctrl register
//
reg    [31:0]     ctrl;

//
//  TMU count register
//
reg    [63:0]     count;

//
//  TMU compare register
//
reg    [63:0]     compare;

//
//  Interrupt replenishment counter
//
reg    [63:0]     rep_count;
reg    [63:0]     rep_compare;

//
//  Interrupt flank register
//
wire     [`OR1200_PIC_INTS-1:0]  intr_edge;
reg    [`OR1200_PIC_INTS-1:0]  intr_single;

//
//  Interrupt counting regisers
//
reg    [`OR1200_PIC_INTS-1:0]     pic_mask;
reg    [31:0]     intr_compare   [0:`OR1200_PIC_INTS-1];
reg    [31:0]     intr_count     [0:`OR1200_PIC_INTS-1];
```

```
//
//  Internal wires and regs
//
wire            sr_sel;
wire            comp_hi_sel;
wire            comp_lo_sel;
wire            count_hi_sel;
wire            count_lo_sel;
wire            repcomp_hi_sel;
wire            repcomp_lo_sel;
wire            repcnt_hi_sel;
wire            repcnt_lo_sel;

wire  [`OR1200_PIC_INTS-1:0] intr_count_sel;
wire  [`OR1200_PIC_INTS-1:0] intr_compare_sel;
reg   [31:0]        spr_dat_o;

wire            start;
wire            stop;
wire            restart;
wire            clear;
wire            ee;
wire            ce;
wire            fcd;
wire            ac;
wire            intr; /*synthesis keep*/
wire            replenish;

reg           running;
reg           count_invalid;
reg           suspended;

//
//  Decode register address
//
assign sr_sel      = (spr_cs && (spr_addr[`OR1200_TMUOFS_BITS] ==
    `OR1200_TMU_OFS_STATUS)) ? 1'b1 : 1'b0;
assign ctrl_sel    = (spr_cs && (spr_addr[`OR1200_TMUOFS_BITS] ==
    `OR1200_TMU_OFS_CTRL)) ? 1'b1 : 1'b0;
assign comp_hi_sel = (spr_cs && (spr_addr[`OR1200_TMUOFS_BITS] ==
    `OR1200_TMU_OFS_COMPARE_HI)) ? 1'b1 : 1'b0;
assign comp_lo_sel = (spr_cs && (spr_addr[`OR1200_TMUOFS_BITS] ==
    `OR1200_TMU_OFS_COMPARE_LO)) ? 1'b1 : 1'b0;
assign count_hi_sel = (spr_cs && (spr_addr[`OR1200_TMUOFS_BITS] ==
    `OR1200_TMU_OFS_COUNT_HI)) ? 1'b1 : 1'b0;
assign count_lo_sel = (spr_cs && (spr_addr[`OR1200_TMUOFS_BITS] ==
    `OR1200_TMU_OFS_COUNT_LO)) ? 1'b1 : 1'b0;
assign repcomp_hi_sel = (spr_cs && (spr_addr[`OR1200_TMUOFS_BITS] ==
    `OR1200_TMU_OFS_REPCOMP_HI)) ? 1'b1 : 1'b0;
assign repcomp_lo_sel = (spr_cs && (spr_addr[`OR1200_TMUOFS_BITS] ==
    `OR1200_TMU_OFS_REPCOMP_LO)) ? 1'b1 : 1'b0;
assign repcnt_hi_sel = (spr_cs && (spr_addr[`OR1200_TMUOFS_BITS] ==
    `OR1200_TMU_OFS_REPCNT_HI)) ? 1'b1 : 1'b0;
assign repcnt_lo_sel = (spr_cs && (spr_addr[`OR1200_TMUOFS_BITS] ==
    `OR1200_TMU_OFS_REPCNT_LO)) ? 1'b1 : 1'b0;
```

```
generate
genvar pic_nr;
  for (pic_nr = 0; pic_nr<`OR1200_PIC_INTS; pic_nr = pic_nr+1) begin :
      INTR_SEL
    assign intr_compare_sel[pic_nr] = (spr_cs && (spr_addr[
        `OR1200_TMUOFS_BITS] == `OR1200_TMU_OFS_INTRCOMP+pic_nr)) ? 1'b1 :
        1'b0;
    assign intr_count_sel[pic_nr] = (spr_cs && (spr_addr[
        `OR1200_TMUOFS_BITS] == `OR1200_TMU_OFS_INTRCNT+pic_nr)) ? 1'b1 :
        1'b0;
  end
endgenerate


//
//  Write to ctrl
//
always @(posedge clk or `OR1200_RST_EVENT rst) begin
  if (rst == `OR1200_RST_VALUE) begin
    ctrl <= 32'b0;
  end else if (ctrl_sel && spr_write) begin
    ctrl <= spr_dat_i;
  end else begin
    ctrl = 32'b0;
  end
end

//
//  Write to sr
//
always @(posedge clk or `OR1200_RST_EVENT rst) begin
  if (rst == `OR1200_RST_VALUE) begin
    //Set default operation
    sr[`OR1200_TMU_SR_R] <= 1'b0;
    sr[`OR1200_TMU_SR_EE] <= 1'b1;
    sr[`OR1200_TMU_SR_CE] <= 1'b0;
    sr[`OR1200_TMU_SR_FC] <= 1'b0;
    sr[`OR1200_TMU_SR_CI] <= 1'b1;
    sr[`OR1200_TMU_SR_CNTI] <= 1'b0;
    sr[`OR1200_TMU_SR_SUS] <= 1'b0;
    sr[`OR1200_TMU_SR_AC] <= 1'b1;

    sr[`OR1200_TMU_SR_UNUSED] <= 0;
  end else begin
    sr[`OR1200_TMU_SR_R] <= running;
    sr[`OR1200_TMU_SR_EE] <= ee;
    sr[`OR1200_TMU_SR_CE] <= ce;
    sr[`OR1200_TMU_SR_FC] <= fcd;
    sr[`OR1200_TMU_SR_CI] <= ci;
    sr[`OR1200_TMU_SR_CNTI] <= count_invalid;
    sr[`OR1200_TMU_SR_SUS] <= suspended;
    sr[`OR1200_TMU_SR_AC] <= ac;

    sr[`OR1200_TMU_SR_UNUSED] <= 0;
  end
end
```

```
//
//  Write to compare and count
//
always @(posedge clk or `OR1200_RST_EVENT rst) begin
  if (rst == `OR1200_RST_VALUE) begin
    compare <= 64'b0;
    count <= 64'b0;
    count_invalid <= 1'b0;
  //Count register
  end else if (running && !suspended) begin
    count <= count + 64'b1;
  end else if (count_hi_sel && spr_write && !running) begin
    count[`OR1200_TMU_HI_BITS] <= spr_dat_i;
    count_invalid <= 0;
  end else if (count_lo_sel && spr_write && !running) begin
    count[`OR1200_TMU_LO_BITS]  <= spr_dat_i;
    count_invalid <= 0;
  end else if (clear == 1'b1) begin
    count <= 64'b0;
    compare <= 64'b0;
    count_invalid <= 0;
  end else if (restart == 1'b1) begin
    count <= 64'b0;
    count_invalid <= 0;
  //Compare register
  end else if (comp_hi_sel && spr_write && !running) begin
    compare[`OR1200_TMU_HI_BITS] <= spr_dat_i;
    count_invalid <= 1'b1;
  end else if (comp_lo_sel && spr_write && !running) begin
    compare[`OR1200_TMU_LO_BITS] <= spr_dat_i;
    count_invalid <= 1'b1;
  end
end


//
//  Read TMU registers
//
always @(spr_addr or sr or compare or count) begin
    case (spr_addr[`OR1200_TMUOFS_BITS])
      `OR1200_TMU_OFS_STATUS: spr_dat_o = sr;
      `OR1200_TMU_OFS_CTRL: spr_dat_o = ctrl;
      `OR1200_TMU_OFS_COMPARE_HI: spr_dat_o = compare[`OR1200_TMU_HI_BITS];
      `OR1200_TMU_OFS_COMPARE_LO: spr_dat_o = compare[`OR1200_TMU_LO_BITS];
      `OR1200_TMU_OFS_COUNT_HI: spr_dat_o = count[`OR1200_TMU_HI_BITS];
      `OR1200_TMU_OFS_COUNT_LO: spr_dat_o = count[`OR1200_TMU_LO_BITS];
      `OR1200_TMU_OFS_REPCOMP_LO: spr_dat_o = rep_compare[
          `OR1200_TMU_LO_BITS];
      `OR1200_TMU_OFS_REPCOMP_HI: spr_dat_o = rep_compare[
          `OR1200_TMU_HI_BITS];
      `OR1200_TMU_OFS_REPCNT_LO: spr_dat_o = rep_count[`OR1200_TMU_HI_BITS
          ];
      `OR1200_TMU_OFS_REPCNT_HI: spr_dat_o = rep_count[`OR1200_TMU_HI_BITS
          ];
      `OR1200_TMU_OFS_PIC_MASK: begin
```

```
          spr_dat_o[31:`OR1200_PIC_INTS] = {32-`OR1200_PIC_INTS{1'b0}};
          spr_dat_o[`OR1200_PIC_INTS-1:0] = pic_mask;
        end
        `OR1200_TMU_OFS_INTRCNT: spr_dat_o = intr_count[0];
        `OR1200_TMU_OFS_INTRCNT+1: spr_dat_o = intr_count[1];
        `OR1200_TMU_OFS_INTRCNT+2: spr_dat_o = intr_count[2];
        `OR1200_TMU_OFS_INTRCNT+3: spr_dat_o = intr_count[3];
        `OR1200_TMU_OFS_INTRCNT+4: spr_dat_o = intr_count[4];
        `OR1200_TMU_OFS_INTRCNT+5: spr_dat_o = intr_count[5];
        `OR1200_TMU_OFS_INTRCNT+6: spr_dat_o = intr_count[6];
        `OR1200_TMU_OFS_INTRCNT+7: spr_dat_o = intr_count[7];
        `OR1200_TMU_OFS_INTRCNT+8: spr_dat_o = intr_count[8];
        `OR1200_TMU_OFS_INTRCNT+9: spr_dat_o = intr_count[9];
        `OR1200_TMU_OFS_INTRCNT+10: spr_dat_o = intr_count[10];
        `OR1200_TMU_OFS_INTRCNT+11: spr_dat_o = intr_count[11];
        `OR1200_TMU_OFS_INTRCNT+12: spr_dat_o = intr_count[12];
        `OR1200_TMU_OFS_INTRCNT+13: spr_dat_o = intr_count[13];
        `OR1200_TMU_OFS_INTRCNT+14: spr_dat_o = intr_count[14];
        `OR1200_TMU_OFS_INTRCNT+15: spr_dat_o = intr_count[15];
        `OR1200_TMU_OFS_INTRCNT+16: spr_dat_o = intr_count[16];
        `OR1200_TMU_OFS_INTRCNT+17: spr_dat_o = intr_count[17];
        `OR1200_TMU_OFS_INTRCNT+18: spr_dat_o = intr_count[18];
        //`OR1200_TMU_OFS_INTRCNT+19: spr_dat_o = intr_count[19];
        //`OR1200_TMU_OFS_INTRCNT+20: spr_dat_o = intr_count[20];
        //`OR1200_TMU_OFS_INTRCNT+21: spr_dat_o = intr_count[21];
        //`OR1200_TMU_OFS_INTRCNT+22: spr_dat_o = intr_count[22];
        //`OR1200_TMU_OFS_INTRCNT+23: spr_dat_o = intr_count[23];
        //`OR1200_TMU_OFS_INTRCNT+24: spr_dat_o = intr_count[24];
        //`OR1200_TMU_OFS_INTRCNT+25: spr_dat_o = intr_count[25];
        //`OR1200_TMU_OFS_INTRCNT+26: spr_dat_o = intr_count[26];
        //`OR1200_TMU_OFS_INTRCNT+27: spr_dat_o = intr_count[27];
        //`OR1200_TMU_OFS_INTRCNT+28: spr_dat_o = intr_count[28];
        //`OR1200_TMU_OFS_INTRCNT+29: spr_dat_o = intr_count[29];
        //`OR1200_TMU_OFS_INTRCNT+30: spr_dat_o = intr_count[30];
        //`OR1200_TMU_OFS_INTRCNT+31: spr_dat_o = intr_count[31];
        default: spr_dat_o = sr;
      endcase
end


//
//  Decode of running
//
always @(posedge clk or `OR1200_RST_EVENT rst) begin
  if (rst == `OR1200_RST_VALUE) begin
    running <= 1'b0;
  end else if (intr == 1'b1) begin
    running <= 1'b0;
  end else if (running == 1'b0 && start == 1'b1 && compare > 0) begin
    running <= 1'b1;
  end else if (running == 1'b1 && stop == 1'b1) begin
    running <= 1'b0;
  end else if (running == 1'b1 && ce == 1'b0 && except_started == 1'b1)
      begin
    running <= 1'b0;
  end else if (running == 1'b1) begin
    running <= 1'b1;
```

```
  end else begin
    running <= 1'b0;
  end
end

//
//  Decode of suspended
//
always @(posedge clk or `OR1200_RST_EVENT rst) begin
  if (rst == `OR1200_RST_VALUE) begin
    suspended <= 1'b0;
  end else if (fcd == 1'b1 && ex_freeze == 1'b1) begin
    suspended <= 1'b1;
  end else if (ci == 1'b0 && 1'b1) begin
    suspended <= 1'b1;
  end else begin
    suspended <= 1'b0;
  end
end

//
//  Write to replenishment compare
//
always @(posedge clk or `OR1200_RST_EVENT rst) begin
  if (rst == `OR1200_RST_VALUE) begin
    rep_compare <= 64'b0;
  end else if (repcomp_hi_sel && spr_write) begin
    rep_compare[`OR1200_TMU_HI_BITS] <= spr_dat_i;
  end else if (repcomp_lo_sel && spr_write) begin
    rep_compare[`OR1200_TMU_LO_BITS] <= spr_dat_i;
  end
end


//
//  Write to interrupt compare registers
//
integer piccm;
always @(posedge clk or `OR1200_RST_EVENT rst) begin
  if (rst == `OR1200_RST_VALUE) begin
    for (piccm = 0; piccm < `OR1200_PIC_INTS; piccm = piccm + 1) begin
      intr_compare[piccm] <= 32'b0;
    end
  end else begin
    for (piccm = 0; piccm < `OR1200_PIC_INTS; piccm = piccm + 1) begin
      if (intr_compare_sel[piccm] && spr_write) begin
        intr_compare[piccm] <= spr_dat_i;
      end
    end
  end
end


//
//  Increment interrupt counters
//
integer pics;
```

```
always @(posedge clk or `OR1200_RST_EVENT rst) begin
  if (rst == `OR1200_RST_VALUE) begin
    for (pics = 0; pics < `OR1200_PIC_INTS; pics = pics + 1) begin
      intr_count[pics] <= 32'b0;
    end
  end else if (replenish == 1'b1) begin
    for (pics = 0; pics < `OR1200_PIC_INTS; pics = pics + 1) begin
      intr_count[pics] <= 32'b0;
    end
  end else begin
    for (pics = 0; pics < `OR1200_PIC_INTS; pics = pics + 1) begin
      if (intr_edge[pics] == 1'b1 && sr[`OR1200_TMU_SR_CI]) begin
        intr_count[pics] <= intr_count[pics] + 1;
      end else begin
        intr_count[pics] <= intr_count[pics];
      end
    end
    pics = 0;
  end
end

//
//  Edge detector for interrupts
//
always @(posedge clk or `OR1200_RST_EVENT rst) begin
  if (rst == `OR1200_RST_VALUE) begin
    intr_single <= `OR1200_PIC_INTS'b0;
  end else begin
    intr_single <= pic_ints;
  end
end
assign intr_edge = pic_ints & (~intr_single);

//
//  Decode interrupt mask
//
integer mask;
always @(posedge clk or `OR1200_RST_EVENT rst) begin
  if (rst == `OR1200_RST_VALUE) begin
    pic_mask <= `OR1200_PIC_INTS'b0;
    mask <= 0;
  end else begin
    for (mask = 0; mask < `OR1200_PIC_INTS; mask = mask + 1) begin
      if ((intr_count[mask] >= intr_compare[mask]) && (intr_compare[mask]
          != 0) && sr[`OR1200_TMU_SR_CI]) begin
        pic_mask[mask] <= 1'b1;
      end else begin
        pic_mask[mask] <= 1'b0;
      end
    end
  end
end

//
//  Increment replenishment counter
//
always @(posedge clk or `OR1200_RST_EVENT rst) begin
```

```
  if (rst == `OR1200_RST_VALUE) begin
    rep_count <= 64'b0;
  end else if (replenish) begin
    rep_count <= 64'b0;
  end else if (rep_compare != 0) begin
    rep_count <= rep_count + 1;
  end else begin
    rep_count <= 64'b0;
  end
end


assign ee = ctrl[`OR1200_TMU_CTRL_EE] & ~ctrl[`OR1200_TMU_CTRL_ED] |
            sr[`OR1200_TMU_SR_EE] & ~ctrl[`OR1200_TMU_CTRL_ED];

assign ce = ctrl[`OR1200_TMU_CTRL_CEE] & ~ctrl[`OR1200_TMU_CTRL_CED] |
            sr[`OR1200_TMU_SR_CE] & ~ctrl[`OR1200_TMU_CTRL_CED];

assign fcd = ctrl[`OR1200_TMU_CTRL_FCE] & ~ctrl[`OR1200_TMU_CTRL_FCD] |
            sr[`OR1200_TMU_SR_FC] & ~ctrl[`OR1200_TMU_CTRL_FCD];

assign ci = ctrl[`OR1200_TMU_CTRL_CIE] & ~ctrl[`OR1200_TMU_CTRL_CID] |
            sr[`OR1200_TMU_SR_CI] & ~ctrl[`OR1200_TMU_CTRL_CID];

assign ac = ctrl[`OR1200_TMU_CTRL_ACE] & ~ctrl[`OR1200_TMU_CTRL_ACD] |
            sr[`OR1200_TMU_SR_AC] & ~ctrl[`OR1200_TMU_CTRL_ACD];

assign start = ctrl[`OR1200_TMU_CTRL_START];
assign stop = ctrl[`OR1200_TMU_CTRL_STOP];
assign restart = ctrl[`OR1200_TMU_CTRL_RESTART];
assign clear = ctrl[`OR1200_TMU_CTRL_CLEAR];

assign replenish = (rep_count == rep_compare) ? 1'b1 : 1'b0;
assign intr = ((count >= compare) && (sr[`OR1200_TMU_SR_EE] == 1'b1) &&
    compare != 0 ) ? 1'b1 : 1'b0;
assign pic_ints_masked = pic_ints & ~pic_mask;


`else

//When TMU is not implemented drive outputs as if it was inactive

assign intr = 1'b0;
assign spr_dat_o = 32'b0;
assign pic_ints_masked = pic_ints;



`endif //OR1200_TMU_IMPLEMENTED

endmodule
```

Appendix B

# Folder layout

The attached files contains the following directories:

**results:** Results from test executions

**hardware:** Hardware source and support files

**simulator:** Simulator source and support files

**software:** FreeRTOS source and support files

## B.1 Test results

The test results file is located in the folder *results*, and has the following subdirectories:

**or1200-tests:** Results of the tests from OrpSoC with the TMU included in the system

**or1200-tmu-full:** Results of the TMU-full test

**tmu-bench:** Results from the TMU testbench

**dualtask:** Results from the processor utilization test

For each of the tests described in tables 4.2 and 4.3 five different log-files are created:

**vvp.log:** Output from the **vvp** simulator

**sprs.log:** Access logs for the special-putpose registers

**lookup.log:** Instruction number and simulation time

**general.log:** Status reports from the simulator

**executed.log:** Executed instructions and register file contents after each instruction

**.fst:** Waveform file

```
results
├── or1200-tests
├── or1200-tmu-tests
├── or1200-tmu-full
├── tmu-bench
└── dualtask
```

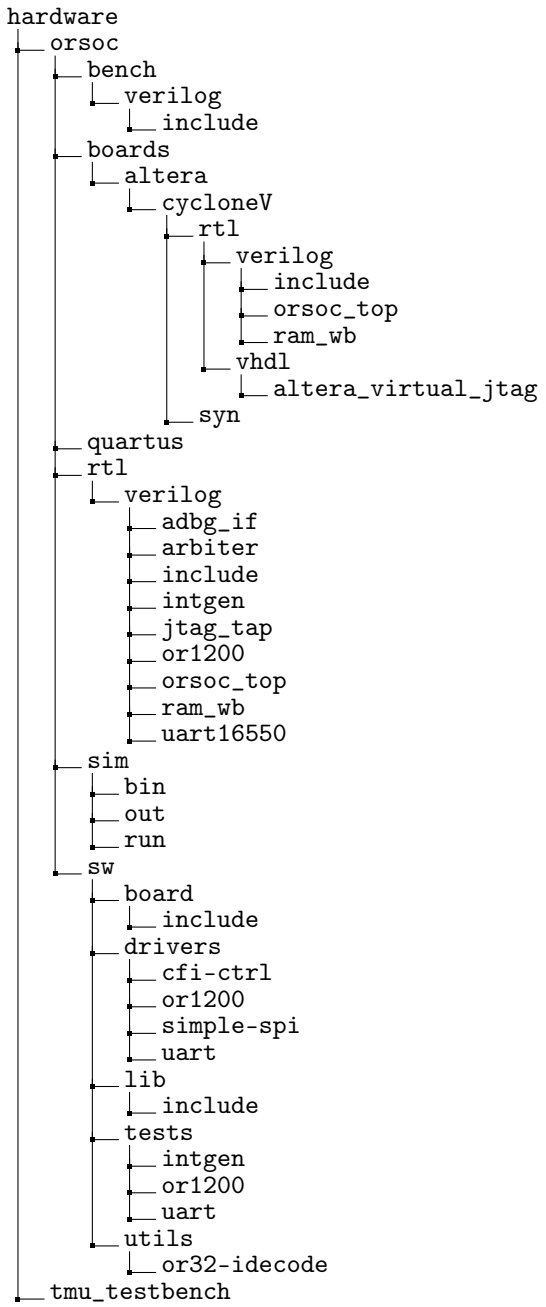
## B.2 Hardware files

The files for the System-on-chip and all components necessary to compile the design for simulation is found in the *hardware* folder. This folder has several subdirectories, each with a different purpose.

**orsoc:** Contains all files for the System-on-Chip

- **bench:** Contains the testbench top-level modules and files
- **board:** Contains board-specific modules and files
- **rtl:** Contains the different modules used in the system, including OR1200
- **sim:** Contains files for simulating the system
- **sw:** Contains test software and drivers
- **utils:** Contains some software utilities

**tmu_testbench:** Contains the TMU testbench


hardware
orsoc
bench
verilog
include
boards
altera
cycloneV
rtl
verilog
include
orsoc_top
ram_wb
vhdl
altera_virtual_jtag
syn
quartus
rtl
verilog
adbg_if
arbiter
include
intgen
jtag_tap
or1200
orsoc_top
ram_wb
uart16550
sim
bin
out
run
sw
board
include
drivers
cfi-ctrl
or1200
simple-spi
uart
lib
include
tests
intgen
or1200
uart
utils
or32-idecode
tmu_testbench

## B.3 Or1ksim files

```
simulator
├── argtable2
├── autom4te.cache
├── bpb
├── cache
├── cpu
│   ├── common
│   ├── or1k
│   └── or32
├── cuc
├── debug
├── doc
├── m4
├── mmu
├── pcu
├── peripheral
│   └── channels
├── pic
├── pm
├── port
├── softfloat
├── support
├── tick
├── tmu
└── vapi
```

## B.4 Software files

The files for FreeRTOS can be found in the *software* folder. The main application files is found in *Application/OpenRISC_SIM_GCC*, which has the main FreeRTOS file, the configuration file and the makefile. In addition this folder contains three subdirectories.

**arch:** Architecture specific files and drivers

**drivers:** Drivers for

**test:** Source files for tests

The back-end source files for FreeRTOS is located in *Source*, and contains all files for the functionality provided by the operating system.

**include** include files

**portable** files for architecture dependent support functionality

```
software
└── FreeRTOSV6.6.6
    ├── Application
    │   └── OpenRISC_SIM_GCC
    │       ├── arch
    │       ├── drivers
    │       └── test
    ├── License
    └── Source
        ├── include
        └── portable
            ├── GCC
            │   └── OpenRISC
            └── MemMang
```

Appendix C

# Full system setup

## C.1 Simulation

### C.1.1 Tools

To run the system simulation the following tools are required:

**make** GNU make utility
**iverilog** Icarus Verilog compiler
**vvp** Open-Source hardwre simulator
**or32-elf-gcc** Compiler tool-chain for OpenRISC
**Quartus** Alteras development tool
**adv_jtag_bridge** Debug communication tool for OR1200

### C.1.2 TMU testbench

To run the TMU testbench, type `make` in the testbench folder. This will compile and run the tests defined in *bench_defines.v*. If some tests are to be excluded from simulation, comment them out in *bench_defines.v*. When the tests complete, the waveform-viewer GTKWave will display the waveform from the tests.

### C.1.3 Full system tests

The *makefile* for the full system tests is located in *hardware/orsoc/sim/run*. The two main targets are `rtl-tests` and `rtl-test`. Where `rtl-tests` will run through all the tests specified in the **TESTS variable**. To execute a single test, build the target `rtl-test` and define the desired test by setting the **TEST** variable. If a waveform file should be created specify `VCD=1`.

#### Examples

Run all the tests:

`make rtl-tests`

Run all the tests and generate waveform:

`make rtl-tests VCD=1`

Run a single test and generate waveform:

`make rtl-test TEST=<name> VCD=1`

Simulate using a specified *elf*-file, and generate waveform:

`make rtl-test USER_ELF=<elf-file> VCD=1`

#### Additional targets

**clean** cleans all files generated during compilation and simulation
**sw-elf** build an *elf*-file from the specified test
**sw-dis** disassemble the *elf*-file from the specified test
**sram.vmem** build the memory input file for the simulator

#### Simulation FreeRTOS

To simulate the execution of FreeRTOS the **FREERTOS_DIR** has to be set, the `freertos-sim` can then be executed.

`make freertos-sim FREERTOS_DIR=</path/to/freertos/makefile>`

## C.2 Compiling for FPGA

The compilation and synthesizing for FPGA was done useing Alteras Quartus. All necessary set-up information for quartus is found in the *.tcl*-files in *hardware/orsoc/boards/altera/cycloneV/syn/quartus/tcl*. The board specific design files are located in *hardware/orsoc/boards/altera/cycloneV/rtl*.

## C.3 Uploading and running programs on the system

Once the system is compiled and loaded to the FPGA, communication with the system is done through *adv_jtag_bridge*. To start this program, navigate to *hardware/orsoc/boards/altera/cycloneV/syn/quartus/bsdl*, where the boundary scan files are located, and run:

`adv_jtag_bridge -a 1 -b ./ -g 9999 ft245`

This will initiate communication with the debugger in the system and set up a communication port for GDB at port `9999`. The debugger uses the USBBlaster on the FPGA board, and this USB-cable has to be connected.

To upload an executable to the system compile it with *or32-elf-gcc*, and execute the following command to start GDB:

```
or32-elf-gdb <target-file>
```

When GDB is started, the following commands will upload and start the program:

```
target remote :9999
load
set $pc=0x100
continue
```

## C.4 Building and running Or1ksim

To build the Or1ksim included in appendix, enter the parent folder of the Or1ksim folder and run the following commands:

```
mkdir bld-or1ksim
cd bld-or1ksim
../or1ksim/configure
make
```

This will configure the Or1ksim and build a executable for the simulator. Programs can be run on this by:

```
./sim --nosrv -f sim.cfg <PROG>.or32
```

The '–nosrv' option indicates that the debug server should not be launched. The '-f' option includes the simulation configuration file.

## C.5 Test tutorial FreeRTOS and Or1ksim

From the **FreeRTOSV6.1.1/Application/OpenRISC_SIM_GCC**-folder all tests are compiled and via makefile commands. **Makefile**, **FreeRTOSConfig.h** and **sim.cfg** are located in this folder, **interrupts.c** is located in the **arch** sub folder. All test source files are located in the **test** sub folder. Other paths are specified.

In **Makefile** the directory where Or1ksim is build must be specified to the *OR1KSIM_DIR*-variable. Depending on the version of *or32/or1k-elf-gcc* this must be specified in **Makefile.inc**, the variables are *TARGET* and *GCCVER*. The variable *CCPATH* must also be specified to the folder where the compiler toolchain was installed. If the compiler still complains about problems regarding the linker, make sure that the *LIBS* variable in the **Makefile** is correct.

All FreeRTOS applications are known to work for version 4.9.0 20140308 (experimental) *or1k-elf-gcc*-compiler.

After setting up a test run the following command in this folder.

```
make sim
```

**tmutest.c**:

- In **FreeRTOSConfig.h**:

  - *configUSE_TMU* 0
  - *configUSE_TICK_HOOK* 1
- Define *tmutest* in:
  - **interrupts.c**
  - **main.c**
  - **<or1ksim-path>/tmu/tmu.c**
- Add **tmutest.c** to *TEST_SRC* in the makefile, exclude all other tests.
- In **sim.cfg**:
  - section tmu:
    - enabled = 1
    - task_timer = 1
    - int_filter = 1
    - status = 0x12
  - section intgen
    - enabled = 1
    - baseaddr = 0xa0000000
    - size = 0x01000008
    - version = 1

Recompile Or1ksim to include the printout for this test. Remember to remove the *tmutest* definition in **<or1ksim-path>/tmu/tmu.c** and **interrupts.c** and recompile after this test is run. Otherwise Or1ksim will spam the simulator output and the other tests will take significantly more time.

**freertostest.c**:

- In **FreeRTOSConfig.h**:
  - *configUSE_TMU* 1
  - *configUSE_PREEMPTION* 0
  - *conigUSE_IDLE_HOOK* 1
  - *configUSE_TICK_HOOK* 1
- Define *freertostest* in:
  - **main.c**
- Add **freertostest.c** to *TEST_SRC* in the makefile, exclude all other tests.
- In **sim.cfg**:
  - section tmu:
    - enabled = 1
    - task_timer = 1
    - int_filter = N/A
    - status = 0x12

**maintest.c**:

- In **FreeRTOSConfig.h**:
  - *configUSE_TMU* 1
  - *configUSE_PREEMPTION* 1
  - *configUSE_TICK_HOOK* 1
- In **main.c** in:
  - Define *maintest*
  - Define *ENDTIME* 20 000
- Add **maintest.c** to *TEST_SRC* in the makefile, exclude all other tests.
- In **sim.cfg**:
  - section tmu:
    - enabled = 1|0
    - task_timer = 1
    - int_filter = 1
    - status = 0x12
  - section intgen
    - enabled = 1
    - baseaddr = 0xa0000000
    - size = 0x01000008
    - version = 1|2

Running maintest with version two of the interrupt generator, line 88 in **maintest.c** must be removed, *intgen_set_timeout (intgen, rand()%RAND_MAX);*. Changing the maximum time-out for version two can be done by changing the *RAND_MAX* in <**or1ksim-path**>/**peripheral**/**intgen.c** and the recompile Or1ksim. To run this test on the FPGA, the part about setting up **sim.cfg** can be discarded.

**dualtask.c**:

- In **FreeRTOSConfig.h**:
  - *configUSE_TMU* 1
  - *configUSE_PREEMPTION* 1
  - *configUSE_TICK_HOOK* 1
- In **main.c** in:
  - Define*maintest*
  - Define *ENDTIME* 2000
- Add **dualtask.c** to *TEST_SRC* in the makefile, exclude all other tests.
- In **sim.cfg**:
  - section tmu:
    - enabled = 1
    - task_timer = 1
    - int_filter = N/A
    - status = 0x12

This test can receive arguments from the make command, specificaly: compare and failure rate(1 failure every *rate*). Without cleaning before recompiling, the variable change will not have any effect.

```
make clean; make sim VARS='-Dcompare=135 -Drate=100'
```

To run this test on the FPGA, the part about setting up **sim.cfg** can be discarded.

Appendix D

# Or1ksim testing

## D.1 TMU test

Listing D.1: Results from the TMU test on Or1ksim

```
~/Master/bld-or1ksim/sim --nosrv -f ~/Master/or1ksim/sim.cfg rtosdemo.or32
Seeding random generator with value 0x355308e0
Warning: Unknown Ethernet type: file assumed.
Or1ksim 2012-04-27
Building automata... done, num uncovered: 0/215.
Parsing operands data... done.
Warning: PS2 keyboard unable to open RX file stream.
Resetting PIC.
loadcode: filename rtosdemo.or32  startaddr=00000000  virtphy_transl
    =00000000
Not COFF file format
ELF type: 0x0002
ELF machine: 0x005c
ELF version: 0x00000001
ELF sec = 18
Section: .vectors, vaddr: 0x00000000, paddr: 0x0 offset: 0x00002000, size:
    0x00000f1c
Section: .text, vaddr: 0x00001000, paddr: 0x1000 offset: 0x00003000, size:
    0x00009df4
Section: .rodata, vaddr: 0x0000adf4, paddr: 0xadf4 offset: 0x0000cdf4, size
    : 0x00000378
Section: .data, vaddr: 0x0000b16c, paddr: 0xb16c offset: 0x0000d16c, size:
    0x0000000c

IntGen device "Interrupt Generator" at 0xa0000000:
  Size 0x1000008
  Full word R/W enabled

Hardware setup successfull
Starting FreeRTOS
Starting scheduler
tmu_driver_test returns with 0 errors
```

```
TMU:: Scheduling exception, Compare = 400000
res2_except
TMU:: Scheduling exception, Compare = 4294967297
res2_except
TMU:: Scheduling exception, Compare = 1200000
res2_except
tt_behavior_test returns with 0 errors
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 0 Pass:: 3 | # 1
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 1 Pass:: 3 | # 2
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 2 Ignored 3 | # 1

TMU:: Replenish ---------------
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 0 Pass:: 3 | # 3
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 1 Pass:: 3 | # 4
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 2 Ignored 3 | # 2

TMU:: Replenish ---------------
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 0 Pass:: 3 | # 5
TMU:: Filter:: Count interrupts are disabled :: Int 3
TMU:: Filter:: Count interrupts are disabled :: Int 3
TMU:: Filter:: Count interrupts are disabled :: Int 3
TMU:: Filter:: Count interrupts are disabled :: Int 3
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 1 Pass:: 3 | # 6
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 2 Ignored 3 | # 3

TMU:: Replenish ---------------
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 0 Pass:: 3 | # 7
TMU:: Filter:: Checking line 4
   Budget = 3    Count = 0 Pass:: 4 | # 1
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 1 Pass:: 3 | # 8
TMU:: Filter:: Checking line 4
   Budget = 3    Count = 1 Pass:: 4 | # 2
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 2 Ignored 3 | # 4
TMU:: Filter:: Checking line 4
   Budget = 3    Count = 2 Pass:: 4 | # 3
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 2 Ignored 3 | # 5
TMU:: Filter:: Checking line 4
   Budget = 3    Count = 3 Ignored 4 | # 1
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 2 Ignored 3 | # 6
TMU:: Filter:: Checking line 4
   Budget = 3    Count = 3 Ignored 4 | # 2
TMU:: Filter:: Checking line 3
   Budget = 2    Count = 2 Ignored 3 | # 7
```

```
TMU:: Replenish ----------
TMU:: Filter:: Checking line 4
   Budget = 3     Count = 0 Pass:: 4 | # 4
TMU:: Filter:: Checking line 3
   Budget = 2     Count = 0 Pass:: 3 | # 9
if_behavior_test returns with 0 errors
TMU test finished with 0 errors
exit(0)
@reset : cycles 0, insn #0
@exit  : cycles 4318617772, insn #3580670882
 diff  : cycles 4318617772, insn #3580670882
```

## D.2 FreeRTOS test

Listing D.2: Results from the FreeRTOS test on Or1ksim

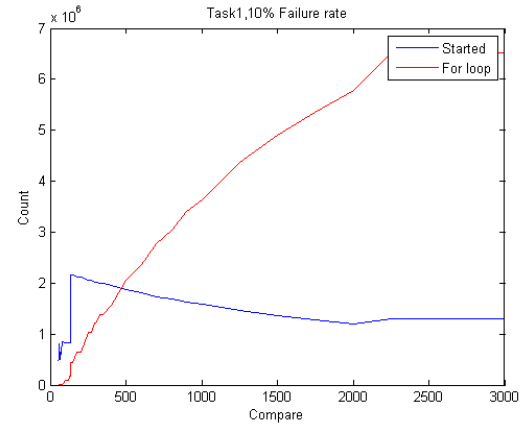
```
~/Master/bld-or1ksim/sim --nosrv -f ~/Master/or1ksim/sim.cfg rtosdemo.or32
Seeding random generator with value 0x8eca1501
Warning: Unknown Ethernet type: file assumed.
Or1ksim 2012-04-27
Building automata... done, num uncovered: 0/215.
Parsing operands data... done.
Warning: PS2 keyboard unable to open RX file stream.
Resetting PIC.
loadcode: filename rtosdemo.or32  startaddr=00000000  virtphy_transl
    =00000000
Not COFF file format
ELF type: 0x0002
ELF machine: 0x005c
ELF version: 0x00000001
ELF sec = 18
Section: .vectors, vaddr: 0x00000000, paddr: 0x0 offset: 0x00002000, size:
    0x00000f0c
Section: .text, vaddr: 0x00001000, paddr: 0x1000 offset: 0x00003000, size:
    0x00008c18
Section: .rodata, vaddr: 0x00009c18, paddr: 0x9c18 offset: 0x0000bc18, size
    : 0x00000338
Section: .data, vaddr: 0x00009f50, paddr: 0x9f50 offset: 0x0000bf50, size:
    0x0000002c

IntGen device "Interrupt Generator" at 0xa0000000:
  Size 0x1000008
  Full word R/W enabled

Hardware setup successfull
Starting FreeRTOS
Starting scheduler
Start   Task 2, Compare = 2000000, Count =  58
End     Task 2, Compare = 2000000, Count =  510841
Start   Task 1, Compare = 1000000, Count =  58
End     Task 1, Compare = 1000000, Count =  510301
Start   Task 2, Compare = 2000000, Count =  549322
End     Task 2, Compare = 2000000, Count =  1080397
Start   Task 1, Compare = 1000000, Count =  548794
End     Task 1, Compare = 1000000, Count =  1079346
```

```
1-exception
Start    Task 2, Compare = 2000000, Count =  1123674
End      Task 2, Compare = 2000000, Count =  1659600
Start    Task 1, Compare = 1000000, Count =  53
End      Task 1, Compare = 1000000, Count =  510301
Start    Task 2, Compare = 2000000, Count =  1703090
2-exception
Start    Task 1, Compare = 1000000, Count =  548789
End      Task 1, Compare = 1000000, Count =  1079337
1-exception
Start    Task 2, Compare = 2000000, Count =  50279
End      Task 2, Compare = 2000000, Count =  576029
Start    Task 2, Compare = 2000000, Count =  614502
End      Task 2, Compare = 2000000, Count =  1145557
Start    Task 2, Compare = 2000000, Count =  1188932
End      Task 2, Compare = 2000000, Count =  1724841
Start    Task 2, Compare = 2000000, Count =  1768344
2-exception
Start    Task 2, Compare = 2000000, Count =  110247
End      Task 2, Compare = 2000000, Count =  640810
Start    Task 2, Compare = 2000000, Count =  679287
End      Task 2, Compare = 2000000, Count =  1210440
Start    Task 2, Compare = 2000000, Count =  1253778
End      Task 2, Compare = 2000000, Count =  1789700
Start    Task 2, Compare = 2000000, Count =  1833230
2-exception
Start    Task 2, Compare = 2000000, Count =  110247
End      Task 2, Compare = 2000000, Count =  640815
Start    Task 2, Compare = 2000000, Count =  679287
End      Task 2, Compare = 2000000, Count =  1210439
Start    Task 2, Compare = 2000000, Count =  1253781
End      Task 2, Compare = 2000000, Count =  1789703
Start    Task 2, Compare = 2000000, Count =  1833233
2-exception
exit(0)
@reset : cycles 0, insn #0
@exit  : cycles 11126601, insn #7209972
 diff  : cycles 11126601, insn #7209972
```

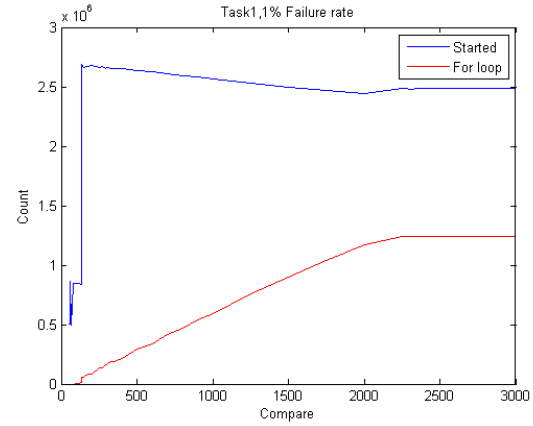
## D.3 Processor utilisation Figures

## D.4 Processor utilisation results

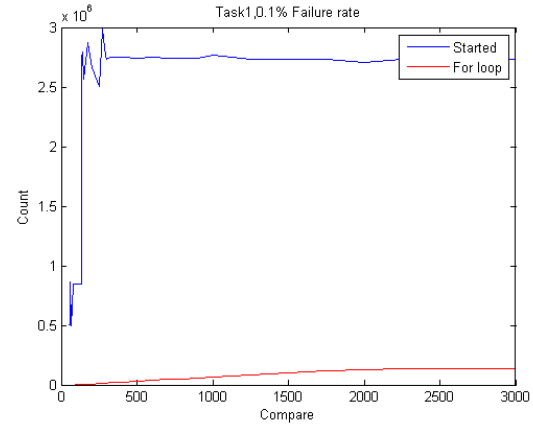
Percentages is calculated in **dualtask.xls** and not included here because of limited space.




(a) 10% failure rate

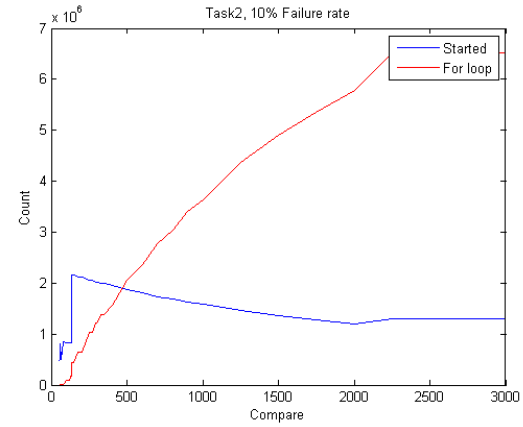



(b) 1% failure rate

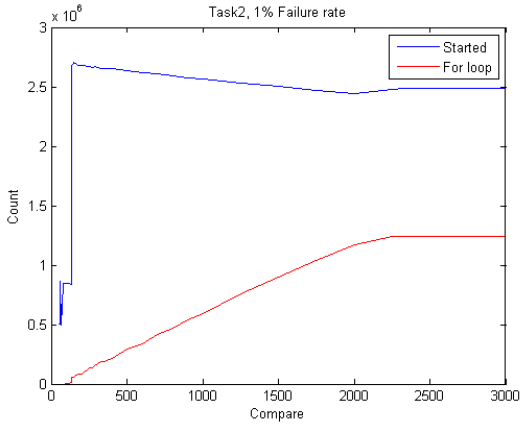



(c) 0.1% failure rate

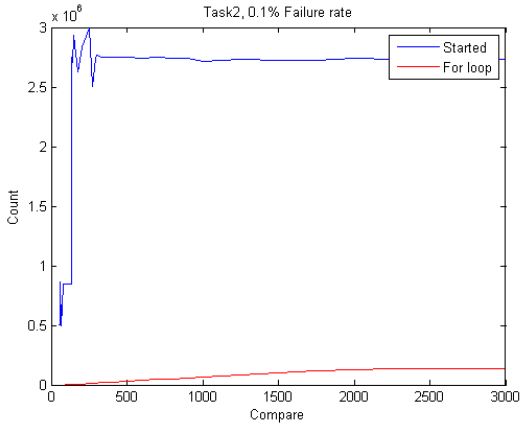
Figure D.1: Task 1 started versus time spent in the for loop, all failure rates, Or1ksim




(a) 10% failure rate




(b) 1% failure rate




(c) 0.1% failure rate

Figure D.2: Task 2 started versus time spent in the for loop, all failure rates, Or1ksim

Table D.1: Processor utilisation results, Task 1, 10% failure rate, Or1ksim

| | | Task 1 | | |
|---|---|---|---|---|
| Compare | Started | Exception | Completed | For-loop |
| 50 | 484 411 | 1 017 312 | 10 | 0 |
| 61 | 504 548 | 1 009 134 | 27 | 1 |
| 62 | 805 043 | 886 674 | 6 | 0 |
| 67 | 496 917 | 993 764 | 447 131 | 2 |
| 80 | 841 122 | 841 314 | 756 986 | 2 |
| 100 | 835 610 | 835 654 | 752 022 | 83 558 |
| 120 | 835 668 | 835 669 | 752 087 | 83 561 |
| 130 | 830 087 | 830 094 | 747 078 | 166 011 |
| 134 | 830 082 | 830 090 | 747 073 | 166 011 |
| 135 | 2 161 517 | 216 892 | 1 945 365 | 430 542 |
| 140 | 2 156 777 | 216 236 | 1 941 093 | 431 869 |
| 150 | 2 157 550 | 216 221 | 1 941 762 | 431 774 |
| 175 | 2 121 657 | 212 409 | 1 909 426 | 636 618 |
| 200 | 2 123 747 | 212 493 | 1 911 286 | 635 985 |
| 250 | 2 051 147 | 205 574 | 1 846 023 | 1 025 827 |
| 270 | 2 053 066 | 205 671 | 1 847 760 | 1 025 152 |
| 290 | 2 019 124 | 201 962 | 1 817 212 | 1 211 502 |
| 300 | 2 019 667 | 202 011 | 1 817 700 | 1 211 115 |
| 325 | 1 986 117 | 198 612 | 1 787 505 | 1 390 327 |
| 350 | 1 986 702 | 198 670 | 1 788 032 | 1 391 191 |
| 400 | 1 955 977 | 195 598 | 1 760 379 | 1 564 796 |
| 500 | 1 867 967 | 186 797 | 1 681 170 | 2 055 278 |
| 600 | 1 813 537 | 181 354 | 1 632 183 | 2 358 325 |
| 700 | 1 738 198 | 173 820 | 1 564 378 | 2 781 122 |
| 800 | 1 691 227 | 169 122 | 1 522 104 | 3 043 869 |
| 900 | 1 625 177 | 162 518 | 1 462 659 | 3 412 540 |
| 1 000 | 1 584 247 | 158 425 | 1 425 822 | 3 642 118 |
| 1 250 | 1 454 867 | 145 487 | 1 309 380 | 4 363 643 |
| 1 500 | 1 359 647 | 135 965 | 1 223 682 | 4 894 620 |
| 1 750 | 1 276 287 | 127 628 | 1 148 658 | 5 360 403 |
| 2 000 | 1 202 477 | 120 248 | 1 082 229 | 5 772 422 |
| 2 250 | 1 309 667 | 16 | 1 309 657 | 6 509 614 |
| 2 331 | 1 304 507 | 3 | 1 304 506 | 6 535 600 |
| 2 332 | 1 305 799 | 0 | 1 305 799 | 6 529 150 |
| 2 500 | 1 305 799 | 0 | 1 305 799 | 6 529 150 |
| 3 000 | 1 305 799 | 0 | 1 305 799 | 6 529 150 |
| No TMU | 1 305 799 | 0 | 1 305 799 | 6 529 150 |

Table D.2: Processor utilisation results, Task 2, 10% failure rate, Or1ksim

| | | Task 2 | | |
|---|---|---|---|---|
| Compare | Started | Exception | Completed | For-loop |
| 50 | 484 412 | 1 017 323 | 6 | 0 |
| 61 | 504 563 | 1 009 158 | 22 | 1 |
| 62 | 806 195 | 887 404 | 22 | 0 |
| 67 | 496 919 | 993 782 | 447 137 | 2 |
| 80 | 841 231 | 841 372 | 757 057 | 2 |
| 100 | 835 659 | 835 695 | 752 042 | 83 558 |
| 120 | 835 653 | 835 655 | 752 055 | 83 561 |
| 130 | 830 116 | 830 123 | 747 103 | 166 011 |
| 134 | 830 122 | 830 126 | 747 110 | 166 011 |
| 135 | 2 153 254 | 216 365 | 1 937 928 | 430 542 |
| 140 | 2 159 478 | 216 331 | 1 943 526 | 431 869 |
| 150 | 2 158 894 | 216 251 | 1 942 980 | 431 774 |
| 175 | 2 122 062 | 212 428 | 1 909 789 | 636 618 |
| 200 | 2 119 984 | 212 361 | 1 907 709 | 635 985 |
| 250 | 2 052 153 | 205 791 | 1 846 938 | 1 025 827 |
| 270 | 2 050 554 | 205 529 | 1 845 498 | 1 025 152 |
| 290 | 2 019 414 | 201 978 | 1 817 472 | 1 211 502 |
| 300 | 2 018 764 | 201 954 | 1 816 887 | 1 211 115 |
| 325 | 1 986 733 | 198 673 | 1 788 060 | 1 390 327 |
| 350 | 1 987 694 | 198 770 | 1 788 924 | 1 391 191 |
| 400 | 1 956 148 | 195 615 | 1 760 533 | 1 564 796 |
| 500 | 1 868 534 | 186 854 | 1 681 680 | 2 055 278 |
| 600 | 1 814 470 | 181 447 | 1 633 022 | 2 358 325 |
| 700 | 1 738 234 | 173 824 | 1 564 410 | 2 781 122 |
| 800 | 1 691 054 | 169 106 | 1 521 948 | 3 043 869 |
| 900 | 1 625 034 | 162 504 | 1 462 530 | 3 412 540 |
| 1 000 | 1 583 542 | 158 354 | 1 425 188 | 3 642 118 |
| 1 250 | 1 454 554 | 145 456 | 1 309 098 | 4 363 643 |
| 1 500 | 1 359 740 | 135 974 | 1 223 766 | 4 894 620 |
| 1 750 | 1 276 294 | 127 630 | 1 148 664 | 5 360 403 |
| 2 000 | 1 202 614 | 120 261 | 1 082 352 | 5 772 422 |
| 2 250 | 1 301 924 | 23 | 1 301 917 | 6 509 614 |
| 2 331 | 1 307 121 | 1 | 1 307 121 | 6 535 600 |
| 2 332 | 1 305 829 | 0 | 1 305 829 | 6 529 150 |
| 2 500 | 1 305 829 | 0 | 1 305 829 | 6 529 150 |
| 3 000 | 1 305 829 | 0 | 1 305 829 | 6 529 150 |
| No TMU | 1 305 829 | 0 | 1 305 829 | 6 529 150 |

Table D.3: Processor utilisation results, Task 1, 1% failure rate, Or1ksim

| | | Task 1 | | |
|---|---|---|---|---|
| Compare | Started | Exception | Completed | For-loop |
| 50 | 503 783 | 1 012 658 | 7 | 0 |
| 61 | 505 889 | 1 011 791 | 17 | 0 |
| 62 | 860 480 | 870 200 | 33 | 0 |
| 67 | 497 446 | 994 821 | 492 383 | 0 |
| 80 | 842 764 | 842 834 | 834 316 | 0 |
| 100 | 842 244 | 842 265 | 833 774 | 8 422 |
| 120 | 842 243 | 842 245 | 833 793 | 8 422 |
| 130 | 841 676 | 841 677 | 833 260 | 16 834 |
| 134 | 841 672 | 841 672 | 833 255 | 16 832 |
| 135 | 2 688 097 | 28 065 | 2 661 216 | 53 519 |
| 140 | 2 675 497 | 27 724 | 2 648 739 | 53 787 |
| 150 | 2 663 297 | 27 209 | 2 636 654 | 54 077 |
| 175 | 2 674 297 | 27 079 | 2 647 529 | 80 460 |
| 200 | 2 677 883 | 27 077 | 2 650 917 | 80 360 |
| 250 | 2 662 230 | 26 975 | 2 635 571 | 133 570 |
| 270 | 2 670 497 | 26 973 | 2 643 792 | 133 171 |
| 290 | 2 656 597 | 26 699 | 2 630 031 | 159 992 |
| 300 | 2 663 268 | 26 666 | 2 636 636 | 159 599 |
| 325 | 2 655 797 | 26 558 | 2 629 239 | 185 920 |
| 350 | 2 656 297 | 26 563 | 2 629 734 | 185 918 |
| 400 | 2 650 797 | 26 508 | 2 624 289 | 212 029 |
| 500 | 2 633 297 | 26 333 | 2 606 964 | 289 763 |
| 600 | 2 623 223 | 26 232 | 2 596 990 | 340 912 |
| 700 | 2 606 497 | 26 065 | 2 580 432 | 416 974 |
| 800 | 2 595 697 | 25 957 | 2 569 740 | 467 161 |
| 900 | 2 579 997 | 25 800 | 2 554 197 | 541 553 |
| 1 000 | 2 568 797 | 25 688 | 2 543 109 | 590 828 |
| 1 250 | 2 532 297 | 25 323 | 2 506 974 | 759 712 |
| 1 500 | 2 499 897 | 24 998 | 2 474 898 | 901 410 |
| 1 750 | 2 472 347 | 24 723 | 2 447 624 | 1 038 276 |
| 2 000 | 2 443 197 | 24 432 | 2 418 765 | 1 172 730 |
| 2 250 | 2 485 507 | 2 | 2 485 505 | 1 240 300 |
| 2 331 | 2 480 597 | 1 | 2 480 596 | 1 242 785 |
| 2 332 | 2 483 072 | 0 | 2 483 072 | 1 241 550 |
| 2 500 | 2 483 072 | 0 | 2 483 072 | 1 241 550 |
| 3 000 | 2 483 072 | 0 | 2 483 072 | 1 241 550 |
| No TMU | 2 483 072 | 0 | 2 483 072 | 1 241 550 |

Table D.4: Processor utilisation results, Task 2, 1% failure rate, Or1ksim

| | | Task 2 | | |
|---|---|---|---|---|
| Compare | Started | Exception | Completed | For-loop |
| 50 | 503 770 | 1 012 659 | 14 | 0 |
| 61 | 505 910 | 1 011 830 | 21 | 0 |
| 62 | 860 570 | 870 253 | 41 | 0 |
| 67 | 497 473 | 994 857 | 492 403 | 0 |
| 80 | 842 794 | 842 856 | 834 341 | 0 |
| 100 | 842 241 | 842 261 | 833 776 | 8 422 |
| 120 | 842 266 | 842 269 | 833 821 | 8 422 |
| 130 | 841 699 | 841 699 | 833 282 | 16 834 |
| 134 | 841 704 | 841 704 | 833 287 | 16 832 |
| 135 | 2 676 170 | 28 026 | 2 649 409 | 53 519 |
| 140 | 2 690 035 | 27 807 | 2 663 135 | 53 787 |
| 150 | 2 704 058 | 27 518 | 2 677 012 | 54 077 |
| 175 | 2 682 373 | 27 168 | 2 655 469 | 80 460 |
| 200 | 2 678 994 | 27 091 | 2 652 014 | 80 360 |
| 250 | 2 671 594 | 27 023 | 2 644 876 | 133 570 |
| 270 | 2 663 530 | 26 932 | 2 636 895 | 133 171 |
| 290 | 2 666 786 | 26 768 | 2 640 119 | 159 992 |
| 300 | 2 660 394 | 26 678 | 2 633 790 | 159 599 |
| 325 | 2 656 502 | 26 565 | 2 629 937 | 185 920 |
| 350 | 2 656 279 | 26 562 | 2 629 717 | 185 918 |
| 400 | 2 650 497 | 26 505 | 2 623 991 | 212 029 |
| 500 | 2 634 455 | 26 344 | 2 608 111 | 289 763 |
| 600 | 2 622 394 | 26 224 | 2 596 170 | 340 912 |
| 700 | 2 606 279 | 26 062 | 2 580 217 | 416 974 |
| 800 | 2 595 402 | 25 954 | 2 569 447 | 467 161 |
| 900 | 2 578 936 | 25 789 | 2 553 147 | 541 553 |
| 1 000 | 2 568 912 | 25 689 | 2 543 223 | 590 828 |
| 1 250 | 2 532 474 | 25 324 | 2 507 149 | 759 712 |
| 1 500 | 2 503 994 | 25 040 | 2 478 954 | 901 410 |
| 1 750 | 2 472 094 | 24 721 | 2 447 373 | 1 038 276 |
| 2 000 | 2 443 194 | 24 432 | 2 418 762 | 1 172 730 |
| 2 250 | 2 480 673 | 5 | 2 480 672 | 1 240 300 |
| 2 331 | 2 485 594 | 0 | 2 485 593 | 1 242 785 |
| 2 332 | 2 483 124 | 0 | 2 483 123 | 1 241 550 |
| 2 500 | 2 483 124 | 0 | 2 483 123 | 1 241 550 |
| 3 000 | 2 483 124 | 0 | 2 483 123 | 1 241 550 |
| No TMU | 2 483 124 | 0 | 2 483 123 | 1 241 550 |

Table D.5: Processor utilisation results, Task 1, 0.1% failure rate, Or1ksim

| | | Task 1 | | |
|---|---|---|---|---|
| Compare | Started | Exception | Completed | For-loop |
| 50 | 505 811 | 1 012 153 | 2 | 0 |
| 61 | 506 018 | 1 012 056 | 34 | 0 |
| 62 | 866 301 | 868 350 | 58 | 0 |
| 67 | 497 621 | 995 200 | 497 078 | 0 |
| 80 | 842 795 | 842 998 | 841 916 | 0 |
| 100 | 842 839 | 842 874 | 841 867 | 842 |
| 120 | 842 903 | 842 903 | 841 986 | 842 |
| 130 | 842 843 | 842 843 | 842 001 | 1 684 |
| 134 | 842 810 | 842 810 | 841 968 | 1 684 |
| 135 | 2 751 997 | 4 054 | 2 749 245 | 5 489 |
| 140 | 2 793 997 | 3 560 | 2 791 203 | 5 412 |
| 150 | 2 569 169 | 3 013 | 2 566 600 | 5 864 |
| 175 | 2 874 997 | 3 130 | 2 872 063 | 7 878 |
| 200 | 2 659 348 | 2 876 | 2 656 534 | 8 523 |
| 250 | 2 502 864 | 2 841 | 2 500 356 | 14 975 |
| 270 | 2 990 997 | 3 290 | 2 988 006 | 12 535 |
| 290 | 2 759 147 | 2 892 | 2 756 388 | 16 433 |
| 300 | 2 734 220 | 2 907 | 2 731 486 | 16 583 |
| 325 | 2 748 997 | 2 749 | 2 746 248 | 19 233 |
| 350 | 2 748 997 | 2 749 | 2 746 248 | 19 232 |
| 400 | 2 749 485 | 2 749 | 2 746 735 | 21 972 |
| 500 | 2 744 858 | 2 744 | 2 742 114 | 30 227 |
| 600 | 2 746 440 | 2 746 | 2 743 694 | 35 672 |
| 700 | 2 739 824 | 2 739 | 2 737 085 | 43 951 |
| 800 | 2 739 412 | 2 739 | 2 736 673 | 49 410 |
| 900 | 2 741 806 | 2 741 | 2 739 065 | 57 513 |
| 1 000 | 2 765 398 | 2 765 | 2 762 633 | 62 399 |
| 1 250 | 2 734 997 | 2 735 | 2 732 262 | 82 044 |
| 1 500 | 2 735 826 | 2 735 | 2 733 091 | 98 167 |
| 1 750 | 2 729 997 | 2 730 | 2 727 267 | 114 450 |
| 2 000 | 2 710 519 | 2 710 | 2 707 809 | 131 424 |
| 2 250 | 2 729 114 | 0 | 2 729 114 | 136 450 |
| 2 331 | 2 729 114 | 0 | 2 729 114 | 136 450 |
| 2 332 | 2 729 114 | 0 | 2 729 114 | 136 450 |
| 2 500 | 2 729 114 | 0 | 2 729 114 | 136 450 |
| 3 000 | 2 729 114 | 0 | 2 729 114 | 136 450 |
| No TMU | 2 729 114 | 0 | 2 729 114 | 136 450 |

Table D.6: Processor utilisation results, Task 2, 0.1% failure rate, Or1ksim

| | | Task 2 | | |
|---|---|---|---|---|
| Compare | Started | Exception | Completed | For-loop |
| 50 | 505 816 | 1 012 179 | 9 | 0 |
| 61 | 506 040 | 1 012 102 | 32 | 0 |
| 62 | 866 539 | 868 494 | 49 | 0 |
| 67 | 497 412 | 994 693 | 496 676 | 0 |
| 80 | 843 014 | 843 025 | 842 163 | 0 |
| 100 | 842 978 | 842 988 | 842 047 | 842 |
| 120 | 842 942 | 842 942 | 842 060 | 842 |
| 130 | 842 885 | 842 885 | 842 043 | 1 684 |
| 134 | 842 918 | 842 918 | 842 076 | 1 684 |
| 135 | 2 745 839 | 4 049 | 2 743 094 | 5 489 |
| 140 | 2 706 276 | 3 469 | 2 703 570 | 5 412 |
| 150 | 2 932 742 | 3 296 | 2 929 810 | 5 864 |
| 175 | 2 626 324 | 2 909 | 2 623 661 | 7 878 |
| 200 | 2 841 994 | 3 168 | 2 838 846 | 8 523 |
| 250 | 2 995 994 | 3 209 | 2 992 987 | 14 975 |
| 270 | 2 507 865 | 2 757 | 2 505 358 | 12 535 |
| 290 | 2 738 994 | 2 938 | 2 736 255 | 16 433 |
| 300 | 2 763 994 | 2 891 | 2 761 230 | 16 583 |
| 325 | 2 748 660 | 2 748 | 2 745 912 | 19 233 |
| 350 | 2 748 691 | 2 748 | 2 745 943 | 19 232 |
| 400 | 2 746 994 | 2 747 | 2 744 247 | 21 972 |
| 500 | 2 747 994 | 2 748 | 2 745 246 | 30 227 |
| 600 | 2 743 994 | 2 744 | 2 741 250 | 35 672 |
| 700 | 2 746 994 | 2 747 | 2 744 247 | 43 951 |
| 800 | 2 744 994 | 2 745 | 2 742 249 | 49 410 |
| 900 | 2 738 994 | 2 739 | 2 736 255 | 57 513 |
| 1 000 | 2 712 994 | 2 713 | 2 710 281 | 62 399 |
| 1 250 | 2 734 994 | 2 734 | 2 732 259 | 82 044 |
| 1 500 | 2 726 994 | 2 727 | 2 724 267 | 98 167 |
| 1 750 | 2 725 659 | 2 725 | 2 722 934 | 114 450 |
| 2 000 | 2 737 994 | 2 738 | 2 735 256 | 131 424 |
| 2 250 | 2 729 176 | 0 | 2 729 176 | 136 450 |
| 2 331 | 2 729 176 | 0 | 2 729 176 | 136 450 |
| 2 332 | 2 729 176 | 0 | 2 729 176 | 136 450 |
| 2 500 | 2 729 176 | 0 | 2 729 176 | 136 450 |
| 3 000 | 2 729 176 | 0 | 2 729 176 | 136 450 |
| No TMU | 2 729 176 | 0 | 2 729 176 | 136 450 |

Table D.7: Processor utilisation results, Task 1, 1% failure rate, FPGA

| | | Task 1 | | |
|---|---|---|---|---|
| Compare | Started | Exception | Completed | For-loop |
| 100 | 49 580 | 299 517 | 0 | 0 |
| 200 | 130 517 | 263 362 | 2 | 0 |
| 300 | 219 711 | 222 961 | 8 | 0 |
| 312 | 212 721 | 215 822 | 210 336 | 1 |
| 325 | 212 752 | 215 789 | 210 370 | 0 |
| 350 | 214 771 | 214 911 | 212 433 | 0 |
| 400 | 213 107 | 215 361 | 210 959 | 2 131 |
| 500 | 214 273 | 214 275 | 212 090 | 2 142 |
| 550 | 214 116 | 214 116 | 211 964 | 4 278 |
| 575 | 214 106 | 214 106 | 211 964 | 4 276 |
| 578 | 214 790 | 214 790 | 212 183 | 460 |
| 579 | 621 497 | 7 440 | 615 282 | 12 603 |
| 580 | 627 497 | 7 477 | 621 222 | 12 484 |
| 586 | 616 597 | 7 422 | 610 431 | 12 637 |
| 590 | 617 197 | 7 426 | 611 025 | 12 688 |
| 600 | 618 937 | 7 435 | 612 748 | 12 655 |
| 700 | 620 990 | 6 502 | 614 779 | 12 698 |
| 800 | 625 897 | 6 506 | 619 575 | 18 812 |
| 900 | 636 799 | 6 594 | 630 284 | 24 511 |
| 1 000 | 629 697 | 6 505 | 623 297 | 24 820 |
| 1 100 | 622 397 | 6 477 | 616 150 | 31 223 |
| 1 158 | 624 166 | 6 451 | 617 925 | 31 102 |
| 1 172 | 619 606 | 6 444 | 613 410 | 31 201 |
| 1 250 | 624 619 | 6 395 | 618 373 | 37 203 |
| 1 400 | 622 728 | 6 227 | 616 500 | 37 355 |
| 1 500 | 621 297 | 6 213 | 615 084 | 43 473 |
| 1 548 | 621 263 | 6 212 | 615 051 | 43 504 |
| 2 000 | 617 197 | 6 172 | 611 025 | 61 719 |
| 3 000 | 610 994 | 6 109 | 604 885 | 91 537 |
| 4 000 | 602 596 | 6 025 | 596 571 | 126 605 |
| 5 000 | 595 842 | 5 958 | 589 883 | 160 414 |
| 6 000 | 588 774 | 5 887 | 582 887 | 188 437 |
| 7 000 | 581 697 | 5 816 | 575 880 | 220 895 |
| 8 000 | 574 597 | 5 746 | 568 851 | 252 497 |
| 9 000 | 567 997 | 5 680 | 562 317 | 278 844 |
| 9 250 | 567 597 | 5 676 | 567 505 | 283 444 |
| 9 500 | 577 805 | 99 | 577 803 | 288 338 |
| 9 750 | 577 997 | 3 | 577 995 | 288 448 |
| 10 000 | 577 450 | 3 | 577 450 | 288 700 |
| 11 000 | 577 442 | 0 | 577 442 | 288 700 |
| 12 000 | 577 442 | 0 | 577 442 | 288 700 |
| No TMU | 577 442 | 0 | 577 442 | 288 700 |

Table D.8: Processor utilisation results, Task 2, 1% failure rate, FPGA

| | | Task 2 | | |
|---|---|---|---|---|
| Compare | Started | Exception | Completed | For-loop |
| 100 | 49 603 | 299 583 | 0 | 0 |
| 200 | 130 532 | 263 372 | 2 | 0 |
| 300 | 219 973 | 223 061 | 9 | 0 |
| 312 | 212 800 | 215 840 | 210 400 | 1 |
| 325 | 212 871 | 215 822 | 210 463 | 0 |
| 350 | 214 711 | 214 781 | 212 240 | 0 |
| 400 | 213 172 | 215 411 | 211 024 | 2 131 |
| 500 | 214 274 | 214 274 | 212 065 | 2 142 |
| 550 | 214 104 | 214 104 | 211 928 | 4 278 |
| 575 | 214 112 | 214 112 | 211 971 | 4 276 |
| 578 | 214 772 | 214 772 | 212 148 | 460 |
| 579 | 630 647 | 7 485 | 624 341 | 12 603 |
| 580 | 624 620 | 7 462 | 618 374 | 12 484 |
| 586 | 632 294 | 7 517 | 625 971 | 12 637 |
| 590 | 634 951 | 7 497 | 628 602 | 12 688 |
| 600 | 633 094 | 7 515 | 626 763 | 12 655 |
| 700 | 634 894 | 6 542 | 628 543 | 12 698 |
| 774 | 617 129 | 6 484 | 612 870 | 18 557 |
| 800 | 627 202 | 6 509 | 620 873 | 18 812 |
| 900 | 613 294 | 6 484 | 606 864 | 24 511 |
| 1 000 | 620 685 | 6 446 | 614 345 | 24 820 |
| 1 100 | 625 038 | 6 492 | 618 782 | 31 223 |
| 1 158 | 623 394 | 6 489 | 617 160 | 31 102 |
| 1 172 | 624 794 | 6 448 | 618 546 | 31 201 |
| 1 250 | 620 394 | 6 369 | 614 190 | 37 203 |
| 1 400 | 622 894 | 6 229 | 616 665 | 37 355 |
| 1 500 | 621 633 | 6 216 | 615 417 | 43 473 |
| 1 548 | 621 594 | 6 216 | 615 378 | 43 504 |
| 2 000 | 617 529 | 6 175 | 611 354 | 61 719 |
| 3 000 | 610 294 | 6 103 | 604 191 | 91 537 |
| 4 000 | 602 894 | 6 029 | 596 865 | 126 605 |
| 5 000 | 594 394 | 5 944 | 588 450 | 160 414 |
| 6 000 | 588 894 | 5 889 | 583 005 | 188 437 |
| 7 000 | 581 394 | 5 814 | 575 580 | 220 895 |
| 8 000 | 574 190 | 5 741 | 568 448 | 252 497 |
| 9 000 | 569 168 | 5 691 | 563 476 | 278 844 |
| 9 250 | 566 971 | 5 669 | 566 884 | 283 444 |
| 9 500 | 576 763 | 102 | 576 757 | 288 338 |
| 9 750 | 576 920 | 5 | 576 917 | 288 448 |
| 10 000 | 577 470 | 3 | 577 469 | 288 700 |
| 11 000 | 577 490 | 0 | 577 490 | 288 700 |
| 12 000 | 577 490 | 0 | 577 490 | 288 700 |
| No TMU | 577 490 | 0 | 577 490 | 288 700 |