Tobias Peter Scheipel, BSc

System-Aware Performance Monitoring Unit für die RISC-V-Architektur

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Masterstudium Telematik

eingereicht an der

Technischen Universität Graz

Betreuer

Univ.-Prof. Dipl.-Inf. Univ. Dr.rer.nat. Marcel Carsten Baunach
Mitwirkender: Dipl.-Ing. Fabian Mauroner, BSc

Institut für Technische Informatik

Graz, April 2017

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Datum Unterschrift

Danksagung

Diese Masterarbeit wurde im Studienjahr 2016/17 am Institut für Technische Informatik der Technischen Universität Graz durchgeführt.

An dieser Stelle möchte ich all jenen danken, die mich im Rahmen dieser Masterarbeit unterstützt und begleitet haben. Besonderer Dank gilt meinen Betreuern, Herrn Univ.-Prof. Marcel Baunach sowie Herrn Dipl.-Ing. Fabian Mauroner, auf deren Auskunft und fachliche Expertise während der Arbeit jederzeit Verlass war. Durch diesen ständigen Input konnte die Motivation sowie das Interesse auf einem hohen Niveau gehalten werden.

Des Weiteren gilt mein Dank den Mitarbeitern der *Embedded Automotive Systems*-Gruppe des Instituts für Technische Informatik, die mir neue Ideen und Hinweise zur Vervollständigung dieser Arbeit lieferten.

Ein besonderer Dank gilt auch meiner Familie und insbesondere meinen Eltern, die mich während meines Studienlebens immer unterstützt und gefördert haben.

Darüber hinaus möchte ich mich bei meinen Freunden bedanken, welche mir Rückhalt gaben und auch für die ein oder andere Ablenkung – sei es in der virtuellen oder realen Welt – zur Verfügung standen, um meinen Kopf frei zu bekommen. Gleiches gilt für meine Teamkollegen und Coaches der Graz Giants, die mit Geduld und Verständnis glänzten.

Abschließend möchte ich mich bei allen Studienkollegen und Freunden für die schönen Studienjahre bedanken.

Kurzfassung

Mit steigender Komplexität der Software von eingebetteten Systemen wird es immer wichtiger, die Performance dieser Softwaresysteme bereits im Entwicklungsprozess miteinzubeziehen. Oftmals mangelt es hier jedoch an Möglichkeiten, Laufzeiten oder Ereignisse gezielt zu messen und/oder zu zählen. Überwachung zur Laufzeit ist ebenfalls relevant, um dynamisch auf interne und externe Ereignisse reagieren zu können.
Dies gilt vor allem für Systeme, die mit mehreren nebenläufigen Tasks arbeiten, welche diverse Abhängigkeiten beinhalten können. Diese Abhängigkeiten können sowohl Tasks untereinander, als auch externe Ressourcen betreffen.
Ein weiteres Problem ist, dass Messungen während der Entwicklungszeit meist durch einen Eingriff in das zu entwickelnde System erfolgen. Das führt in weiterer Folge zu einer Verzerrung der Messergebnisse, da das endgültige System ohne diese Eingriffe – und somit oft performanter ausgeliefert wird, als es zum Entwicklungszeitpunkt vorlag.

Ziel dieser Arbeit ist es nun, ein Modul in einer Hardwarebeschreibungssprache zu entwickeln, welches in der Lage ist, ohne direkte Veränderungen des Systems Laufzeiten und Ereignisse sowohl taskabhängig, als auch -unabhängig zu messen und dem Programmierer über eine einfache Schnittstelle zur Verfügung zu stellen.
Großes Augenmerk soll dabei auf die Skalierbarkeit, die Plattformunabhängigkeit hinsichtlich Prozessor und Betriebssystem sowie auf die einfache Erweiterbarkeit gelegt werden.

Das finale Hardwaremodul ist anschließend in ein bestehendes System aus einem Softcore-Prozessor und einem minimalen Betriebssystem zu integrieren und zu testen.

Abstract

Due to increases in the complexity of the software of embedded systems, it is more important than ever to take performance aspects of these software systems into account. This should happen very soon in the development process. Often runtimes and events are not easily countable or measurable due to a lack of functionality in these systems. Runtime monitoring is also relevant in terms of reacting to internal and external events dynamically.
This applies especially for systems with multiple tasks with dependencies within the tasks or to external resources.
Another problem is that measurements during the development process are often done by interfering the system as a whole. This method leads to biases in the measurement results, because the finalized system gets deployed without these interfering functionalities – and can therefor work more efficiently than the development system.

The scope of the present work is to develop a module in a hardware description language, which is able to measure runtime and events taskaware and unaware without interfering the system. The measurements of this module must be handed to the programmer through an easy accessible interface.
The main focuses of the project are the scalability, platform independency concerning processor and operating system as well as easy extendability.

The final hardware module has to be integrated into an existing system containing a softcore processor and a minimal operating system. The resulting system must be tested.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ASIC Application Specific Integrated Circuit

CLB Configurable Logic Block

CSR Control and Status Register

DMIPS Dhrystone Million Instructions Per Second

EMU Event Monitoring Unit

FIFO First-In-First-Out

FPGA Field Programmable Gate Array

ISA Instruction Set Architecture

LUT Look-Up-Table

MSR Model Specific Register

OS Operating System

PC Program Counter

PMC Performance Monitoring Counter

PMU Performance Monitoring Unit

RAM Random Access Memory

RISC Reduced Instruction Set Computer

SIMD Single Instruction, Multiple Data

TCB Task Control Block

TP Task Pointer

Kapitel 1

Einleitung

Diese Arbeit beschäftigt sich mit dem Entwurf und der Entwicklung einer Lösung zum Messen von Laufzeiten und Ereignissen in einem eingebetteten System bestehend aus Hard- und Softwarekomponenten.

Die Einleitung enthält die Motivation und den Hintergrund der Arbeit, ihre Aufgabenstellung und Zielsetzung sowie die Gliederung dieses Dokuments.

1.1 Motivation

Das Ermitteln und Überwachen der Laufzeiten und Ereignisse in einem eingebetteten System werden heutzutage immer wichtiger. Moderne Prozessoren beinhalten in ihrer Architektur hierfür bereits seit geraumer Zeit Funktionalitäten, um das Leistungsvermögen eines Systems mit sogenannten Profilern oder Performance Monitoring Units (PMUs) festzuhalten und gegebenenfalls auf gelieferte Ergebnisse zu reagieren.
Bei Prozessoren von Intel ist dies im Kapitel 18 *Performance Monitoring* in [4] näher beschrieben, für ARM Cortex-A5 Prozessoren ist die Funktionalität im Kapitel 10 *Performance Monitoring Unit* in [5] erläutert.

Da jedoch in der vorliegenden Arbeit auf den Forschungsprozessor RISC-V[1] zurückgegriffen wird, der an der University of California, Berkeley, entwickelt wurde, sind solche Funktionalitäten zwar grundlegend definiert, jedoch in keiner Implementierung beinhaltet.
Der Prozessor an sich ist als freie und offene Reduced Instruction Set Computer (RISC) Instruction Set Architecture (ISA) definiert, daher gibt es mehrere Implementierungen[2] von verschiedenen Institutionen, die auf dem Definitionsdokument aufbauen. Dabei sind sowohl diskrete Hardwareausführungen als

[1] https://riscv.org
[2] z.B. [6], [7] und [8]

auch Implementierungen in verschiedenen Hardwarebeschreibungssprachen wie VHDL oder Verilog vorhanden, welche offen zur Verfügung stehen und von jedermann verwendet und angepasst werden können. Weiters gibt es Compiler und Toolchains, um auch Programmcode in C oder Assembler auf der Plattform lauffähig machen zu können. Die verschiedenen Varianten und Methoden werden in den späteren Kapiteln genauer erläutert.

Im Gegensatz zu einer Hardwarelösung gibt es auch die Möglichkeit, die Leistung eines Systems über Software zu messen. Hier läuft man jedoch Gefahr, das System als Ganzes durch Eingriffe zu verändern und somit die Messergebnisse zu verfälschen. Weiters verkompliziert sich die Situation, wenn man ein Betriebssystemkonstrukt auf dem System laufen hat, welches mehrere nebenläufige Tasks beinhaltet. Durch softwarebasierte Messvorgänge kann im schlechtesten Fall sogar der gesamte Taskablauf beeinträchtigt werden.

Ein weiterer Vorteil einer Hardwarelösung ist, dass sie softwareunabhängiger betreibbar ist und selbst in ausgelieferten Systemen beinhaltet bleiben kann.

1.2 Zielsetzung

Ziel der Arbeit ist es nun, verschiedene Implementierungen der RISC-V-Architektur zu evaluieren und eine passende Plattform auszuwählen. Diese soll dann um eine Überwachungseinheit erweitert werden, welche als externer Beobachter nicht direkt in das Softwaresystem eingreift, jedoch trotzdem relevante Informationen über das aktuelle Verhalten liefern kann.

Bei der Erstellung dieser Erweiterung, im Nachfolgenden als Performance Monitoring Unit (PMU) bezeichnet, soll besonderes Augenmerk auf die Skalierbarkeit und die Erweiterbarkeit gelegt werden. Wichtig ist auch, dass das Modul mit verschiedenen Varianten des RISC-V-Prozessors arbeiten kann. Idealerweise ist das Modul sogar insoweit plattformunabhängig, dass es auch mit anderen Softcores[3] arbeiten kann.

Das Ergebnis der Entwicklungsarbeit kann dann anschließend einerseits simuliert, andererseits direkt auf einem Field Programmable Gate Array (FPGA) getestet werden. Das erfolgt dabei mit dem Development Board Nexsy4[4] von Digilent, welches auf der Artix-7[5]-FPGA-Familie von Xilinx basiert.

[3] Prozessoren, die mittels einer logischen Synthese (z.B. mit einer Hardwarebeschreibungssprache wie Verilog oder VDHL) erstellt wurden

[4] https://reference.digilentinc.com/reference/programmable-logic/nexys-4/start

[5] https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html

1.3 Gliederung

In Kapitel 2 wird auf ähnliche Arbeiten und Ideen eingegangen. Dabei werden sowohl verschiedene Aspekte des Profilings als auch der Prozessorarchitekturen behandelt und Unterschiede herausgearbeitet.

Kapitel 3 beschäftigt sich mit der Evaluation der Spezifikation und verschiedener Implementierungen der RISC-V-Architektur. Es wird untersucht, welche der bestehenden Versionen am besten für eine Anpassung geeignet ist.

In Kapitel 4 wird die Entwicklung des Hardwaremoduls erläutert. Einzelne Designentscheidungen sind hierbei genauso wichtig wie der Hinblick auf die Anbindung an das spätere Softwaresystem.

Das darauf folgende Kapitel 5 beschäftigt sich damit, wie die Verknüpfung von Hard- und Software in diesem System arbeitet. Hierbei wird auf Betriebssystemebene erarbeitet, wie das System softwareseitig funktioniert. Des Weiteren wird behandelt, wie der Benutzer mit dem zuvor entwickelten Modul umgeht und wie er es effizient in seine eigene Entwicklung eingliedern kann.

In Kapitel 6 werden verschiedene Testsysteme betrachtet und mit geeigneten Methoden durchgemessen. Anschließend werden die Ergebnisse mit den vom entwickelten Modul gelieferten Werten verglichen. Die Hardware wird ebenfalls analysiert und auf Ressourcenverbrauch getestet.

Abschließend wird in Kapitel 7 ein Ausblick auf zukünftige mögliche Weiterentwicklungen gegeben. Außerdem wird die Arbeit zusammengefasst.

Kapitel 2

Related Work

In diesem Kapitel werden bestehende Lösungen zur Messung von Laufzeiten und anderen Informationen von Hard- und Softwaresystemen betrachtet. Dabei werden sowohl Lösungen von großen Prozessorherstellern wie Intel und ARM, als auch Entwicklungen aus Forschung und Wirtschaft analysiert. Im weiteren Verlauf soll auch noch darauf eingegangen werden, was diese bereits entwickelten Ansätze nicht abdecken und wo ein Innovationspotential besteht, welches in diese Arbeit einfließen soll.

2.1 Allgemeines

Prinzipiell werden nach Sprunt [9] meist Ereignisse während der Laufzeit einer Applikation gemessen, mit welchen man Rückschlüsse auf dessen Leistungsfähigkeit schließen kann. Oftmals werden die Ergebnisse einer solchen Messung – auch Performance Profile genannt – zur Ermittlung des elektrischen Leistungsverbrauchs verwendet. Aufgrund der so ermittelten Werte können Anpassungen der verwendeten Algorithmen oder Betriebssysteme durchgeführt werden, um diese Werte zu optimieren. Hierbei muss jedoch stark unterschieden werden, ob es sich um einen Hochleistungsprozessor in einem PC oder einen Mikroprozessor in einem eingebetteten System handelt, da hier unterschiedliche Anforderungen gelten.

Einzuteilen sind diese Performance-Profile grundlegend in zeit- und ereignisbasierte Profile. Mit zeitbasierten Profilen lassen sich Bereiche identifizieren, für die Applikationen lange brauchen, während ereignisbasierte Profile dazu dienen, Informationen über Ereignisse und deren zugrundeliegenden Programmabschnitte zu liefern. Solche Ereignisse können in modernen Prozessoren etwa Speicherzugriffe, Pipeline-Stalls[1], Cache-Misses und Ressourcenverwendungen

[1] Verwerfen des Inhalts der Pipeline

sein. Für Mikroprozessoren könnte man hier als Beispiel externe Interrupts oder Taskwechsel anführen.

2.2 Intel Performance Monitoring Unit

Jeder moderne Intel-Prozessor seit dem Pentium beinhaltet heutzutage eine sogenannte PMU. Patil et al. zeigen die Architektur dieser PMU in [10] als Zusammensetzung aus einem Performance Monitoring Counter (PMC) und einigen Model Specific Registers (MSRs) an. Der PMC hält die Zählwerte für das Auftreten gewisser Ereignisse, während die MSRs zum konfigurieren und steuern der Einheit dienen.

Man kann diese Zähler so konfigurieren, dass sie nur bei gewissen Ereignissen zählen, wie von Singh et al. [11] gezeigt. Die Anzahl der unterschiedlichen Zähler für die verschiedenen Intel-Architekturen findet man in [12] und entspricht zum Beispiel für einen Nehalem-Core[2] vier allgemein verwendbare Zähler und drei Zähler für Instruktionen und verschiedene Laufzeiten. Daneben noch vier Kontrollregister zum Selektieren von Ereignissen und für die Konfiguration der allgemein verwendbaren Zähler. Alle Register sind als 32-Bit-Register ausgelegt.

Um diese Zähler nun in einer Anwendung verwenden zu können, kann zum Beispiel das Linux Subsystem *perf* oder ein eigener Treiber verwendet werden. Die Anwendung wird in [10] und [11] ausführlich erläutert. Man kommt dabei zum Schluss, dass die Zähler zwar global verwendet und auch für gewisse Hardwarethreads je Prozessorkern individuell konfiguriert werden können, jedoch keinerlei Aussage über das Verhalten eines dezidierten Softwareteils auf dem System treffen können.

2.3 ARM Performance Monitoring Unit

Die PMU in einem ARM-Prozessor arbeitet ähnlich wie die zuvor gezeigte Intel-Variante. Für einen Cortex-A5-Prozessor sind es nach [5] zwei 32-Bit-Zählregister, welche jedes beliebige Prozessorereignis aufzeichnen können. Daneben gibt es noch ein einzelnes 32-Bit-Register zum Messen von Prozessortaktzyklen. Konfigurierbar ist die Einheit über diverse Konfigurations- und Kontrollregister.

Zugreifen kann man auf diese Zähler und die korrespondierenden Kontrollregister einerseits über ein internes Interface, welches nur für den Prozessor selbst relevant ist und andererseits über einen memory-mapped Speicherbereich.

[2] Intel-Prozessorfamilie

2.4 Ansätze für Performance Monitoring in anderen Prozessoren

Im Bereich der eingebetteten Systeme gibt es ebenfalls eine Vielzahl an Ansätzen, um Prozessoren überwachen zu können. Dabei konzentrieren sich bestehende Arbeiten hauptsächlich auf das Messen des Energieverbrauchs und der Leistungsfähigkeit auch in konfigurierbaren Systemen. In den beiden in Folge behandelten Arbeiten wird konkret auf verschiedene Lösungen in Multicore-Plattformen eingegangen.

2.4.1 Skalierbares Performance Monitoring für Multiprozessorsysteme

Ambrose et al. schlagen in ihrer Arbeit [1] vor, eine skalierbare Event Monitoring Unit (EMU) einzubinden, welche zentral gehalten wird und die Statusinformationen der Prozessoren, aber auch von First-In-First-Out (FIFO)-Buffern über eigene Eingänge erhält und weiterverarbeitet. Die Architektur hierfür ist in Abbildung 2.1 zu sehen.

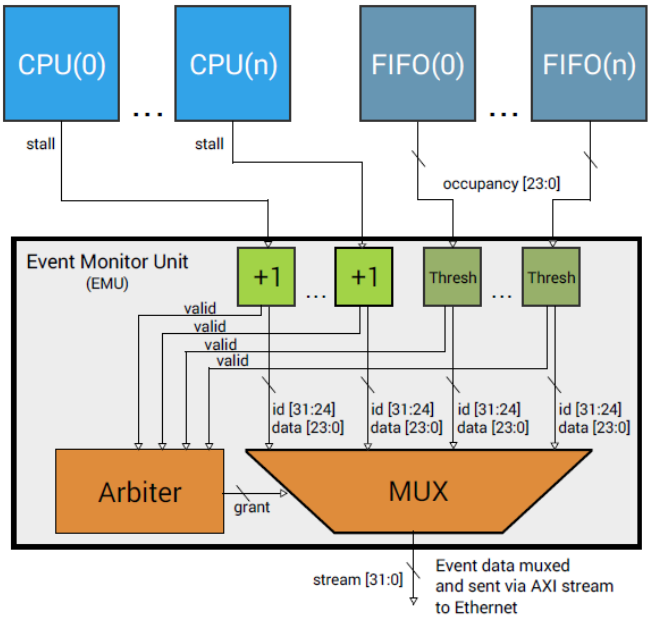


Abbildung 2.1: Die Darstellung der Architektur nach Ambrose et al. [1].

Konkret werden dabei für jeden Prozessorkern die Anzahl an Pipeline-Stalls und für jeden FIFO die Belegungsrate gemessen und die Ergebnisse an einen Bus

weitergegeben. Die EMU muss demnach je nach Kern- und FIFO-Anzahl im System unterschiedlich konfiguriert werden.
Der Hardwareaufwand kann so sehr klein gehalten werden, da die Einheit nur einmal zentral vorhanden ist und nicht jeder Core eine eigene benötigt. Ein direkter Systemeingriff ist nicht nötig, das Grundsystem bleibt wie gehabt. Laufzeiteinbußen sind bei dieser Lösung ebenfalls nicht zu erwarten, da die EMU nur als Beobachter fungiert.
Ein Nachteil dieser Lösung ist jedoch, dass für jedes weitere Signal, das man überwachen möchte, ein eigener Zähler verwendet werden muss, da keine weitere Konfigurationsmöglichkeit vorhanden ist. Des Weiteren ist es nicht möglich, Tasks des Betriebssystems zu überwachen.

2.4.2 Performance Monitoring für LEON3-Multicore-Systeme

Eine weitere Arbeit zu eingebetteten Systemen mit Performance Monitoring liefern Ho et al. [2]. Das System baut hier auf einem System aus mehreren LEON3-Prozessoren auf. Die entwickelte PMU ist hier an das ARM-Cortex-A9-System angelehnt und beinhaltet Event-Zähler sowie fixe Taskzyklenzähler. Dabei wird speziell auf das Zusammenspiel von Hard- und Software geachtet, um effiziente Ergebnisse zu erzielen. Die Architektur des Systems ist in Abbildung 2.2 dargestellt.

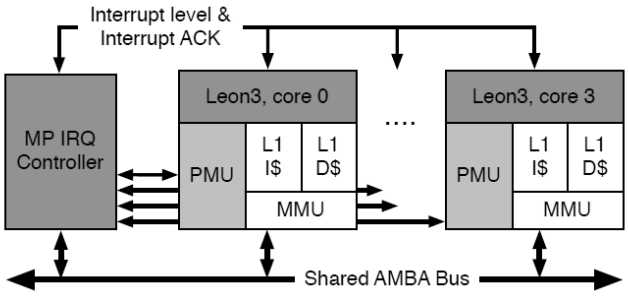


Abbildung 2.2: Die Darstellung der Architektur nach Ho et al. [2].

Hier sieht man, dass jeder Prozessorkern seine eigene PMU beinhaltet. Zum Konfigurieren werden Register verwendet, welche mit dem erweiterten Address Space Identifier der LEON3-Architektur angesprochen werden können. Statusinformationen werden direkt an diversen Stellen der CPU abgegriffen und verarbeitet. Die Einheit kommuniziert über Interrupts mit dem Gesamtsystem.
Der Hardwareaufwand ist in dieser Lösung höher als in [1], jedoch können eine Vielzahl an Ereignissen miteinbezogen werden und somit eine detailliertere Messung erreicht werden. Möglichkeiten, Tasks direkt zu messen sind jedoch auch hier nicht vorhanden.

2.5 Fazit

Die hier gezeigten Arbeiten funktionieren alle nach einem ähnlichen Prinzip: Sie zeichnen hauptsächlich Ereignisse, nebensächlich Laufzeiten auf. In keiner der Lösungen wurde dezidiert Task-Awareness gezeigt, also kein System wurde so entwickelt, dass es mit dem Softwaresystem zusammenarbeitet.
Die Zähler sind außerdem immer explizit einer Messung zugeordnet und können zur Laufzeit nicht wiederverwendet werden. Werden zum Beispiel für verschiedene Tasks am System mehrere Zähler benötigt, ist es immer notwendig, diese direkt zu reservieren. Einzige Lösungsmöglichkeit wäre hier, mehr Zähler in das System einzubringen, was wiederum einen höheren Hardwareaufwand bedeuten würde.

Diese Betrachtungen führen dazu, dass es interessant wäre, eine PMU zu entwickeln, welche nicht nur Ereignisse, sondern auch Laufzeiten von Taskkonstrukten auf eingebetteten Systemen messen kann, um daraus Schlüsse zu ziehen. Um dies nicht in einem horrenden Hardwareaufwand enden zu lassen, sollten auch Überlegungen zur Rekonfiguration und Zwischenspeicherung von Messwerten zur Laufzeit angestellt werden.

Die Merkmale Task-Awareness und Wiederverwendung von Hardwarezählern durch Zwischenspeicherung ist hierbei als Neuheit zu sehen.

Kapitel 3

CPU Evaluation

Im Verlauf des *mosartMCU*-Projektes[1] der Arbeitsgruppe Embedded Automotive Systems des Instituts für Technische Informatik der Technischen Universität Graz soll ein Prozessor eingesetzt werden, der den von der University of California, Berkeley, entwickelten Befehlssatz *RISC-V* unterstützt.

In diesem Kapitel wird nun in weiterer Folge eine Übersicht über diese Architektur und dessen Befehlssatz gegeben. Darauffolgend werden einige Implementierungen in verschiedenen Hardwarebeschreibungssprachen vorgestellt, analysiert und verglichen.

3.1 RISC-V-Spezifikation

RISC-V ist eine offene und freie ISA, die von der University of California, Berkeley, von Waterman et al. in [13] und [3] definiert wurde und sich an etablierten RISC-Architekturen wie ARMv6 [14], Hitachi SH-4 [15], PA-RISC [16] oder SPARCv8 [17] orientiert. Im Unterschied zu vielen proprietären Befehlssätzen ist RISC-V jedoch im Hinblick auf Verwendung und Veränderung frei verfügbar. Zusätzlich ist die ISA extra dazu ausgelegt, einfach erweiterbar zu sein.

Der Standard kann dabei nach Waterman et al. [13] in verschiedenen Varianten mit unterschiedlichen Erweiterungen vorkommen. Die unterschiedlichen Bitbreiten für den Basis-Integerbefehlssatz sind 32, 64 und 128 Bit und werden mit RV32I, RV64I und RV128I gekennzeichnet. Zusätzlich zum minimalen I-Befehlssatz, der Integer-Instruktionen definiert, gibt es noch die in Folge angegebenen Erweiterungen.

[1] Multi-Core Operating-System-Aware Real-Time Microcontroller

- **M** für Integermultiplikationen und -divisionen
- **A** für Atomic-Instruktionen
- **F** für Single-Precision Floating-Point Operationen
- **D** für Double-Precision Floating-Point Operationen
- **IMAFD** zusammengefasst in **G**
- **Q** für Quad-Precision Floating-Point Operationen
- **L** für dezimale Floating-Point Operationen
- **C** für 16-Bit komprimierte Instruktionen
- **B** für Bitmanipulationen
- **T** für Transactional Memory
- **Q** für Packed-Single Instruction, Multiple Data (SIMD)-Erweiterungen

Die Definitionen für die einzelnen Befehlssätze können in [13] und [18] nachgelesen werden.

Eine CPU oder MCU, die den RISC-V-Befehlssatz implementiert, muss nach der minimalen Spezifikation 32 Register beinhalten. Dabei dürfen die arithmetischen, bitweisen Aufrufe sowie Subroutinesprünge nur diese Register referenzieren, um Speicherzugriffe zu vermeiden. Die Nullkonstante ist im Register x0 zu finden.

Neben den CPU-Registern existieren noch sogenannte Control and Status Register (CSR) nach [3]. Diese speziellen Register, auf welche mit privilegierten Instruktionen und Adressen zugegriffen werden kann, beinhalten diverse Einstellmöglichkeiten und Statusinformationen um den Prozessor zu konfigurieren und Informationen zu erhalten. Die zugehörigen Instruktionen zum Lesen und Schreiben der Werte sind atomar ausgelegt. Bekannte Funktionalitäten wie Timer und Counter, eine Systemuhr sowie Kontrollregister für die Floating-Point-Einheiten sind hier mit inbegriffen. Die Adressierung erfolgt nach einem gewissen Schema und ist nach User-, Supervisor-, Hypervisor- und Machine-Modus definiert. Zusätzlich dazu gibt es festgelegte Bereiche für exklusive Lesezugriffe. Hier sind auch Bereiche definiert, welche explizit von Erweiterungen genutzt werden können und sollen.
Insgesamt ist für die CSRs ein Adressraum von 12 Bit reserviert, was bis zu 4096 verschiedene Register erlaubt. Die oberen beiden Bits der Adresse legen dabei fest, ob das korrespondierende Register für exklusiven Lesezugriff (11) vorgesehen ist, oder ob darauf auch geschrieben werden darf (00, 01, 10).

3.2 Implementierungen

Für die vorliegende Arbeit muss ein Softcore in einer Hardwarebeschreibungssprache gefunden werden, der in vielerlei Hinsicht leicht erweiterbar ist. Da RISC-V ein offener Befehlssatz ist, existieren mehrere verschiedene Implementierungen mit gewissen Vor- und Nachteilen.

Dieses Kapitel gibt eine Einführung zu den Implementierungen *ORCA*, *RI5CY/PULP* und *Z-scale* beziehungsweise dessen Verilog-Äquivalent *V-scale*.

Im weiteren Verlauf werden diese noch miteinander Verglichen und eine Auswahl getroffen.

3.2.1 ORCA

ORCA ist eine Familie an konfigurierbaren VHDL-RISC-V-Implementierungen von VectorBlox, welche frei verwendet werden können[2].
Unterstützt wird der Befehlssatz in der Konfiguration RV32I und RV32IM. Zur Speicheranbindung kann ein Wishbone-Bus [19], ein AXI-Bus [20] oder ein Avalon-Bus [21] verwendet werden. Die Implementierung ist für FPGA-Nutzung optimiert. Konfigurierbar sind unter anderem die Multiplizier- und Dividiereinheit, die Anzahl der Zählregister sowie der Pipeline-Stufen.

3.2.2 RI5CY/PULP

Der RI5CY, definiert in [22], ist eine Implementierung in SystemVerilog, welche RV32IC voll unterstützt. Zusätzlich ist noch der Multiplizierer der M-Spezifikation beinhaltet.
Der Core wurde im Zuge des PULP-Projekts[3] an der ETH Zürich entwickelt. Das verwendete Businterface ist AXI und die Pipeline ist vierstufig. In dieser Implementierung sind viele Erweiterungen für Signalverarbeitungsanwendungen beinhaltet und dementsprechend für die Verwendung in diesem Gebiet angepasst.

3.2.3 Z-scale/V-scale

Der Z-scale ist ein RISC-V-Softcore in Chisel [23] der University of California, Berkeley, und wurde dort neben anderen RISC-V-Cores entwickelt.
Hierbei handelt es sich um eine RV32IM-Architektur mit einer Busanbindung über AHB-Lite Interconnect [24], alternativ kann auch AXI verwendet werden.

[2] https://github.com/VectorBlox/orca
[3] http://www.pulp-platform.org/

Die Pipeline ist dreistufig und es werden die Privilege Modi *Machine* und *User* unterstützt. Diese Implementierung ist speziell für die Verwendung als Mikrocontroller in eingebetteten Systemen optimiert und frei verfügbar[4]. Neben der Chisel-Variante gibt es noch einen Verilog-Klon des Cores[5], um eine breitere Verwendung zu gewährleisten.

3.2.4 Vergleich

Tabelle 3.1 zeigt einen Vergleich der oben angegebenen RISC-V-Varianten mit einigen Kenngrößen (Quellen: [6], [7] und [8]). In diesem Vergleich wird der Z-scale zugunsten des V-scale ausgelassen, da dieser die gleichen Eigenschaften hat, jedoch in der populäreren Sprache Verilog implementiert ist. Die Anzahl an verwendeten Look-Up-Tables (LUTs) auf einem FPGA sowie die Dhrystone Million Instructions Per Second (DMIPS), ein Leistungstest für Rechenleistung spielen dabei eine Rolle.

	ORCA	*RI5CY*	*V-scale*
Sprache	VHDL	SystemVerilog	Verilog
ISA	RV32I, RV32IM	RV32IC (partial M)	RV32IM
Bus	Wishbone, Avalon, AXI	AXI	AHB-Lite, AXI
DMIPSs	122	-	44
DMIPSs/MHz	0,98	-	1,35
LUTs	2354 LUT4	-	2678 LUT4

Tabelle 3.1: Vergleich verschiedener RISC-V Varianten nach Kenngrößen.

Der RI5CY scheidet bereits aufgrund der sehr starken Anpassung verglichen mit dem RISC-V Standard aus.
Die obigen Werte charakterisieren eine 4-stufige Pipeline beim ORCA und eine 3-stufige Pipeline beim V-scale. Deshalb können bei Ersterem höhere Frequenzen erreicht werden, was jedoch auch bedeutet, dass Zweiterer leistungsfähiger ist.
Aufgrund der Tatsache, dass es sich beim V-scale um den am wenigsten vom Standard abweichenden Core handelt, der noch dazu in der bevorzugten Hardwarebeschreibungssprache Verilog entwickelt wurde, fällt die Wahl in diesem Projekt auf diese Implementierung. Auch der Support hinsichtlich Compiler und Toolchains ist hier besser gegeben.

[4] https://github.com/ucb-bar/zscale
[5] https://github.com/ucb-bar/vscale

Kapitel 4

Performance Monitoring Unit - Hardware

Diese Kapitel beschäftigt sich mit der Entwicklung der Hardware der PMU für den ausgewählten V-scale-Softcore. Verwendete Hardwarebeschreibungssprache ist dabei – wie auch für den Core selbst – Verilog.

Im weiteren Verlauf wird zunächst auf die gewünschten Funktionen eingegangen, anschließend werden grundsätzliche Überlegungen zur Architektur des Moduls erörtert und im weiteren Verlauf der innere Aufbau beschrieben.

4.1 Gewünschte Funktionen

Prinzipiell soll die PMU folgende Funktionen und Eigenschaften beinhalten:

- Diverse Zählregister, die konfigurierbar sind.
- Zähler können durch verschiedene Ereignisse getriggert werden:
  - Veränderung des Task Pointer (TP)
  - Veränderung des Program Counter (PC)
  - Interrupts
  - Externe Ereignisse
- Verschiedene Zählerarten:
  - Allgemeine Zähler
  - Taskabhängige Zähler, zählen für jeden Task
  - Taskzugehörige Zähler, zählen nur für einen bestimmten Task

- Skalierbar im Bezug auf die Anzahl
  - der Zählregister selbst und
  - der Konfigurationsregister pro Zählregister.
- Zugriff auf den Speicher, um Werte zwischenzuspeichern.
- Zählerstände beim Taskwechsel ohne Verzögerung im Speicher ablegen.

Die Systemuhr des V-scale Prozessors arbeitet mit einem 64 Bit Register in den CSRs, wie in [13] nachlesbar. Diese Register bieten verschiedene Systemfunktionalitäten an und können mit speziellen Instruktionen verwendet werden.

Die Zähler werden also auch als 64 Bit Register ausgelegt. Demnach können sie also von 0 bis $2^{64} - 1 = 1,845 \cdot 10^{19}$ inkrementieren, was bei dauerndem Zählen bei gewissen CPU-Taktraten den in Tabelle 4.1 angegebenen zeitlichen Wertebereichen entspricht.

CPU Takt	*Erreichdauer Maximalwert*
$1MHz$	$1,845 \cdot 10^{13}s \hat{=} 584942 \ \ Jahre$
$50MHz$	$3,689 \cdot 10^{11}s \hat{=} 11699 \ \ Jahre$
$100MHz$	$1,845 \cdot 10^{11}s \hat{=} 5849 \ \ Jahre$
$1GHz$	$1,845 \cdot 10^{10}s \hat{=} 584,9 \ \ Jahre$
$5GHz$	$3,689 \cdot 10^{9}s \hat{=} 117 \ \ Jahre$

Tabelle 4.1: Erreichdauer der Maximalwerte eines dauernd zählenden 64 Bit Zählers bei angegebener Taktrequenz.

Aufgrund dieser Werte lässt sich erkennen, dass selbst für einen Takt von 5 GHz der Maximalwert erst nach 117 Jahren erreicht wird, was einen ausreichenden Wertebereich darstellt, in dem nicht mit Überläufen zu rechnen ist.

Die Bitbreite der Konfigurationsregister sollte dem System angepasst eingeführt werden, sodass es hier zu keinen Sonderbehandlungen kommen muss. Beim V-scale sind das 32 Bit.

Damit die erstellte Hardware auch direkt auf einem Chip getestet werden kann, wird das in der Einleitung erwähnte Development-Board verwendet. Im weiteren Verlauf der Arbeit werden einige Besonderheiten dieser Plattform ausgenutzt und verwendet, um bessere Ergebnisse zu erzielen.

4.2 Architektur und Grundsatzüberlegungen

Es gibt verschiedene Möglichkeiten, die PMU für den V-scale Core zu positionieren. Sie kann am Peripheriebus angedockt werden, was den Vorteil einer einfachen Ansprechbarkeit von außen liefert und die Kommunikation gänzlich über den Bus abwickelt. Der Zugriff auf den Speicher wäre somit sehr einfach gehalten.

Nachteil ist jedoch, dass aufgrund der nicht vorherzusehenden Buszugriffe etwaiger anderer Geräte die PMU nicht mehr deterministisch arbeiten könnte, was zu Messungenauigkeiten führen würde. Demnach ist es besser, das Modul näher am Prozessorkern zu positionieren.

Nach Analyse des Verilog-Codes stellte sich heraus, dass sich hierfür das Pipeline-Modul am besten eignet, da hier viele weitere Module, wie in Abbildung 4.1 dargestellt, zusammengeschalten sind.

Bei den Modulen handelt es sich unter anderem um die Control Unit (ctrl), den PC Multiplexer (PC mux), das Registerfile (regfile), diverse Multiplexer und Generatoren, die arithmetisch-logische Einheit (alu), den Multiplizierer/Dividierer (mul div) und die CSRs.

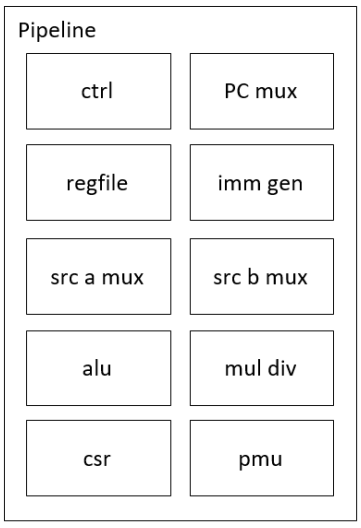


Abbildung 4.1: Die Darstellung der Module, die in der V-scale-Pipeline verbunden sind (ohne ihre Verbindungen).

Hier können alle wichtigen Statusinformationen des Prozessors abgefragt und verarbeitet werden, ohne interne Prozesse verzögern zu müssen.

Der Zugriff auf die Messwerte kann dann in weiterer Folge einfach über die CSRs erfolgen.

Der Speicherzugriff ist in dieser Variante etwas aufwändiger gestaltet, da er durch das Pipeline-Modul durchgeschleift werden muss. Hierbei muss natürlich gewährleistet werden, dass keine Speicherzugriffsverletzungen durch andere Module oder die Software geschehen. Aufgrund dieser Tatsache wird auf einen Dual-Port Random Access Memory (RAM) gesetzt. Dieser RAM hat also, wie der Name schon sagt, zwei Kanäle beziehungsweise Ports, durch die Speicherzugriffe stattfinden können. Der erste Port ist vom bestehenden System bereits für das Betriebssystem reserviert, somit kann der zweite Port des Speichers für CPU-Zwecke ausgelegt werden. Da es in dem hier entwickelten System kein weiteres Modul gibt, das Speicherzugriff benötigt, gibt es auch keine Bedenken im Bezug auf Zugriffsverletzungen oder Ähnlichem.
Die genaue Beschreibung dieses Speichers wird in Kapitel 4.6 geliefert.

Die nun festgelegte, noch sehr grobe Architektur wird in Abbildung 4.2 verdeutlicht.

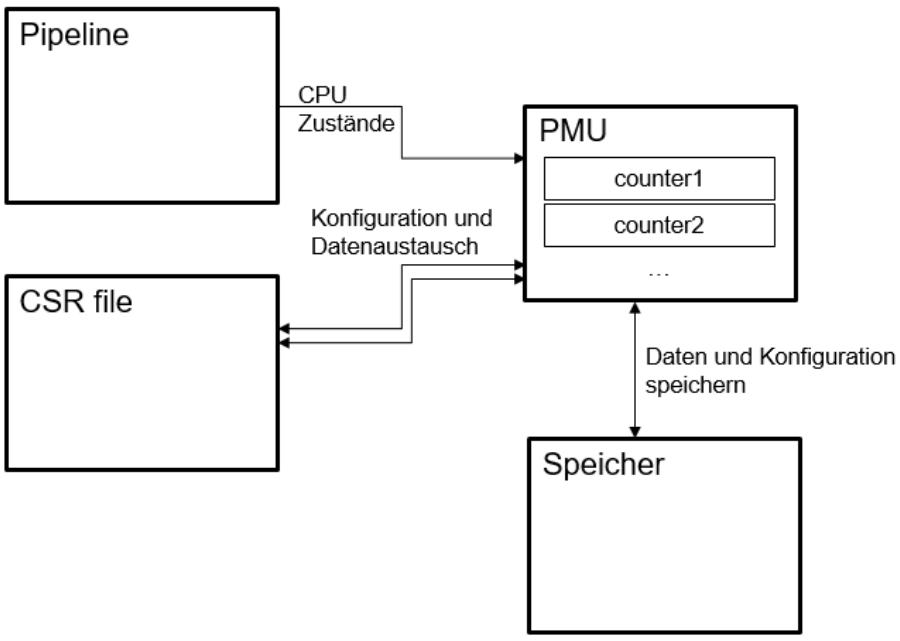


Abbildung 4.2: Die grobe Darstellung der PMU und ihrer verbundenen Module im System. Weitere Verbindungen, die nicht von der PMU ausgehen, werden hier nicht veranschaulicht.

Da nun Position und Architektur der zu entwickelnden PMU bestimmt sind, müssen im nächsten Schritt die Ein- und Ausgänge des Moduls festgelegt werden.

4.3 Ein- und Ausgänge

Die Verbindungspunkte bestehen, neben den obligatorischen Eingängen Reset und Prozessortakt aus:

- Zustandseingängen von der Pipeline:
  - `prv` → Privilege Mode, in welchem sich die CPU befindet
  - `pc` → Aktueller Program Counter (PC) der Pipeline
  - `tp` → Task Pointer (TP) des momentan aktiven Tasks im System
  - `mip` → die Interrupts, die aktuell vorliegen (Machine Interrupt Pending)
  - `ext` → Port zum Anlegen externer Signale
- Ein- und Ausgängen zum Lesen und Schreiben von CSR-Werten,
- Ein- und Ausgängen zur Kommunikation mit dem Speicher und einem
- Messausgang zur Messung und Validierung von Zeiten mit einem Messgerät (für Debugging-Zwecke).

4.4 Konfiguration der PMU

Um die PMU skalier- und erweiterbar zu halten gibt es neben dem eigentlichen Hardwarebeschreibungsfile `vscale_pmu.v` noch ein Verilog-Headerfile namens `vscale_pmu.vh`, in welchem die gesamte Konfiguration des Moduls zu finden ist.
Das File teilt sich in drei Sektionen:

- Definition der Zählerarten, wiederum aufgeteilt in:
  - allgemeine Zähler,
  - taskabhängige Zähler und
  - taskzugehörige Zähler;
- Registerkonfiguration und
- Speicher-Swap Konfiguration (siehe Kapitel 4.6).

Die Zählerarten, welche als 1-Byte-Wert definiert und in die drei verschiedenen Gruppen einteilbar sind, werden in weiterer Folge kurz besprochen, um in Kapitel 4.5.1 genauer vorgestellt zu werden. Um die Gruppen zu differenzieren, unterscheiden sie sich jeweils um die obersten zwei Bit. Die unteren sechs Bit bezeichnen dann die Zählerart selbst.
Anschließend wird noch die Registerkonfiguration erklärt. Die gesamte Registerbeschreibung ist in Anhang A zu finden.

4.4.1 Zählerarten

Im folgenden Abschnitt werden die unterschiedlichen Zählerarten mit ihren jeweiligen führenden Bits vorgestellt und kurz erläutert. Die mittlels einer Zahl identifizierbaren Zählerarten werden später dazu verwendet, um die Zählerart eines Hardwarezählers im Hauptkonfigurationsregister der PMU festzulegen. Die genaue Struktur des genannten Hauptkonfigurationsregisters ist in Kapitel 4.4.2 beschrieben. Soviel vorab: Jeder Zähler besitzt im PMU-Hauptkonfigurationsregister ein Byte, das seine Zählerart festsetzt. Die Bytes für die diversen unterschiedlichen Arten sind im Folgenden beschrieben.

Die **allgemeinen** Zähler sind global gültig und zählen Laufzeiten und Ereignisse unabhängig vom aktuellen Task. Ihre Konfiguration ist ebenfalls allgemeingültig. Bei diesen Zählerarten müssen keine Informationen am Speicher abgelegt werden.
Ein Beispiel hierfür ist ein Zähler, der wie eine Systemuhr immer zählt. Dieses Kapitel definiert unter anderem die Zählerart `PMU_ALL_COUNTER`.

Ein **taskabhängiger** Zähler wird einmal **zentral** konfiguriert und ist für alle Tasks gültig. Dieser Zähler zählt immer dann, wenn ein Task aktiv ist und gewisse weitere Zählbedingungen erfüllt sind. Die inkrementierten Werte sind jedoch von Task zu Task unterschiedlich. Zum Zeitpunkt eines Taskwechsels muss also der aktuelle Stand in den Taskkontrollblock des zu beendenden Tasks im RAM geschrieben und der alte Zählerstand des zu startenden Tasks in das Zählregister geladen werden. Demnach befindet sich im Zählregister dann immer nur der Wert, der für den aktuell laufenden Task Gültigkeit besitzt.
Hier kann als Beispiel ein Zähler angegeben werden, welcher die Tasklaufzeiten für jeden einzelnen Task zählt. Konfiguriert muss dieser nur einmal werden, da er sich für alle Tasks des Systems gleich verhält, jedoch unterschiedliche Werte, je nach Laufzeit, ermittelt. Ein solcher Zähler ist mit `PMU_TASK_OVERALL_COUNTER` definiert.

Ein **taskzugehöriger** Zähler ist ein Zähler, dessen Werte und Konfiguration von Task zu Task unterschiedlich sind. Dieser Zähler muss also **individuell** in jedem Task einmal konfiguriert werden und es müssen sowohl Zählerstände als auch Konfigurationswerte in den RAM gesichert sowie von diesem geladen werden.
Als Beispiel dient der `PMU_TASK_PART_COUNTER`, dessen Konfiguration von Task zu Task verschieden ist, da er nur dann zählt, wenn sich der PC in einem gewissen Bereich befindet.

Allgemeine Zähler müssen dem folgenden Byteformat entsprechen:

7	6	5					0
0	0	X	X	X	X	X	X

Die definierten **allgemeinen** Zähler sind in Listing 4.1 zu sehen.

```
1 `define PMU_ALL_COUNTER                  8'h01
2 `define PMU_USERMODE_OVERALL_COUNTER     8'h02
3 `define PMU_SINGLE_TASK_COUNTER          8'h03
4 `define PMU_TASKTIME_COUNTER             8'h04
5 `define PMU_TASKTIME_COUNTER_NO_IDLE     8'h05
6 `define PMU_INTERRUPT_OVERALL_COUNTER    8'h06
7 `define PMU_INTERRUPT_MISSED             8'h07
8 `define PMU_EXT_MATCH_OVERALL            8'h08
```

Listing 4.1: Implementierte allgemeine Zähler.

Die führenden zwei Nullbits können anhand der Hexadezimaldarstellung mit der führenden Null (was einem Bitformat von $0000XXXX_b$ entspricht) erkannt werden.

Die **taskabhängigen** Zähler entsprechen folgendem Byteformat:

7	6	5					0
1	0	X	X	X	X	X	X

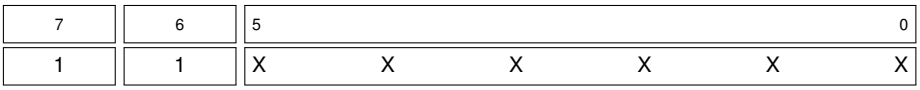
Implementiert sind die in Listing 4.2 angegebenen Zählerarten.

```
1 `define PMU_TASK_OVERALL_COUNTER         8'h81
2 `define PMU_TASK_INTERRUPT_COUNTER       8'h82
3 `define PMU_TASK_EXT_MATCH               8'h83
```

Listing 4.2: Implementierte taskzugehörige Zähler.

Das Bitmuster eines mit hexadezimal 8 beginnenden Bytewerts ist dabei immer $1000XXXX_b$, womit sich die führenden zwei Bits mit 10 manifestieren. Taskzugehörige Zähler sind für alle Tasks gültig, werden also in allen Tasks inkrementiert und zum jeweiligen Taskkontrollblock gespeichert. Hierbei ist wichtig, dass nur die Zählerstände, jedoch nicht die Konfiguration beim Taskwechsel zwischengespeichert werden müssen.

Abschließend noch das Format für einen **taskzugehörigen** Zähler:



Dabei gibt es den in Listing 4.3 angegebenen Zähler.

```
1 `define PMU_TASK_PART_COUNTER          8'hC2
```

Listing 4.3: Implementierte taskabhängige Zähler.

Mit einem führenden hexadezimalen C hat ein Bytewert immer die Form $1100XXXX_b$, wobei man wieder die führenden zwei Bits 11 erkennen kann. Der Unterschied zu einem taskzugehörigen Zähler ist, dass hierbei sowohl die Zählerstände, als auch die Konfiguration bei Taskänderung zwischengespeichert werden müssen, da jeder Task eine andere Konfiguration dieses Zählers haben kann.

Im Allgemeinen werden alle Zählerstände und Konfigurationsregister bei jedem Taskwechsel in den Taskkontrollblock gesichert, um die Werte für das Betriebssystem zur Verfügung zu stellen. Wie genau die Speicherung beziehungsweise das Laden der Werte erfolgt, wird in Kapitel 4.6 beschrieben.

4.4.2 Konfiguration und Register

Prinzipiell werden für jeden definierten Zähler in den Hauptkonfigurationsregistern ein Byte für die Zählerart reserviert. Dabei ist zu beachten, dass ein CSR nicht breiter als 32 Bit sein kann, wodurch das Hauptkonfigurationsregister zu mehreren Registern entarten kann. Zusätzlich dazu werden noch die definierte Anzahl an 32-Bit-Konfigurationsregister erstellt, die dem Zähler zugehörig sind. Das Hauptkonfigurationsregister beinhaltet die Art des Zählers. Die zugehörigen Konfigurationsregister werden zur Konfiguration speziell dieser Zählerart benötigt und im Kapitel 4.5 genauer erläutert.

Um nun PMUs mit verschiedener Anzahl an Zähl- und Konfigurationsregistern zu erstellen, gibt es in dem Headerfile zwei Einstellmöglichkeiten, wie in Listing 4.4 angegeben. Hier wird die Konfiguration mit vier Zählregistern und zwei Konfigurationsregistern pro Zählregister illustriert.

```
1 `define CSR_PMU_N     12'h4
2 `define CSR_PMU_CFG_N 12'h2
```

Listing 4.4: Konfiguration der Anzahl an Registern.

Dieser Sachverhalt ist in Abbildung 4.3 dargestellt. Man sieht auch farblich, welcher Wert des Hauptkonfigurationsregisters welchem Konfigurationsregister zugeordnet ist. Für diese Ansicht wurde eine Darstellung mit zwei Konfigurationsregistern pro Zähler gewählt. Links sieht man noch den Offset der einzelnen Register, wie sie in weiterer Folge auch zu adressieren sind.

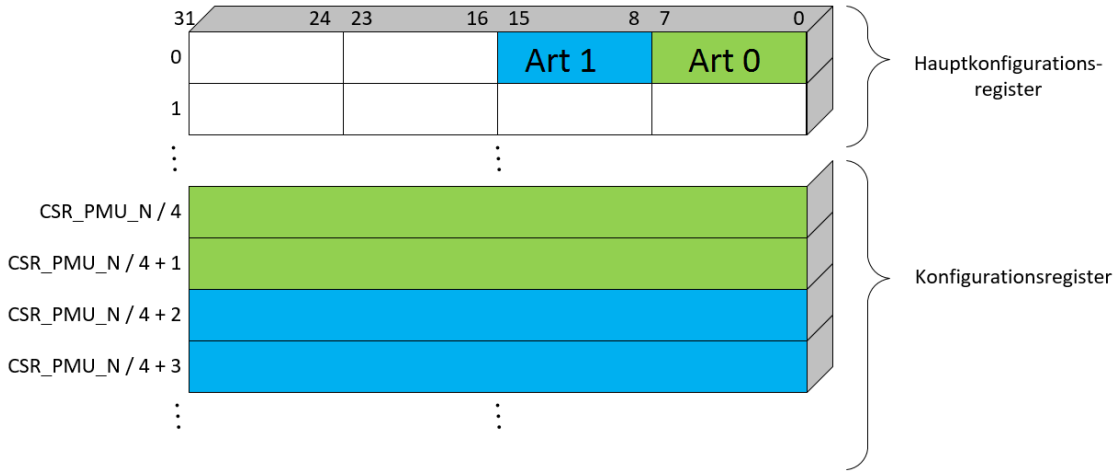


Abbildung 4.3: Die aufgrund der Einstellungen generierten Konfigurationsregister. Eine Farbe verdeutlicht dabei die Zugehörigkeit zu einem Hardwarezähler.

Aus Gründen der Skalierbarkeit wurde die gesamte Konfiguration, wie abgebildet,

in einem Array aus 32-Bit-Werten in der Hardware erstellt, sodass sie auch in der später erklärten CSR-Adressierung hintereinander liegen.

Die Logik der Hardware erstellt nun aufgrund der vorher erläuterten Einstellungen eine PMU mit vier 64-Bit-Zählregistern, einem Hauptkonfigurationsregister mit vier mal einem Byte sowie acht Konfigurationsregister in Integerlänge, was auf unserer CPU 32 Bit entspricht. Diese Register werden als CSRs gemapped, um der Softwareschicht zur Verfügung zu stehen. Hierbei sind diese natürlich wieder nur als 32-Bit-Werte schreib- und lesbar.

Die Zugriffe auf die CSRs werden über Adressbereiche geregelt. Die Startadressen für die genannten Werte können ebenfalls in diesem Headerfile eingestellt werden, wie in Listing 4.5 gezeigt.

```
1 `define CSR_ADDR_PMU_CFG_START 12'h8F0
2 `define CSR_ADDR_PMU_START     12'hC03
3 `define CSR_ADDR_PMU_STARTH    12'hC83
4 `define CSR_ADDR_PMU_STARTW    12'h903
5 `define CSR_ADDR_PMU_STARTHW   12'h983
```

Listing 4.5: Startadressen der Register.

Die hier verwendeten Adressen wurden so gewählt, dass sich die Konfigurationsregister `CSR_ADDR_PMU_CFG_START` im lese- und schreibberechtigten Bereich der CSRs des RISC-V befinden. Waterman et al. definieren die CSR-Adressbereiche in [3] dabei wie in den Tabellen 4.2 und 4.3 dargestellt.

CSR Address			*Hex*	*Use and Accesibility*
[11:0]	[9:8]	[7:6]		
User CSRs				
00	00	XX	0x000-0x0FF	Standard read/write
01	00	XX	0x400-0x4FF	Standard read/write
10	00	XX	0x800-0x8FF	Non-standard read/write
11	00	00-10	0xC00-0xCBF	Standard read-only
11	00	11	0xCC0-0xCFF	Non-standard read-only

Tabelle 4.2: Verteilung der RISC-V-CSR-Adressen im User-Modus aus [3].

CSR Address			*Hex*	*Use and Accesibility*
[11:0]	[9:8]	[7:6]		
Supervisor CSRs				
00	01	XX	0x100-0x1FF	Standard read/write
01	01	00-10	0x500-0x5BF	Standard read/write
01	01	11	0x5C0-0x5FF	Non-standard read/write
10	01	00-10	0x900-0x9BF	Standard read/write shadows
10	01	11	0x9C0-0x9FF	Non-standard read/write shadows
11	01	00-10	0xD00-0xDBF	Standard read-only
11	01	11	0xDC0-0xDFF	Non-standard read-only

Tabelle 4.3: Verteilung der RISC-V-CSR-Adressen im Supervisor-Modus aus [3].

Neben diesen Adressen gibt es noch weitere, die Hypervisor- und Machine-CSRs adressieren, jedoch sind diese für unsere Zwecke irrelevant, da sie nicht zum Einsatz kommen. Im vorliegenden Betriebssystem wie auch in der V-scale-Implementierung werden nur User- und Machine-Mode verwendet.

Die Startadresse für die Konfigurationsregister wurde so gewählt, dass sie sich unter den User-CSRs im Bereich *Non-standard read/write* befindet, um auch im Usermode der CPU auf die Konfiguration lesend und schreibend zugreifen zu können. Dabei stehen laut [3] und Tabelle 4.2 die Adressen 0x8F0-0x8FF, also Adressen für 16 Werte zur Verfügung. Braucht man in einer Konfiguration mehr Adressen, muss die Startadresse im Wertebereich 0x800-0x8FF angepasst werden.

Die Startadresse für die Zählwerte wurde so platziert, dass die der RISC-V-Definition für Hardware Performance Counter entsprechen. Die Spezifikation [3] definiert dabei eigene Register zum Starten der Zählvorgänge, was in dieser Arbeit anderweitig gelöst wurde.
Um nun einen Zählwert zu adressieren, muss man sowohl die Adresse für den High- als auch den Low-Wert kennen. Diese Adressen starten bei 0xC03 für den Low-Wert und 0xC83 für den High-Wert. Zusätzlich dazu gibt es noch die sogenannten Shadow-Adressen im Supervisor Mode, die bei 0x903 respektive 0x983 liegen und den Zugriff aus dem Supvervisor Level auf das gleiche Register ermöglichen.

Um die Endadresse für die Konfigurationsregister zu berechnen, muss wie folgt vorgegangen werden:

$$AddrCFG_{end} = AddrCFG_{start} + N_{cnt} \cdot N_{cfg} + \left\lceil \frac{N_{cnt}}{4} \right\rceil - 1 \tag{4.1}$$

Dabei ist $AddrCFG_{start}$ die vorher definierte Startadresse, N_{cnt} die Anzahl an Zählregistern und N_{cfg} die Anzahl an Konfigurationsregistern pro Zählregister. Die Summe stellt die Anzahl an gesamten Konfigurationsregistern und dem Hauptkonfigurationsregister, welches für jeden Zähler ein Byte beinhaltet, dar. Sie muss um eins dekrementiert werden, da der Adressbereich beim Offset 0 beginnt.

Es ist zu beachten, dass alle Register 32 Bit lang sind. Demnach ergibt sich das Hauptkonfigurationsregister nicht zwangsläufig nur aus einem 32 Bit Register, sondern kann aus mehreren – genauer gesagt aus $\lceil \frac{N_{cnt}}{4} \rceil$ Registern bestehen. Für vier Zählregister braucht man $4 \cdot 8Bit = 32Bit \hat{=}$ 1 Register, für sechs Zählregister bereits $6 \cdot 8Bit = 48Bit \hat{=}$ 2 Register.
Die gesamte Anzahl an 32-Bit-Konfigurationsregistern $N_{cfg,reg}$ ergibt sich durch die Berechnung von

$$N_{cfg,reg} = N_{cnt} \cdot N_{cfg} + \left\lceil \frac{N_{cnt}}{4} \right\rceil \tag{4.2}$$

und wird in weiterer Folge des öfteren benötigt.

Die Endadressen $AddrCNT_{n,end}$ für den Zugriff auf die Werte der Zählerstände werden allesamt in gleicher Manier nach der Vorschrift

$$AddrCNT_{n,end} = AddrCNT_{n,start} + N_{cnt} - 1 \tag{4.3}$$

berechnet, wobei $AddrCNT_{n,start}$ für die jeweilige definierte Startadresse steht.

4.4.3 Lese- und Schreibzugriffe auf die Register

Da nun die Adressen und Konfigurationen feststehen, kann sich um die Zugriffe von außen gekümmert werden. Diese werden über das CSR-File der CPU abgehandelt, damit die Schnittstelle zur Software nicht verändert werden muss. Liegt nun ein Zugriff auf eines der PMU-Register vor, wird dieser an das PMU-Modul weitergegeben und von diesem bearbeitet.

Die zugehörige Softwareschnittstelle wird in Kapitel 5 betrachtet, hier wird nur auf die Hardwareseite eingegangen.

Je nachdem in welchem CSR-Adressbereich sich das jeweilige Register nun befindet, erlaubt die CPU Lese- und/oder Schreibzugriff in dem zugehörigen CPU-Modus. Um der CPU mitzuteilen, welche Adressen von der entwickelten PMU verwendet werden, müssen diese noch im CSR-File (bei V-scale in `vscale_csr_file.v`) in der in Listing 4.6 angegebenen Manier definiert werden.

```
1  always @(*) begin
2  case (addr)
3   [...]
4   default : begin
5      if ( (addr >= `CSR_ADDR_PMU_START &&
6               addr < `CSR_ADDR_PMU_START + `CSR_PMU_N) ||
7            (addr >= `CSR_ADDR_PMU_STARTH &&
8               addr < `CSR_ADDR_PMU_STARTH + `CSR_PMU_N) ||
9            (addr >= `CSR_ADDR_PMU_STARTW &&
10              addr < `CSR_ADDR_PMU_STARTW + `CSR_PMU_N) ||
11           (addr >= `CSR_ADDR_PMU_STARTHW &&
12              addr < `CSR_ADDR_PMU_STARTHW + `CSR_PMU_N) ||
13           (addr >= `CSR_ADDR_PMU_CFG_START &&
14              addr < `CSR_ADDR_PMU_CFG_START +
15                   (`CSR_PMU_N * `CSR_PMU_CFG_N + `CSR_PMU_N/4)) )
16     begin
17          rdata = pmu_csr; defined = 1'b1;
18     end else begin
19          rdata = 0; defined = 1'b0;
20     end
21   end
22 endcase
23 end
```

Listing 4.6: Definition des Adressbereichs für CSRs.

Die Leitung `addr` steht dabei für die Adresse des CSR, auf das zugegriffen werden soll. Hier sieht man in Zeile 17 auch, wie die Weiterleitung vom CSR-File an das PMU-Modul im Lesefall funktioniert. Dabei ist `pmu_csr` die Ausgangsleitung für Lesevorgänge von der PMU und `rdata` die Lesedaten für CSR-Zugriffe.

Die Leitung `defined` wird demnach – zusätzlich zu den bereits bestehenden CSR-Adressen – high-aktiv, wenn sich die angelegte Adresse innerhalb der gültigen Adresskonfiguration des Moduls befindet und kennzeichnet einen gültigen Registerzugriff.

Lesezugriff

Auf Seiten des Zählmoduls sieht diese Leseverbindung dann so aus, dass in den jeweiligen Adressbereichen entweder der korrespondierende Zählwert, oder aber der Wert des adressierten Konfigurationsregisters an die Leitung `pmu_csr` angelegt wird. Realisiert wird dies in Form des in Abbildung 4.4 dargestellten Multiplexers.

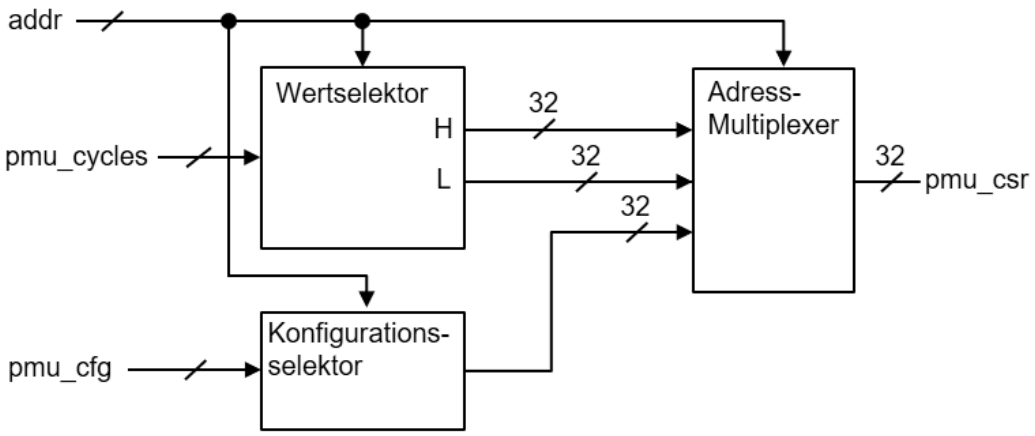


Abbildung 4.4: Der Multiplexer, der beim Auslesen eines Registers die Wertauswahl trifft, mit seinen Verbindungen und externen Beschaltung.

Die Eingangsbitbreite kann dabei je nach Konfiguration variieren, die Ausgangsbitbreite der Leitung `pmu_csr` ist jedoch aufgrund der Architektur mit 32 Bit festgelegt. Die Leitung `pmu_cycles` kennzeichnet die aktuellen Zählerstände, `pmu_cfg` die Konfigurationswerte. In `addr` kann die Adresse des Zugriffs gefunden werden.

Schreibzugriff

Die Schreibzugriffe auf die einzelnen Register – vorwiegend auf die Konfigurationsregister, da sich die Zählregister nach momentaner Konfiguration im exklusiven Lesezugriffadressbereich befinden – muss, wie üblich, taktgesteuert ablaufen. Es werden also Werte an die Speicherzelle angelegt, welche bei einer steigenden Taktflanke des CPU-Takts in diese gespeichert werden.

In diesem Anwendungsfall gibt es zwei Teilnehmer, die auf die gleichen Werte schreibend zugreifen können: Einerseits die Software, welche über Befehle Daten in CSRs schreibt, andererseits die PMU selbst, die beim Taskwechsel selbstständig Daten aus dem Speicher in die Register lädt (siehe Kapitel 4.6). Hier muss gewährleistet werden, dass sich diese beiden Teilnehmer nicht gegenseitig behindern können und so ungültige Werte in den Registern landen.

Der gewählte Ansatz sieht hier einen Zustandsautomaten vor, der nur im Leerlauffall Schreibzugriffe von der Software zulässt. Benötigt und beschrieben wird dieser Automat zu einem späterem Zeitpunkt. Es sei jedoch darauf hingewiesen, dass hier genaue Status festgelegt sind, wann welche Seite schreibberechtigt ist. Somit sind Ambiguitäten ausgeschlossen.

Die Logik der Schreibzugriffe folgt wieder der von CSR-Schreibzugriffen. Das CSR-File, in dem die Zugriffe auf die CSRs geregelt ist, leitet diese Zugriffe in Form der Schreibzugriffsleitung `pmu_wen`, der Adressleitung `pmu_addr` sowie der 32-Bit-Datenleitung `pmu_d` an das PMU-Modul weiter. Dieses errechnet sich dann aufgrund der anliegenden Adresse, in welches Register geschrieben werden soll. Dabei wird auch die prinzipielle Möglichkeit des Schreibens in ein Zählregister implementiert. Die Verschaltung ist in Abbildung 4.5 dargestellt.

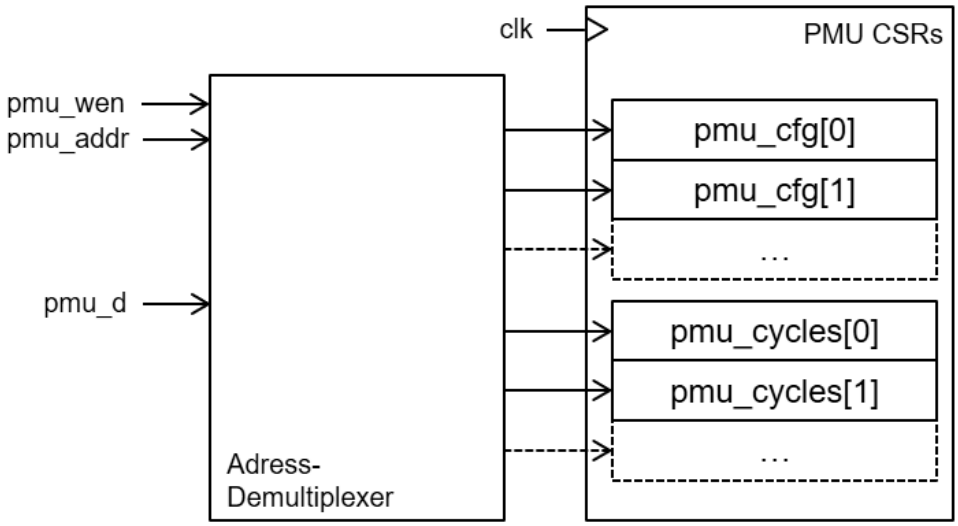


Abbildung 4.5: Der Demultiplexer, der beim Schreiben eines Registers zum Tragen kommt, mit seinen Verbindungen und externen Beschaltung.

Der Demultiplexer ist so gestaltet, dass er sich je nach Konfiguration der Registerzahlen mitskaliert. Es wird also gewährleistet, dass immer die gewünschten Register angesprochen werden, egal wie viele Zähler oder Konfigurationsregister je Zähler vorhanden sind.
Auf Verilog-Code wird an dieser Stelle verzichtet, da die Adressierung bereits bei den Lesezugriffen abgehandelt wurde.

4.5 Kombinatorische Logik der Zähler

Da das Setzen von Konfigurationswerten für die Zähler, sowie das Auslesen der aktuellen Zählwerte abgehandelt wurde, kann nun erarbeitet werden, wann ein Zähler wirklich zählt und wann nicht. Eine Zähleinheit besteht dabei aus einem Konfigurationsbyte im Hauptkonfigurationsregister, diversen Konfigurationsregistern und einem 64-Bit-Zählregister.
Um nun einen Zähler inkrementieren zu lassen, muss eine kombinatorische Logik entwickelt werden, welche alle unterschiedlichen Zählerarten abwickelt. Das Prinzipschaltbild hierzu ist in Abbildung 4.6 dargestellt.

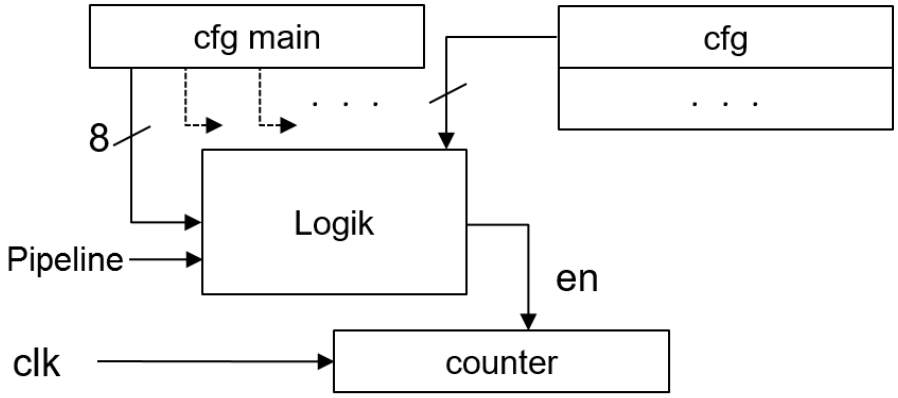


Abbildung 4.6: Eine vereinfachte Darstellung der kombinatorischen Logik, um einen Zähler zu aktivieren.

Das Zählregister – in der obigen Abbildung mit `counter` bezeichnet – wird also genau dann um eins erhöht, wenn die Enable-Leitung `en` high-aktiv ist und eine steigende Flanke des Prozessortakts `clk` auftritt. Ermittelt wird diese Leitung aus den beiden angegebenen Quellen. Jeder Zähler hat also seine eigene kombinatorische Logik, welche das Enable-Flag erstellt. Kombinatorische Logiken beinhalten im Gegensatz zu sequentiellen Logiken keine Speicherelemente und sind demnach auch nicht taktgetriggert.

Dem Logikblock stehen zusätzlich zu den gezeigten Eingängen noch alle Zustandsinformationen der CPU, also Dateneingänge seitens der Pipeline, zur Verfügung. Mit diesen Informationen und der Konfiguration werden anschließend die Zählzeitpunkte ermittelt.

Um hier skalierbar zu bleiben, wird das Verilog-Statement `generate` verwendet. Dies gewährleistet, dass immer die gleiche Anzahl der oben dargestellten Logiken in der Hardware zur Verfügung stehen, wie es Zählregister gibt. Einen Ausschnitt aus dem korrespondierenden Code ist in Listing 4.7 zu sehen.

```
1  genvar i;   // Generierungsvariable
2  generate
3  for (i = 0; i < `CSR_PMU_N; i = i + 1) begin
4    assign cfg[i] = pmu_cfg[i/4][`XPR_LEN-1 - (i\%4) * (`XPR_LEN/4)
5              : `XPR_LEN - (`XPR_LEN/4) - (i\%4) * (`XPR_LEN/4)];
6    assign cnt_en[i] =
7     ((cfg[i] == `PMU_ALL_COUNTER && 1) ||
8      (cfg[i] == `PMU_USERMODE_OVERALL_COUNTER && prv == `PRV_U) ||
9      [...] );     // weitere Zählerarten
10 end
11 endgenerate
```

Listing 4.7: Kombinatorische Logik eines Zählers.

Der Wert `CSR_PMU_N` entspricht dabei der Anzahl der Zählregister, `XPR_LEN` der Integer-Länge am System, die hier 32 Bit beträgt. Die Leitungen `cfg[i]` stellen demnach das zugehörige Byte aus dem Hauptkonfigurationsregister und `cnt_en[i]` die Enable-Leitung des Zählers mit dem Index `i` dar.

Die eigentliche Kombinatorik beginnt mit Zeile 6, wo dann für jede Zählerart eine eigene Logik implementiert werden muss. Illustriert wird dies hier mit dem vorher vorgestellten `PMU_ALL_COUNTER`, der immer zählt. Im Gegensatz dazu zählt der gezeigte `PMU_USERMODE_OVERALL_COUNTER` nur dann, wenn die Eingangsleitung `prv` der PMU den Wert `PRV_U` hat, sich die CPU also im User-Modus befindet. Eine genauere Beschreibung zur Implementierung findet sich in Kapitel 4.5.1.

Um nun einen Zähler zu Inkrementieren, ohne mit den etwaigen Schreibprozessen von User- oder Hardwareseite zu kollidieren, wird wie in Listing 4.8 gezeigt, vorgegangen. Die Logik skaliert hier abermals mit den eingestellten Werten.

```
1  integer j;
2  always @(posedge clk) begin
3      [...]   // Reset
4      for (j = 0; j < `CSR_PMU_N; j = j + 1) begin
5          if (cnt_en[j] && ((!osmem_busy && !osmem_wait)
6                                  || cfg[j][(`XPR_LEN/4)-1] == 0))
7              pmu_cycles[j] <= pmu_cycles[j] + 1;
8          else if(cfg[j] == 0)
9              pmu_cycles[j] <= 0;
10     end
11     [...]   // Zustandsautomat, siehe später
12 end
```

Listing 4.8: Implementierung von mehreren kollisionsresistenten Hardwarezählern.

Erkennbar ist dabei, dass der Zähler mit dem Index `j` in Zeile 8 nur dann inkrementiert wird, wenn seine Enable-Leitung high-aktiv ist und außerdem keine Speicherzugriffe stattfinden. Ist der Zähler so konfiguriert, dass Speicherzugriffe sowieso keine Änderung des Zählwerts hervorrufen können (oberstes Bit der Zählerart 0, allgemeiner Zähler), werden selbst die Speicherzugriffe ignoriert und trotzdem weitergezählt. Die Leitungen `osmem_busy` und `osmem_wait` sind Leitungen, die high-aktiv werden, sobald der RAM arbeitet.
Ein weiteres Feature ist in Zeile 10 zu sehen: Setzt man die Zählerart eines Zählers auf 0, wird der Zähler ebenfalls auf 0 gesetzt.

4.5.1 Zählerarten und deren individuelle Logik

Momentan sind 12 Zähler, wie in Kapitel 4.4.1 angegeben, implementiert, die in diesem Kapitel näher beleuchtet werden. Außerdem wird noch erläutert, wie ein neuer Zähler hinzuzufügen ist.

Eine Zählerart manifestiert sich einerseits als Definition des individuellen Bytes in `vscale_pmu.vh`, welche den bereits erklärten Konventionen bezüglich Gruppeneinteilung folgen müssen, und andererseits als eigene Einheit der kombinatorischen Logik jedes Zählers in `vscale_pmu.v`. Die Zusammenschaltung mehrerer solcher Einheiten ist in Abbildung 4.7 illustriert und kann beliebig erweitert werden.

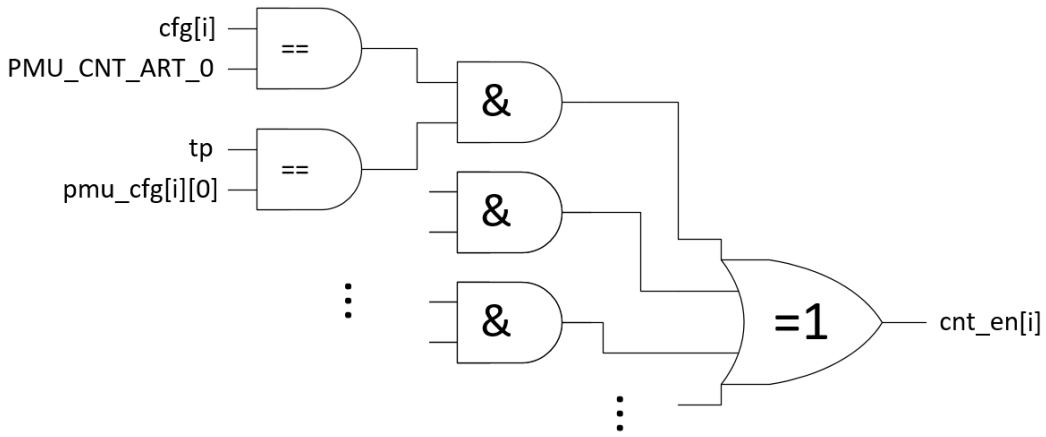


Abbildung 4.7: Schemenhafte Darstellung der Logik bestehend aus kombinatorischen Einheiten, um einen Zähler zu aktivieren.

Das gezeigte Beispiel ist so zu verstehen: Ist der Zähler mit dem Index `i` auf die Zählerart `PMU_CNT_ART_0` konfiguriert und gleicht der Inhalt des zugehörigen Konfigurationsregisters mit dem Index 0 `pmu_cfg[i][0]` dem Taskpointer `tp`, so generiere ein high-aktives Signal `cnt_en[i]` und inkrementiere damit in

weiterer Folge das Zählregister. Für jede Zählerart muss dem Oder-Gatter am Ende ein neuer Eingang hinzugefügt werden, der die Zählerart implementiert. Diese Implementierungen sind in Listing 4.7 in den Zeilen 7 bis 9 zu sehen.

PMU_ALL_COUNTER (0x01)

Nun jedoch zur Beschreibung der einzelnen individuellen Logiken, beginnend mit dem `PMU_ALL_COUNTER (0x01)` in Listing 4.9.

```
1 (cfg[i] == `PMU_ALL_COUNTER)
```

Listing 4.9: Implementierung des `PMU_ALL_COUNTER`.

Dieser Zähler zählt demnach jeden Taktzyklus und kann als Systemuhr verwendet werden.

PMU_USERMODE_OVERALL_COUNTER (0x02)

`PMU_USERMODE_OVERALL_COUNTER (0x02)` ist in Listing 4.10 zu sehen.

```
1 (cfg[i] == `PMU_USERMODE_OVERALL_COUNTER && prv == `PRV_U)
```

Listing 4.10: Implementierung des `PMU_USERMODE_OVERALL_COUNTER`.

Hier wird nur dann gezählt, wenn sich die CPU im User Privilege Mode befindet. Dies wird dadurch abgefragt, dass die `prv`-Leitung den Wert `PRV_U` hat, was laut [3] dem Binärwert 00 entspricht.

PMU_SINGLE_TASK_COUNTER (0x03)

Der `PMU_SINGLE_TASK_COUNTER (0x03)` ist in Listing 4.11 ausgeführt.

```
1 (cfg[i] == `PMU_SINGLE_TASK_COUNTER
2             && tp == pmu_cfg[(`CSR_PMU_N/4) + i*`CSR_PMU_CFG_N])
```

Listing 4.11: Implementierung des `PMU_SINGLE_TASK_COUNTER`.

Gezählt wird hierbei nur, wenn der aktuelle Taskpointer dem im ersten Konfigurationsregister des Zählers gesetzten Wert entspricht, es wird also ein spezieller Task gemessen.

PMU_TASKTIME_COUNTER (0x04)

In Listing 4.12 ist der `PMU_TASKTIME_COUNTER (0x04)` definiert.

```
1 (cfg[i] == `PMU_TASKTIME_COUNTER && tp != 0)
```

Listing 4.12: Implementierung des `PMU_TASKTIME_COUNTER`.

Dieser zählt nur dann, wenn der Taskpointer einen Wert ungleich 0 hat, sprich ein gültiger Taskpointer gesetzt wurde.

PMU_TASKTIME_COUNTER_NO_IDLE (0x05)

Nun zum `PMU_TASKTIME_COUNTER_NO_IDLE (0x05)` in Listing 4.13.

```
1 (cfg[i] == `PMU_TASKTIME_COUNTER_NO_IDLE
2     && tp != pmu_cfg[(`CSR_PMU_N/4) + i * `CSR_PMU_CFG_N]
3     && tp != 0)
```

Listing 4.13: Implementierung des `PMU_TASKTIME_COUNTER_NO_IDLE`.

Dieser Zähler zählt genau dann, wenn der Taskpointer weder dem Wert im Konfigurationsregister, noch 0 entspricht. In diesem Konfigurationsregister soll der Wert des Pointers auf den Idle-Task eingetragen werden, um die Laufzeit aller Tasks ohne dem Idle-Task zu messen.
Diese Funktion ist vor allem für die Ermittlung der Schedulability [25] von Bedeutung.

PMU_INTERRUPT_OVERALL_COUNTER (0x06)

Der `PMU_INTERRUPT_OVERALL_COUNTER (0x06)` ist in Listing 4.14 zu sehen.

```
1 (cfg[i] == `PMU_INTERRUPT_OVERALL_COUNTER
2     && mip_changed == 1
3     && |(mip & pmu_cfg[(`CSR_PMU_N/4) + i*`CSR_PMU_CFG_N]) == 1)
```

Listing 4.14: Implementierung des `PMU_INTERRUPT_OVERALL_COUNTER`.

Getriggert wird dieser Zähler nur, falls sich die Interrupt-Pending-Leitung `mip` ändert und der neue Wert auch in der Bitmaske des Konfigurationsregisters vorhanden ist. Dabei werden die beiden Werte binär und-verknüpft und die Ergebnisbits zusammen binär oder-verknüpft.
Die Leitung `mip_changed` wird durch Zwischenspeicherung in ein internes Register erzeugt.

PMU_INTERRUPT_MISSED (0x07)

Den PMU_INTERRUPT_MISSED (0x07)-Zähler sieht man in Listing 4.15.

```
1 (cfg[i] == `PMU_INTERRUPT_MISSED
2     && mip_changed == 1
3     && |(mip & pmu_cfg[(`CSR_PMU_N/4) + i * `CSR_PMU_CFG_N]) == 1
4     && tp != pmu_cfg[(`CSR_PMU_N/4) + i * `CSR_PMU_CFG_N + 1])
```

Listing 4.15: Implementierung des PMU_INTERRUPT_MISSED.

Hier werden nur die maskierten Interrupts gezählt, die auftreten, während der im zweiten Konfigurationsregister definierte Task nicht aktiv ist.

PMU_EXT_MATCH_OVERALL (0x08)

PMU_EXT_MATCH_OVERALL (0x08) ist in Listing 4.16 ausgeführt.

```
1 (cfg[i] == `PMU_EXT_MATCH_OVERALL
2     && ext_changed == 1
3     && ext == pmu_cfg[(`CSR_PMU_N/4) + i * `CSR_PMU_CFG_N])
```

Listing 4.16: Implementierung des PMU_EXT_MATCH_OVERALL.

Dieser Zähler wird hochgezählt, wenn sich der externe Eingangsport ändert – wie zuvor bei den Interrupts – und dieser dem Konfigurationsregister entspricht. So kann auf externe Ereignisse reagiert werden.

PMU_TASK_OVERALL_COUNTER (0x81)

Der erste Zähler in der Riege der taskabhängigen Zähler, der mit dem Wert PMU_TASK_OVERALL_COUNTER (0x81) definiert ist, ist in Listing 4.17 zu sehen.

```
1 (cfg[i] == `PMU_TASK_OVERALL_COUNTER && tp != 0)
```

Listing 4.17: Implementierung des PMU_TASK_OVERALL_COUNTER.

Dieser Zähler zählt also immer, außer der TP hat den Wert null. Zählstände von taskabhängigen Zählern werden jedoch, wie bereits erwähnt, zum Task zugehörig im Speicher abgelegt. Die Abfrage, ob der TP nicht Null ist, ist dazu notwendig, um beim Ablegen in den Speicher keine ungültigen Speicherstellen um Null herum zu beschreiben.

PMU_TASK_INTERRUPT_COUNTER (0x82)

Der `PMU_TASK_INTERRUPT_COUNTER (0x82)` folgt in Listing 4.18.

```
1 (cfg[i] == `PMU_TASK_INTERRUPT_COUNTER
2     && mip_changed == 1
3     && |(mip & pmu_cfg[(`CSR_PMU_N/4) + i*`CSR_PMU_CFG_N]) == 1)
```

Listing 4.18: Implementierung des `PMU_TASK_INTERRUPT_COUNTER`.

Dieser entspricht dem `PMU_INTERRUPT_OVERALL_COUNTER`, jedoch in taskabhängiger Variante.

PMU_TASK_EXT_MATCH (0x83)

Gleiches gilt für den `PMU_TASK_EXT_MATCH (0x83)` in Listing 4.19.

```
1 (cfg[i] == `PMU_TASK_EXT_MATCH
2     && ext_changed == 1
3     && ext == pmu_cfg[(`CSR_PMU_N/4) + i * `CSR_PMU_CFG_N])
```

Listing 4.19: Implementierung des `PMU_TASK_EXT_MATCH`.

PMU_TASK_PART_COUNTER (0xC2)

Ein Vertreter der taskzugehörigen Zähler ist der `PMU_TASK_PART_COUNTER (0xC2)` in Listing 4.20.

```
1 (cfg[i] == `PMU_TASK_PART_COUNTER
2     && (pmu_cfg[(`CSR_PMU_N/4) + i * `CSR_PMU_CFG_N][0]
3       && !pmu_cfg[(`CSR_PMU_N/4) + i * `CSR_PMU_CFG_N + 1][0]
4       ||(pc == pmu_cfg[(`CSR_PMU_N/4) + i * `CSR_PMU_CFG_N]
5         ||pc == pmu_cfg[(`CSR_PMU_N/4) + i * `CSR_PMU_CFG_N + 1]
6     && tp != 0)))
```

Listing 4.20: Implementierung des `PMU_TASK_PART_COUNTER`.

Dieser kann für jeden Task individuell konfiguriert werden und beginnt genau dann zu zählen, wenn der PC den Wert des ersten Konfigurationsregisters erreicht und hört auf, sobald er gleich dem Wert des zweiten Registers ist. Dazu werden beim jeweiligen Ereignis die untersten Bits der Konfigurationsregister getoggelt. Um keine Zyklen zu verlieren, muss außerdem bei Gleichheit des PCs mit den Werten in den Registern gezählt werden.

4.6 Speichern von Zählwerten und Konfigurationen

Komplexe Softwaresysteme können eine Vielzahl von Tasks beinhalten. Will man zu jedem dieser Tasks eigene Laufzeitmessungen vornehmen, kann dies eine große Anzahl an Ressourcen benötigen, da für jeden Task ein eigenes Hardwareelement reserviert werden muss.

Abhilfe schafft hier das Konzept der Zähler mit sogenannter *Task-Awareness*. In diesem Konzept können einzelne Hardwarezähler für verschiedene Tasks verwendet werden, ohne dass es zu Einbußen bei der Messung kommt. Nachteil ist jedoch, dass Zählerstände zwischengespeichert werden müssen – und dies deterministisch und effizient, um keine Messungenauigkeiten zu produzieren. Voraussetzung dafür ist, dass das Betriebssystem und die Hardware nebeneinander entwickelt werden, damit diese so gut wie möglich zusammenarbeiten können. Weiters darf diese Funktionalität nicht dazu führen, dass das System beeinträchtigt wird. Demnach hat die Speicherauslagerung folgende Anforderungen:

- Eine schnelle Speicheranbindung,
- keinen oder wenig Eingriff ins Gesamtsystem,
- konsistente Werte während der Laufzeit
  - sowohl im Speicher
  - als auch den Registern und
- kollisionsresistentes Lesen und Schreiben.

Demnach wäre eine direkte und exklusive Leitung vom Modul zum Speicher wünschenswert. Um diese Exklusivität zu erreichen, kann ein sogenannter Dual-Channel- oder Dual-Port-RAM, wie in Kapitel 4.6.1 erläutert, verwendet werden. Dabei wird einer dieser Kanäle für das Betriebssystem reserviert, der andere steht dann der Hardware zur Verfügung. Dieses Konzept zeigen auch Mauroner et al. in der Arbeit [26].

Um die Lese- und Schreibvorgänge ohne Eingriff und kollisionsresistent abwickeln zu können, darf nur zu jenen Zeiten zugegriffen werden, in denen sichergestellt werden kann, dass das Betriebssystem diese Speicherstellen nicht verwendet. Dazu ist es wichtig zu wissen, wie und wo relevante Werte gespeichert sind und wann auf diese zugegriffen wird. Diese Struktur wird in Kapitel 4.6.2 behandelt. Diese Task Awareness impliziert auch, dass die Registerwerte des alten Tasks bei einem Taskwechsel in den Speicher kommen und die neuen in die Register geladen werden. Dies wird in Kapitel 4.6.3 abgearbeitet.

4.6.1 Dual-Port-RAM

Um von zwei unterschiedlichen Stellen – in diesem Fall einerseits der Software, andererseits der PMU – gleichzeitig auf dieselben Speicherstellen zugreifen zu können, wird ein synchroner Dual-Port-Speicher verwendet, wie er auf dem eingesetzten Artix-7-FPGA vorhanden ist. Dabei handelt es sich um ein Block-RAM mit zwei synchronen Zugriffsstellen. Xilinx beschreibt diesen Speicher in [27] wie folgt:

> *The true dual-port 36 Kb block RAM dual-port memories consist of a 36 Kb storage area and two completely independent access ports, A and B. Similarly, each 18 Kb block RAM dual-port memory consists of an 18 Kb storage area and two completely independent access ports, A and B. The structure is fully symmetrical, and both ports are interchangeable. (...)*
> *Data can be written to either or both ports and can be read from either or both ports. Each write operation is synchronous, each port has its own address, data in, data out, clock, clock enable, and write enable. The read and write operations are synchronous and require a clock edge.*
> *There is no dedicated monitor to arbitrate the effect of identical addresses on both ports. It is up to you to time the two clocks appropriately. Conflicting simultaneous writes to the same location never cause any physical damage but can result in data uncertainty.*

Aufgrund dieser Spezifikation kann von beiden Ports parallel gelesen und geschrieben werden, Konflikte kann es bei Schreibzugriffen auf gleiche Speicherstellen jedoch trotzdem geben.

Um die Anforderungen dieser Applikation umsetzen zu können, wird folgendes festgesetzt:
Dem Betriebssystem beziehungsweise der ausführenden Einheit (CPU) selbst wird der erste Kanal A des Speichers zugeordnet. Dieser wird demnach für das Speichern und Laden von Daten und Programmen verwendet. Verbunden ist dieser Kanal direkt mit der Pipeline des Prozessors (siehe `vscale_pipeline.v`).
Der zweite Kanal B des Speichers wird direkt mit der PMU verbunden und somit dieser zugeordnet. Sollte der Kanal in Zukunft von weiteren Hardwareelementen benötigt werden, muss festgelegt werden, welches Hardwaremodul wann Zugriff erhält. Am einfachsten kann dies über einen simplen Zustandsautomaten erreicht werden. Dieses Konzept kommt auch schon in dieser Anwendung zum Einsatz und wird abschließend in diesem Kapitel erläutert.

Ein Blockschaltbild des nunmehr eingesetzten Speichers ist in Abbildung 4.8 dargestellt. Dabei ist lediglich der Takt und die Reset-Leitung für den Speicher gleich, die zwei unterschiedlichen Ports sind mit den Präfixen `memA_` und `memB_` gekennzeichnet. Für eine genauere Ausführung der Ein- und Ausgänge sei auf das File `mosart_dpmemory/memory.v` verwiesen.

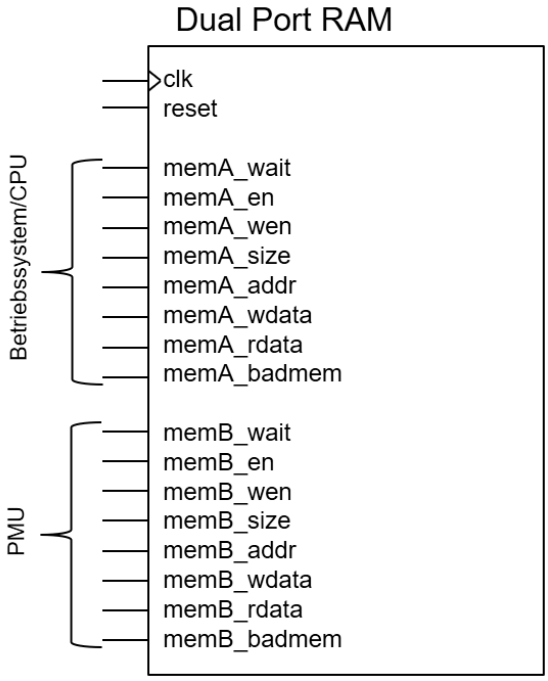


Abbildung 4.8: Die Darstellung des Port Channel RAM mit seinen Ein- und Ausgängen.

Die `memB_`-Anschlüsse des Speichers werden über die Pipeline direkt mit den im PMU-Modul mit dem Präfix `osmem_` gekennzeichneten Ein- und Ausgängen verbunden, wie in Abbildung 4.9 dargestellt.

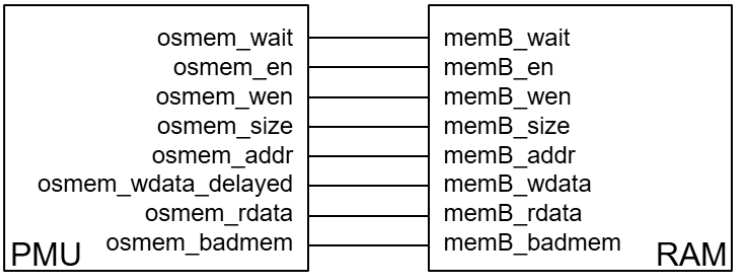


Abbildung 4.9: Blockschaltbild der Verbindung von Speicher und PMU.

Der gesamte Speicherzugriff aus dem Verilog-Code ist in Listing 4.21 zu finden. Dieser ist als Task ausgeführt und somit leicht wiederverwendbar. Auffallend dabei ist die Leitung `osmem_wdata_delayed` in Zeile 2. Hier werden die zu schreibenden Daten vor dem tatsächlichen Ablegen in ein Register gesichert.

```
1  always @(posedge clk)
2      osmem_wdata_delayed <= osmem_wdata;
3
4  task osmem_read;     // Lesezugriff
5          input [`XPR_LEN-1:0] addr;
6      begin
7          osmem_en = 1;
8          osmem_wen = 0;
9          osmem_addr = addr;
10         osmem_wdata = `XPR_LEN'h0;
11     end
12 endtask
13
14 task osmem_write;    // Schreibzugriff
15         input [`XPR_LEN-1:0] addr;
16         input [`XPR_LEN-1:0] data;
17     begin
18         osmem_en = 1;
19         osmem_wen = 1;
20         osmem_addr = addr;
21         osmem_wdata = data;
22     end
23 endtask
```

Listing 4.21: Speicherzugriff auf den Dual-Port-RAM.

Hingewiesen sei an dieser Stelle darauf, dass die Werte erst bei der nächsten steigenden Taktflanke tatsächlich in den Speicher geschrieben oder von diesem gelesen werden, da es sich um ein synchrones RAM handelt.

4.6.2 Speicherverwaltung

Da nun die Architektur sowie die Funktionalitäten des Speicherzugriffs feststehen, muss eine einheitliche Speicherstruktur zwischen Betriebssystem und Hardwaremodul bestimmt werden, damit die Werte konsistent bleiben können. Dabei bietet sich eine ähnliche Struktur wie bei den CSRs an. Aufgrund der Tatsache, dass die Konfigurations- und Zählwerte taskzugehörig sind, ist die wohl passendste Variante, diese im Task Control Block (TCB) abzulegen. Dieser Block wird vom Betriebssystem definiert und startet an der Speicherstelle, auf die der TP zeigt. Der aktuelle TP muss vom Betriebssystem immer im CPU-Register `x4 (tp)` abgelegt werden. Die genaue Registerbeschreibung der Architektur kann in

Waterman et al. [13] nachgelesen werden. Hier wird dieses Register als *Thread Pointer* bezeichnet, was in unserer Konvention einem Task entspricht.
Die Struktur eines TCB ist in Abbildung 4.10 dargestellt.

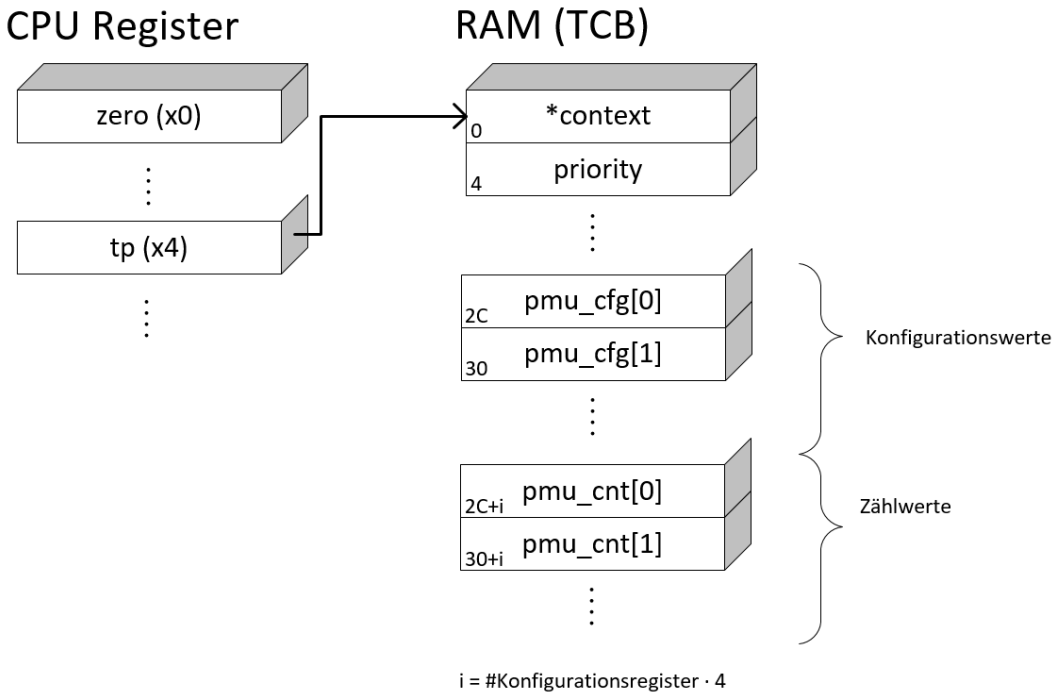


Abbildung 4.10: Schematische Darstellung des Speicherbereichs des Taskkontrollblocks (TCB) im Betriebssystem. Die Indexzahlen im TCB stellen den Adressoffset vom TP in Hexadezimaldarstellung dar.

Wie in der Abbildung zu sehen ist, beginnen die für die PMU interessanten Werte ab einem Offset von 0x2C. Die vorherigen Werte werden nur vom Betriebssystem benötigt und sind für die Hardware uninteressant. Die Konfigurationswerte beginnen also bei einer Adresse von

$$AddrCFG_{start} = tp + 0x2C, \tag{4.4}$$

wobei `tp` für die im TP-Register gespeicherte Adresse steht. Die Zählwerte beginnen demnach bei

$$AddrCNT_{start} = AddrCFG_{start} + \left(N_{cnt} \cdot N_{cfg} + \left\lceil \frac{N_{cnt}}{4} \right\rceil \right) \cdot 4. \tag{4.5}$$

Die Multiplikation mit vier erklärt sich dahingehend, dass eine Adresse in einem 32-Bit-System immer vier Bytes adressiert, weswegen die Adressen auch immer 4-Byte-aligned sind.

Sämtliche bis hierher gewählten Definitionen treffen nur auf dieses eine System zu. Wird eine andere Datenstruktur gewählt, müssen im Hardwaremodul aber lediglich die Offsets neu definiert werden. Diese Offsets befinden sich im Konfigurationsfile `vscale_pmu.vh` und sind, wie in Listing 4.22 gezeigt, ausgelegt, jedoch frei konfigurierbar.

```
1 `define OS_OFFSET_PMU_CFG   32'h2C
2 `define OS_OFFSET_PMU_CNT   (`OS_OFFSET_PMU_CFG +
3                               (`CSR_PMU_N * `CSR_PMU_CFG_N
4                               + `CSR_PMU_N/4) * 4)
```

Listing 4.22: Adressoffsets im TCB.

4.6.3 Speichern und Laden beim Taskwechsel

In diesem Kapitel wird die eigentliche Ein- und Auslagerung von Konfigurations- und Zählwerten in den Speicher erläutert. Dabei werden, um die Daten konsistent zu halten, immer alle Zähl- und Konfigurationsregister in den Speicher geladen. Zurück vom Speicher in die Register kommen jedoch nur die in Kapitel 4.4.1 erklärten Werte.

Als Zeitpunkt für den Wertwechsel wird die Änderung des TP gewählt, da da ein neuer Task gestartet oder fortgeführt wird. Ein weiterer Vorteil ist, dass das Betriebssystem zu diesem Zeitpunkt meist noch mit organisatorischen Tätigkeiten beschäftigt ist, bevor der Programmcode des Tasks wirklich ausgeführt wird. Ist die Logik also schnell genug beim Aus- und Einlagern, kann der Programmcode des Tasks schon mit gültigen Werten der PMU arbeiten. Andernfalls kann es vorkommen, dass die Hardware noch mit dem Speicherzugriff beschäftigt ist und demnach ungültige Werte in den Registern stehen.

Um den Speicherzugriff aus einem Hardwaremodul zu ermöglichen, wird ein endlicher Zustandsautomat zur Hilfe genommen. Dabei müssen für jede *Wertart* ein Zustand für die Speicherung im RAM und ein Zustand für das Lesen aus dem Speicher implementiert werden. Da die Zählwerte aufgetrennt in High- und Low-Byte abgelegt sind, ergeben sich insgesamt sechs Zustände und ein Leerlaufzustand.
Um die durchgängige Skalierbarkeit des Moduls aufrechtzuerhalten, müssen die einzelnen Zustände immer abhängig von der Konfiguration der PMU unterschiedlich viele Werte ihrer Wertart umspeichern.

Die dazu nötige Next-State-Logik inklusive dem Schreiben in den Speicher ist im Anhang B zu finden.

Die Zustände des Automaten sind in Abbildung 4.11 dargestellt.

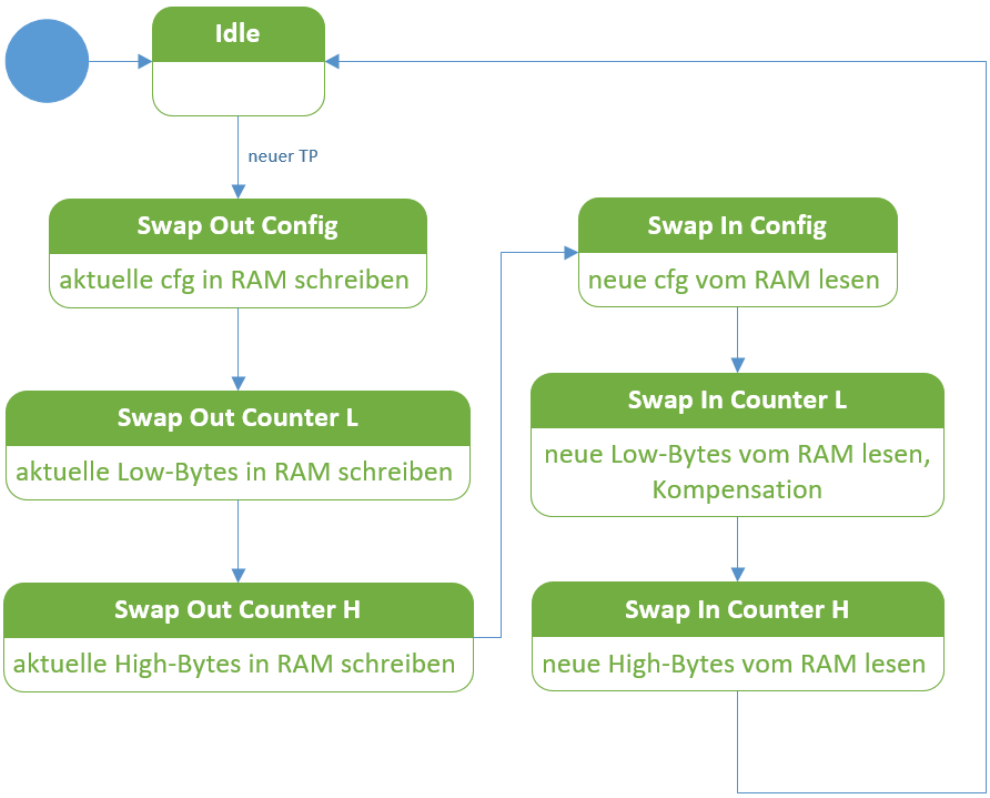


Abbildung 4.11: Zustandsautomat, der die Ein- und Auslagerung von Zähl- und Konfigurationswerten auf den RAM abarbeitet.

Dabei ist im Zustand *Swap In Counter L* die Kompensation angegeben, mit welcher etwaige verlorene Taktzyklen während des Speicherzugriffsprozesses kompensiert werden können. In der Implementierung der einzelnen Lesezustände wird außerdem überprüft, ob der aktuell bearbeitete Wert überhaupt eingelesen werden soll und darf, wie in Kapitel 4.4.1 definiert.

```
1  case(state)
2    PMU_IDLE: [...]
3    PMU_SWAP_IN_CFG:
4      begin
5        if((index - 1) >= (`CSR_PMU_N/4) && // cfg_main auslassen
6          // nur cfgs mit MSB = 11 behandeln
7          (cfg[(index - 1 - `CSR_PMU_N/4) /
8            `CSR_PMU_CFG_N][(`XPR_LEN/4)-1 :
9            (`XPR_LEN/4)-2] == 2'b11))
10       begin
11         pmu_cfg[index-1] <= osmem_rdata;
12       end
13     end
14   PMU_SWAP_IN_CNT_L:
15     begin
16       if(cfg[index - 1][(`XPR_LEN/4)-1] == 1)
17       begin
18         // Kompensation
19         case (cfg[index - 1])
20           `PMU_TASK_OVERALL_COUNTER:
21             pmu_cycles[index-1][0+:`XPR_LEN] <= osmem_rdata +
22                                                 `PMU_LOST_CYCLES_MAX;
23           default:
24             pmu_cycles[index-1][0+:`XPR_LEN] <= osmem_rdata;
25         endcase
26       end
27     end
28   PMU_SWAP_IN_CNT_H:
29     begin
30       if(cfg[index - 1][(`XPR_LEN/4)-1] == 1)
31         pmu_cycles[index-1][`XPR_LEN+:`XPR_LEN] <= osmem_rdata;
32     end
33 endcase
```

Listing 4.23: Implementierung der Lesezustände.

Die Kompensation wird, wie in den Zeilen 19ff zu sehen, für jede Zählerart gesondert behandelt. Sinnhaft ist diese Kompensation ohnehin nur für Zähler, die auf den Speicher gesichert werden, da die anderen Zähler gar nicht angehalten werden. Momentan hat lediglich der `PMU_TASK_OVERALL_COUNTER` diese ausgeführt, da der Kompensationswert hier gleich der Dauer des gesamten Speicherzugriffs ist. Dieser Wert kann in `vscale_pmu.vh` eingestellt werden und ergibt sich für die vorliegende Zählerart in

$$LostCycles_{max} = \left(N_{cnt} \cdot 2 + N_{cnt} \cdot N_{cfg} + \left\lceil \frac{N_{cnt}}{4} \right\rceil \right) \cdot 2 + 4. \tag{4.6}$$

Er setzt sich also aus der doppelten Summe aller Register – da ja ein- und ausgelagert werden muss – und einem Offset von vier Zyklen zusammen. Diese vier Zyklen lassen sich wie folgt erklären: Ein Zyklus zu Beginn, um den Speicher zu aktivieren, zwei Zkyklen in der Mitte, wo von Schreib- auf Lesezugriff umgestellt wird und ein Zyklus am Ende zum Abschließen des Vorganges.

Dieser Vorgang wird anhand der Simulationsausgabe in Abbildung 4.12 erkennbar. Verwendet wurde dabei ein Modul mit vier Zählregistern und einem Konfigurationsregister pro Zählregister, also insgesamt 13 32-Bit-Werte. Auf die Werte in der Darstellung ist dabei nicht zu achten, da es sich nur um eine Testsimulation handelt. Die gezeigten Werte entsprechen den Registerwerten, der Speicher ist hier nicht gezeigt.
Genauere Anwendungen und Messungen werden in den Kapiteln 5 und 6 ausgeführt.

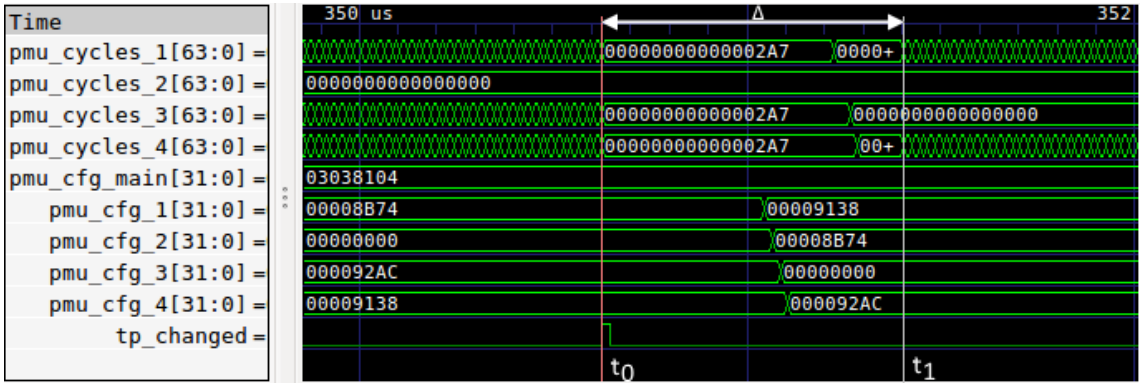


Abbildung 4.12: Die Ein- und Auslagerung von Konfigurations- und Zählwerten in einer Simulation.

Der Zeitpunkt t_0 ist jener Zeitpunkt, zu dem der Prozess mit dem Auslagern auf den Speicher gestartet wird, t_1 markiert das Ende des Einlesens aus dem Speicher. Ab t_1 beginnen auch, wie man erkennen kann, einige Zähler wieder zu laufen. Die Gesamtdauer dieses Prozesses kann mit $\Delta = t_1 - t_0$ berechnet werden.

Als Beispiel sei hier zuerst die Hauptkonfiguration in `pmu_cfg_main` herangezogen: Diese ändert sich während des gesamten Prozesses nicht, wird jedoch trotzdem in den Speicher geschrieben. Im Gegensatz dazu wird das Konfigurationsregister `pmu_cfg_1` von 0x8B74 zu 0x9138 geändert. Die Leitung `tp_changed` kennzeichnet den Zeitpunkt der TP-Änderung.

Kapitel 5

Eingliederung und Anwendung in *mosartMCU*-OS

Da nun die Hardware als solches beschrieben wurde, ist der nächste Schritt, das entwickelte PMU-Modul auch softwareseitig ansprechen zu können. Das dabei verwendete Betriebssystem ist ein minimales Operating System (OS), das im Zuge des *mosartMCU*-Projektes entwickelt wurde.

Dieses Kapitel gibt einführend einen Überblick über das Betriebssystem selbst, anschließend wird auf die Umbaumaßnahmen zur Eingliederung der PMU-Funktionen eingegangen um abschließend die Anwendung eines Benutzers zu schildern.

5.1 Überblick über *mosartMCU*-OS

Das *mosartMCU*-OS ist ein in C und Assembler geschriebenes Betriebssystem, welches ein einfaches Task- und Ressourcenverwaltungssystem für eine RISC-V-Architektur beinhaltet. Dazu gibt es diverse Syscalls, um einem Benutzer zu ermöglichen, das Betriebssystem zu starten, Tasks anzulegen und zu verwalten sowie Ressourcen und Events zu verwenden. Der eingebaute Scheduler kümmert sich dann um die einzelnen Tasks und deren Verwaltung.

5.1.1 Anwendungsüberblick

Um das Betriebssystem verwenden zu können, muss das Headerfile `mosart_os.h` eingebunden werden, in dem alle Strukturen und Funktionen zum Betrieb definiert sind. Ein simples Programm mit einem einzigen Task würde demnach wie in Listing 5.1 aussehen.

```
1  #include <mosart_os.h>
2
3  int task1(void);
4
5  void main(void)
6  {
7      os_register_task(task1, 1280, 100);
8      os_run();
9
10     while(1);
11 }
12
13 int task1(void)
14 {
15     while(1) sleep(5);
16     return 0;
17 }
```

Listing 5.1: Simples Programm mit einem Task.

Mit `os_register_task(..)` in Zeile 7 wird der Task `task1` mit einem Stack von 1280 Bytes und einer Priorität von 100 registriert und anschließend nach dem Aufruf von `os_run()` in Zeile 8 ausgeführt. Der Task selbst besteht aus der Funktion `task1`, welche ab Zeile 13 zu finden ist.

Für weitere Funktionen und Syscalls des Betriebssystems sei auf Anhang C verwiesen.

5.1.2 Interne Taskverwaltung

Jeder Task hat in diesem Betriebssystem seinen eigenen Kontext sowie eine Priorität. Angedeutet wird dies bereits in Abbildung 4.10, wo man den TCB eines solchen Tasks sieht. Wie bereits erwähnt findet sich der Pointer auf den aktuellen TCB im Register `tp (x4)` des Prozessors. Dies geschieht bereits automatisch und wird vom Scheduler in der nachfolgend als `ready_queue` bezeichneten Variable verwaltet.

Die Verwendung eines Registers in C wird in Listing 5.2 gezeigt.

```
1  register os_tcb_t* ready_queue asm ("tp");
```

Listing 5.2: Verwendung des Registers `tp` als Variable `ready_queue` im Programmcode.

Der Scheduler kümmert sich also betriebssystemseitig um das Hardwareregister `tp`, wodurch eine sehr einfache Schnittstelle zwischen Hard- und Software

geschaffen wird. Die Hardware hat nun Zugriff auf den TP und kann auf Änderungen reagieren, was für die PMU essentiell ist.
Geändert wird der Wert in diesem Register, sobald ein Task im System zur Ausführung kommt. Bei diesem Taskwechsel werden Informationen über diesen Task vom RAM geladen und der Task dort weiter ausgeführt, wo er zuletzt unterbrochen wurde. Zu Unterbrechungen eines Tasks kann es zum Beispiel beim manuellen Schlafenlegen mittels `sleep(..)` oder beim Warten auf eine Ressource kommen.

5.2 Integration der PMU

Die PMU stellt einerseits CSRs zur Konfiguration und zum Auslesen von Zählwerten zur Verfügung, andererseits muss sie auf Taskwechsel, die vom Betriebssystem initiiert werden, reagieren. Da die Reaktion auf einen solchen Wechsel bereits in den vorhergehenden Kapiteln abgehandelt wurde, wird hier nur auf den Zugriff auf die CSRs eingegangen.

Weiters müssen die Zähl- und Konfigurationswerte, wie in Kapitel 4.6.2 erklärt, in den TCB der Tasks eingefügt werden. Der Anwendungsentwickler sollte dabei ebenfalls Zugriff auf diese Werte haben.

Die in weiterer Folge verwendeten C-Makros und Konstanten sind in `pmu.h` definiert. Eine Übersicht über die Syscalls und Funktionen kann in Anhang C nachgeschlagen werden. Die zugehörige Registerbeschreibung findet sich in Anhang A.

5.2.1 Zugriff auf die Register

Um auf CSRs einer RISC-V-Maschine zuzugreifen, definieren Waterman et al. im Befehlssatz [13] diverse Instruktionen. Diese Instruktionen sind dabei als nicht-unterbrechbare Atomic-Instruktionen ausgeführt. Die wichtigsten, als Assembler-Befehle formulierten Instruktionen sind `csrr` und `csrw` zum Lesen respektive Schreiben auf eine CSR-Adresse.

Dabei wird der Lesebefehl als

```
csrr rd, csr
```

verwendet, wobei `rd` das Prozessorregister definiert, in dem der zu lesende Wert gespeichert ist und `csr` die Adresse, von welcher gelesen werden soll, festlegt.

Um auf ein CSR zu schreiben, muss der Befehl

csrw csr, rs1

ausgeführt werden, wobei `csr` die Schreibadresse und `rs1` das Register, welches den zu schreibenden Wert beinhaltet, darstellt.

Will man diese Funktionen in C-Code zu integrieren, muss sogenanntes Inline-Assembler mit dem Schlüsselwort `asm volatile` verwendet werden. Dies wird in weiterer Folge in den Listings 5.3 und 5.4 gezeigt und erläutert.

```
1 asm volatile (" csrr    %0, hpmcounter3" : "=r" (dest));
```

Listing 5.3: Inline-Assembler eines `csrr`-Aufrufs.

Hier wird auf das CSR `hpmcounter3` lesend zugegriffen und der Wert im Register `dest` abgelegt. `hpmcounter3` stellt dabei laut [3] die Adresse 0xC03 dar. Dabei gibt es laut Definition die Assembler-Kürzel der Form `hpmcounterN` und `hpmcounterNh`, wobei `N` auf einem Intervall von $[3, 31]$ definiert ist. Diese Kürzel stehen für die Adressen der jeweiligen Zählwerte der PMU.

Schreibend kann jedoch nur auf die Konfigurationsregister zugegriffen werden.

```
1 asm volatile (" mv      s1, %0  \n\t\
2                 csrw    %1, s1  "
3                 : : "r"(source),  "g"(PMU_CFG_START));
```

Listing 5.4: Inline-Assembler eines `csrw`-Aufrufs.

Dabei muss, wie in Zeile 1 gezeigt, zuerst der gewünschte Wert in `source` in einem Register (hier `s1`) abgelegt werden. Dies geschieht mit dem Befehl `mv`. Nachfolgend wird dieser Wert in Zeile 2f in das CSR mit der Adresse `PMU_CFG_START (0x8f0)` geschrieben. Illustriert wird in diesem Beispiel also das Setzen des ersten Hauptkonfigurationsregisters.

Um nun explizit die Werte des Hardwaremoduls anzusprechen, werden in weiterer Folge die Syscalls des Betriebssystems und weitere Funktionen und Makros erläutert, die dies implementieren. Ausgegangen wird von einer PMU mit vier Zählregistern und zwei Konfigurationsregistern pro Zählregister. Dabei wird genau ein Hauptkonfigurationsregister erstellt. Die Software ist dabei nicht skalierbar gehalten, da dies einen nicht zu vertretenden Mehraufwand bedeuten würde. Jedoch ist der Code so gehalten, dass er im Falle einer Rekonfiguration einfach verändert und erweitert werden kann.
Die zu implementierenden Funktionen beinhalten das Setzen des Hauptkonfigurationsregisters und der Konfigurationsregister sowie das Auslesen der aktuellen Zählwerte.

Setzen des Hauptkonfigurationsregisters

Das Setzen des Hauptkonfigurationsregisters wird mit dem Syscall `pmu_cfg_main(uint32_t main)` abgehandelt. Dieser schreibt den Wert des Übergabeparameters in `main` in das Register mit der Adresse `PMU_CFG_START` und ist in Listing 5.5 prinzipiell dargestellt.

```
1 void pmu_cfg_main(uint32_t main )
2 {
3     asm volatile (" mv      s1, %0  \n\t\
4                     csrw    %1, s1  "
5                     : : "r"(main),  "g"(PMU_CFG_START));
6 }
```

Listing 5.5: Syscall zum Setzen des Hauptkonfigurationsregisters.

Um einen gültigen Wert für das Hauptkonfigurationsregister zu generieren, bietet `pmu.h` das Makro `CFG(a, b, c, d)` an, welches aus vier Bytes einen 32-Bit-Wert erstellt. Als Parameter können dann die Bytes für die ebenfalls hier als Konstanten definierten Zählerarten (`PMU_ALL_COUNTER`, ...) verwendet werden. Eine beispielhafte Verwendung ist in Listing 5.6 gezeigt.

```
1 pmu_cfg_main(CFG(PMU_TASK_PART_COUNTER,
2                  PMU_TASK_INTERRUPT_COUNTER,
3                  PMU_TASK_OVERALL_COUNTER,
4                  PMU_TASKTIME_COUNTER));
```

Listing 5.6: Verwendung des Makros `CFG`.

Setzen der Konfigurationsregister

Um nun die weiteren acht Konfigurationsregister zu beschreiben, kann der Syscall `pmu_cfg(..)` verwendet werden. Diese Funktion setzt dabei die beiden zu einem Zähler zugehörigen Konfigurationsregister. Die prinzipielle Funktionalität ist in Listing 5.7 dargestellt.

```
1  void pmu_cfg(uint32_t offset, uint32_t cfg_1, uint32_t cfg_2)
2  {
3      if(offset == 0)
4      {
5          asm volatile (" mv      s1, %0  \n\t\
6                          csrw    0x8f1, s1   " :: "r"(cfg_1));
7          asm volatile (" mv      s1, %0  \n\t\
8                          csrw    0x8f2, s1   " :: "r"(cfg_2));
9      } else // ... (weitere Offsets)
10 }
```

Listing 5.7: Syscall zum Setzen der Konfigurationsregister.

Der Parameter `offset` bezeichnet dabei den Index des Zählers, für welchen die Konfiguration gesetzt werden soll. `cfg_1` und `cfg_2` sind die jeweils zu konfigurierenden Werte. Für vier Zählregister müssen die korrespondierenden vier Verzweigungen mit ihren Adressen implementiert werden.

Lesen der Zählwerte

Die aktuellen Werte in den Zählregistern der PMU können mit dem Syscall `pmu_get_cycles(uint32_t index)` ermittelt werden. Diese Funktion greift je nach übergebenem Index auf die beiden 32-Bit-Werte des zugehörigen Zählregisters zu und liefert diese als 64-Bit-Wert zurück. Diese Funktionalität ist prinzipiell in Listing 5.8 ausgeführt.

```
1  os_time_t pmu_get_cycles(uint32_t index)
2  {
3      uint32_t cyclesH, cyclesL; // 32-Bit-Werte
4
5      switch(index)
6      {
7          case 0:
8              asm volatile (" csrr %0, hpmcounter3 \n\t\
9                              csrr %1, hpmcounter3h"
10                             : "=r" (cyclesL), "=r" (cyclesH));
11             break;
12         // ... (weitere Indizes)
13     }
14     // Umwandlung in 64-Bit-Wert
15     return ((os_time_t)cyclesH << 32) | cyclesL;
16 }
```

Listing 5.8: Syscall zum Lesen der aktuellen Zählwerte.

Um auf den aktuellen Wert eines Zählers zuzugreifen, muss sowohl der Low- als auch der High-Wert gelesen werden. Aus den beiden in den Zeilen 8 und 9 gelesenen Werten wird dann der in Zeile 15 zurückgegebene 64-Bit-Wert des Datentyps `os_time_t` generiert.
Für vier Zähler müssen hier wiederum die vier Verzweigungen ausgeführt werden.

5.2.2 Task Control Block

Der Task Control Block (TCB) eines Tasks ist jener Speicherbereich, auf den der Task Pointer (TP) eines Tasks verweist. In diesem sind Informationen, wie die Adresse des Stacks oder die Priorität abgelegt. Die bestehende Struktur ist in Listing 5.9 dargestellt.

```
1  struct os_tcb{
2      os_reg_t         *context;
3      os_priority_t    priority;
4      struct os_tcb    *next;
5      os_time_t        timeout;
6      struct os_tcb    *next_timeout;
7      os_priority_t    base_priority;
8      struct os_tcb    **member_list;
9      uint32_t         dummy[2];
10 };
11 typedef struct os_tcb os_tcb_t;
```

Listing 5.9: Bestehende TCB-Struktur.

Diese Struktur wird nun um die in Kapitel 4.6.2 erwähnten Speicherstellen erweitert, sodass sowohl Hard- als auch Software auf dieselben Werte zugreifen. Dies ist in Listing 5.10 illustriert.

```
1  struct os_tcb{
2      [...]
3      uint32_t     pmu_cfg[CSR_PMU_N * CSR_PMU_CFG_N + CSR_PMU_N/4];
4      uint32_t     pmu_cnt[CSR_PMU_N * 2];
5  };
6  typedef struct os_tcb os_tcb_t;
```

Listing 5.10: Erweiterte TCB-Struktur.

Um nun von einem Anwendungsprogramm auf die hier gespeicherten Zählwerte zugreifen zu können, wird ein Syscall verwendet. Dieser greift aufgrund des übergebenen TP-Wertes auf die gespeicherten Werte zu und übergibt sie der aufrufenden Stelle. Der Syscall `pmu_get_task_cycles(..)` ist in Listing 5.11 in Pseudocode dargestellt.

```
1  integer[] pmu_get_task_cycles(os_id task_pointer)
2      integer[] counters
3      counters <- get_counters_of_task(task_pointer)
4      return (counters)
```

Listing 5.11: Syscall zum Auslesen von Zählwerten aus dem TCB.

Hier wird nur auf die gespeicherten Werte zugegriffen, nicht auf die Zählwerte in den CSRs. Demnach kann es sein, dass gewisse Werte nicht aktuell sind. Die Funktion dient daher Zwecken der Überwachung verschiedener Tasks aus einem anderen Task heraus. Als Rückgabewert wird ein Array aus 32-Bit-Zählwerten geliefert.

Um also einen anderen Task überwachen zu können, muss man über diese Funktion dessen Daten auslesen. Den TP bekommt man beim Erstellen eines Tasks mit `os_register_task(..)` zurückgeliefert, wie in Listing 5.12 illustriert.

```
1  // ...
2  os_id_t tp1, tp2;
3  void main()
4  {
5      // ...
6      tp1 = os_register_task(task1, 16*80, 100);
7      tp2 = os_register_task(task2, 16*80, 10);
8      // ...
9  }
10
11 int task1(void)
12 {
13     // ...
14     uint32_t* cnts = pmu_get_task_cycles(tp2);
15     // ...
16     return 0;
17 }
18 // ...
```

Listing 5.12: Profiling eines anderen Tasks.

Wesentliche Teile, wie das Konfigurieren der Zähler, wurde hier ausgelassen. Es wird nur beispielhaft veranschaulicht, wie ein Task auf die Zählwerte eines anderen Tasks zugreifen kann. Dies ist natürlich für eine beliebige Anzahl an Tasks erweiterbar. Abschließend sei noch einmal darauf hingewiesen, dass es sich bei diesen Ständen um die zuletzt in den RAM ausgelagerten Werte des jeweiligen Tasks handelt.

Kapitel 6

Messungen und Vergleiche

In diesem Kapitel wird das entwickelte Modul hinsichtlich hardwareseitigem Ressourcenverbrauch und Korrektheit der inneren Funktionalität analysiert. Dazu wird es verschiedenen Tests und Messungen unterzogen.

Zuerst wird auf den Hardwareressourcenverbrauch der Lösung am Zielsystem, also am verwendeten FPGA-Board eingegangen. Hier ist von Interesse, wie viele Logikeinheiten das entwickelte Modul benötigt und wie sich der Verbrauch bei verschiedenen Konfigurationen hinsichtlich der Anzahl der Zähler und Konfigurationsregister je Zähleinheit verhält. Diskutiert werden soll auch, in welcher Art und Weise sich die Anzahl der Logikeinheiten ändert und was der zugrundeliegende Faktor für die Änderung ist.

Im nächsten Schritt werden verschiedene Softwaretests implementiert. Die dabei ermittelten Zählwerte werden mit real gemessenen Werten verglichen. Dazu müssen die mit dem Hardwaremodul ermittelten Laufzeiten in Sekunden umgerechnet werden und anschließend mit real gemessenen Zeiten verglichen werden. Es muss ein geeignetes Messgerät gepaart mit einem Messausgang an der Hardware verwendet werden. Auf etwaige Abweichungen dieses Vergleichs und insbesondere deren Gründe muss eingegangen werden.

Abschließend wird – auch aufgrund der in diesem Kapitel gelieferten Ergebnisse – ein Fazit gezogen.

6.1 Ressourcenverbrauch am FPGA

In Folge wird der Ressourcenverbrauch analysiert, den die entwickelte PMU auf dem Entwicklungsboard benötigt. Da das Modul skalierbar im Bezug auf Zählregister und Konfigurationsregister je Zählregister ausgelegt ist, wird insbesondere diese Eigenschaft beleuchtet.

Als Referenzmodell dient eine Einheit, die aus vier Zählregistern und zwei Konfigurationsregistern je Zählregister aufgebaut ist. Der Ressourcenverbrauch kann mit dem Synthesetool Vivado 2016.2[1] von Xilinx dargestellt werden. Hier ergibt sich nach der Synthese für das FPGA der in Tabelle 6.1 angegebene Verbrauch. Es handelt sich folglich um die sogenannten *Primitive Statistics*, *Net Boundary Statistics* und den *Clock Report*.

Typ	*Anzahl*
FLOP_LATCH	684
LUT	2083
MUXFX	68
CARRY	167
MULT	2
OTHERS	7
NETS	623
CLK Inst	686

Tabelle 6.1: Der Ressourcenverbrauch am FPGA bei einer PMU mit vier Zählern und je zwei Konfigurationsregistern.

Dabei steht FLOP_LATCH für ein Flip-Flop, LUT für Look-Up-Tables (LUTs), MUXFX für Multiplexer, CARRY für Carry-Chains, also lokale Routings zwischen Logikblöcken, MULT für Multiplikatoren und OTHERS für die restlichen weiteren Stammfunktionen. Die genannten Kenngrößen lassen sich in die Gruppe der *Primitives* eines einzelnen Configurable Logic Block (CLB) eines FPGAs zusammenfassen.

Des Weiteren gibt es dann noch einen Kennwert für Netze (*Net Boundary*), die sich über die Grenze des Moduls hinaus erstrecken. Diese sind in der Tabelle mit dem Kürzel NETS angegeben.

Im *Clock Report* ist angegeben, wie viele Taktinstanzen (CLK Inst) von der Lösung verwendet wurden.

[1] https://www.xilinx.com/products/design-tools/vivado.html

6.1.1 Kennwerte bei verschiedenen Konfigurationen

Die vorher genannten Kennwerte werden nun bei verschiedenen Konfigurationen beleuchtet und in Tabelle 6.2 gegenübergestellt. Die Spaltenüberschrift bezeichnet dabei immer, welche Anzahl an Zählern und Konfigurationsregistern verwendet wurde (4x2 bedeutet 4 Zähler mit je 2 Konfigurationsregistern).

	4x2	*4x4*	*4x8*	*8x2*	*16x2*
FLOP_LATCH	682	941	1453	1235	3141
LUT	1955	2380	2735	3994	9865
MUXFX	40	163	328	317	1095
CARRY	198	197	208	371	840
MULT	2	2	2	2	2
OTHERS	7	7	7	7	7
NETS	513	515	502	661	1473
CLK Inst	684	943	1455	1237	3113

Tabelle 6.2: Der Ressourcenverbrauch am FPGA bei einer PMU mit verschiedenen Konfigurationen hinsichtlich Zähler- und Konfigurationsregisteranzahl.

Man kann bereits erkennen, dass sich die Werte für MULT und OTHERS nicht ändern, andere Werte scheinen sich jedoch einem gewissen Muster folgend zu verändern. Das bedeutet, dass sich bei der Rekonfiguration die Anzahl der verwendeten Multiplizierer und anderen Stammfunktionen nicht ändert.

Um die Änderung der verbleibenden Werte in Relation zum Ausgangsmodul zu verdeutlichen, wird in Tabelle 6.3 das Verhältnis prozentuell dargestellt. Zusätzlich werden in dieser Tabelle auch noch die Dimension der rekonfigurierten Größen getrennt betrachtet und demnach die Konfiguration 4x2 zweimal angeführt.

	4x2	*4x4*	*4x8*	*4x2*	*8x2*	*16x2*
FLOP_LATCH	100%	137,98%	213,08%	100%	181,09%	460,56%
LUT	100%	121,74%	139,90%	100%	204,30%	504,60%
MUXFX	100%	407,50%	820,00%	100%	792,50%	2737,50%
CARRY	100%	99,49%	105,05%	100%	187,37%	424,24%
NETS	100%	100,39%	97,86%	100%	128,85%	287,13%
CLK Inst	100%	137,87%	212,72%	100%	180,82%	458,04%

Tabelle 6.3: Die Änderung des Ressourcenverbrauchs am FPGA bei einer PMU mit verschiedenen Konfigurationen hinsichtlich Zähler- und Konfigurationsregisteranzahl in Prozent.

6.1.2 Auswertung und Diskussion

Zusammengefasst kann man die Änderung der verschiedenen Ressourcen bei Variation der Anzahl der Konfigurationsregister je Zähleinheit in Abbildung 6.1 sehen.

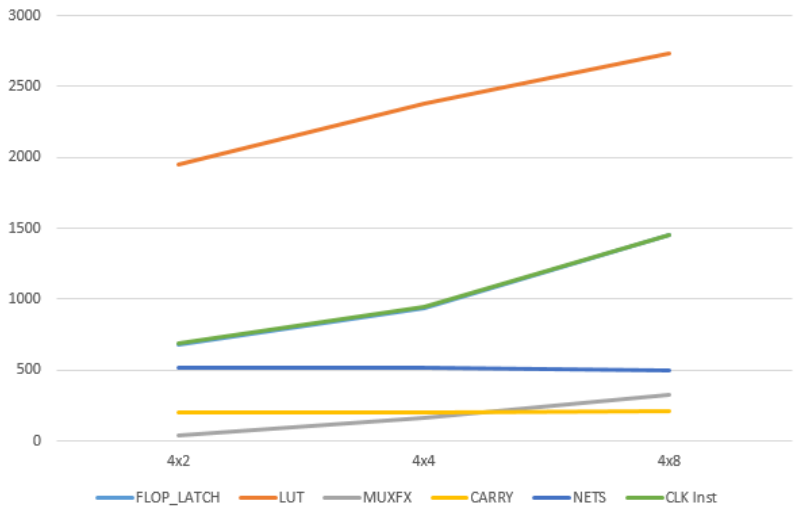


Abbildung 6.1: Die Darstellung der Ressourcenänderung bei der Variation der Anzahl der Konfigurationsregister je Zähleinheit.

Es ist ersichtlich, dass die Anzahl an Flip-Flops im ersten Schritt, also bei der Erhöhung auf vier Konfigurationsregister, um 37,98% erhöht. Im zweiten Schritt erhöht sich der Wert dann um 113,05%. Die Änderung verdoppelt sich also in etwa beim zweiten Schritt. Dies kann darauf zurückgeführt werden, dass die Anzahl der Zähleinheiten gleich bleibt, sich jedoch die Anzahl der Konfigurationsregister verdoppelt. Register manifestieren sich hier als Flip-Flops.
Die LUTs verändern sich im ersten Schritt kaum, beim zweiten Schritt kommen insgesamt 21,74% an Ressourcen hinzu. Da Konfigurationsregister im Modul nicht weitschichtig verknüpft sind und auch keine Logikeinheiten daran hängen, kann man diese kleine Änderung damit erklären.
Die Anzahl der Multiplexer ändert sich im ersten Schritt um den Faktor 4,08 und im zweiten Schritt um 8,20. Dieser Verlauf ist fast linear, was sich durch die hinzugekommenen Adressierungen der Register erklären lässt. Doppelt so viele Register benötigen doppelt so viele Adressmultiplexer.
Die Carry-Chains verändern sich noch minimaler als die LUTs, was sich mit den gleichen Eigenschaften erklären lässt.
Die Nets sind als Kenngröße hier wenig aussagekräftig, da diese vom Synthesetool optimiert eingefügt werden. Erkennbar ist jedoch, dass die Konfiguration 4x8 mit den wenigsten Nets auskommt.
Die Änderung der Taktinstanzen verhält sich gleich wie die der Flip-Flops, was ebenfalls mit der gleichen Erklärung begründbar ist.

Die Änderung der Ressourcen bei der Variation der Zählregister wird in Abbildung 6.2 dargestellt.

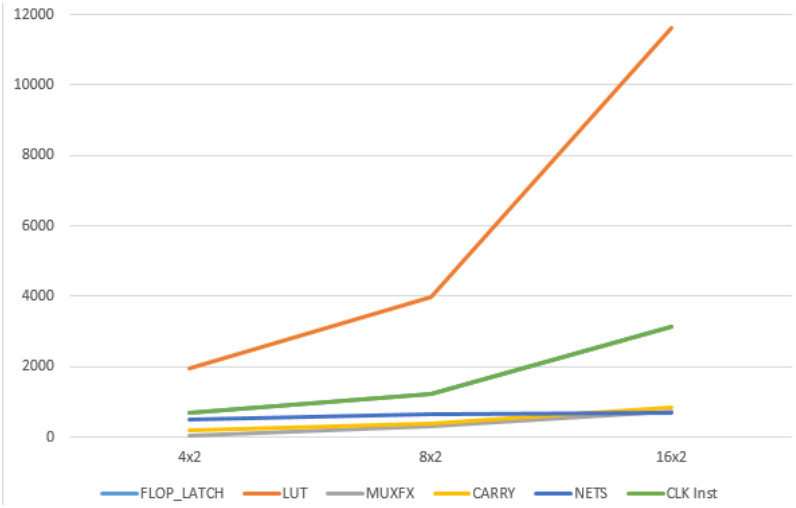


Abbildung 6.2: Die Darstellung der Ressourcenänderung bei der Variation der Anzahl der Zählregister.

Hier erkennt man schon, dass sich die Anzahl an Logikeinheiten bedeutend stärker ändert als im vorherigen Fall.
Die Anzahl der Flip-Flops steigt auf 181,09% im ersten und auf 460,56% des Referenzmoduls im zweiten Schritt an. Dabei muss man beachten, dass sich bei einer Verdoppelung der Anzahl der Zählregister auch die Gesamtanzahl an Konfigurationsregistern verdoppelt, da diese ja gekoppelt sind. Damit lässt sich auch die enorme Änderung erklären. Der Verlauf der Taktinstanzen folgt diesem in ähnlicher Art und Weise.
Da jede Zähleinheit mit einer eigenen Logikinstanz versorgt wird, erfährt die Anzahl an LUTs in beiden Schritten in etwa eine Verdoppelung ihres Werts. Auch die Carry-Chains folgen in etwa dieser Verteilung.
Die Erhöhung in der Anzahl an Multiplexern, die ebenfalls um einiges höher als im vorherigen Fall ausfällt, kann mit dem Umfang an Adressierungen der Register erklärt werden. Der Anstieg folgt dabei in etwa einem quadratischen Verlauf, wie auch schon zuvor.
In diesem Fall ist die Konfiguration 4x2 diejenige, die mit den wenigsten Nets auskommt.

6.2 Messungen an Testsystemen

Die hier durchgeführten Messungen beziehen sich auf Softwaretestsysteme, die auf der Hardware ausgeführt werden. Die so erreichten Zählerstände werden zu ausgewählten Zeiten festgehalten und auf Zeiten umgerechnet. Dies geschieht, indem man

$$t = Cnt \cdot \frac{1}{f_{CPU}} = Cnt \cdot \frac{1}{50MHz} \tag{6.1}$$

rechnet. Dabei ist Cnt der Zählerstand und f_{CPU} die Taktfrequenz der CPU, welche mit 50 MHz angenommen wird. Das Ergebnis t ist die gemessene Zeit in Sekunden.

Um diese berechneten Werte vergleichen zu können, wird ein digitales Oszilloskop vom Typ PicoScope 2205 MSO verwendet. Der Messaufbau inklusive dem Entwicklungsboard ist in Abbildung 6.3 ersichtlich.

Abbildung 6.3: Die Darstellung des Messaufbaus mit dem USB-Oszilloskop PicoScope 2205 MSO.

6.2.1 Durchführung

Das PicoScope ist ein 2+16-Kanal-Oszilloskop. Es hat laut [28] neben zwei analogen auch noch 16 digitale Kanäle, die allesamt simultan verwendet werden können. Es muss über USB an einen PC angeschlossen werden, die Darstellung der Messwerte erfolgt über eine PC-Software. Die maximale Eingangsfrequenz für digitale Signale beträgt 100 MHz.

Im hier gezeigten Messaufbau werden alle 16 digitalen Kanäle verwendet, die analogen Kanäle finden hingegen keine Verwendung.

Der C-Code zu den einzelnen Testfällen ist in Anhang D zu finden. Jeder dieser Testfälle beinhaltet zwei Tasks mit den TPs 0x8B64 und 0x9228 und bis zu vier Zähler mit verschiedenen Konfigurationen.

Bei der Durchführung der Messungen werden zuerst Simulationen mit dem Wave-Viewer GTKWave[2] analysiert und diese dann mit einer Hardwaremessung mit dem PicoScope verglichen. In der Simulation sind die einzelnen Zählwerte mit `pmu_cycles_1-4` gekennzeichnet. Zusätzlich sind die Enable-Flags der jeweiligen Zähler mit `cnt_en_1-4` ersichtlich. Weitere vorkommende Signale sind `pmu_cfg_main` für das Hauptkonfigurationsregister, `tp` für den TP, `prv` für den Privilege Mode der CPU, `pc` für den aktuellen PC, `mip` für die aktuell anstehenden Interrupts und `ext` für den externen Port der PMU.
Bei der Oszilloskopmessung kommt der Measure-Port der PMU zum Einsatz, über den einerseits Signale, auf die getriggert werden kann, und andererseits Zählerstände aus den Registern herausgeführt werden. In den folgenden Oszilloskopbildern sind immer erstere Signale dargestellt. Die genaue Konfiguration ist in den einzelnen Testfällen angegeben.

t_overall.c

Dieser Testfall illustriert die Verwendung von `PMU_ALL_COUNTER`, `PMU_USERMODE_OVERALL_COUNTER`, `PMU_TASKTIME_COUNTER` und `PMU_TASKTIME_COUNTER_NO_IDLE`. Dabei wurde der Idle-Task vom `PMU_TASKTIME_COUNTER_NO_IDLE` auf den Task 2 (0x9228) konfiguriert, da der *richtige* Idle-Task des Betriebssystems in diesem Konstrukt sonst nicht ausgeführt werden würde.
Am Messausgang der PMU wird der Konfigurationszeitpunkt in `cnt`, ein Signal, das den Taskwechsel darstellt (`tas`), der aktuelle Privilege Mode (`prv`) sowie 13 Bit des aktuellen TP in `tp` herausgeführt.
Im Anschluss werden die einzelnen Zähler getrennt beleuchtet.

[2] http://gtkwave.sourceforge.net/

Die Simulation des PMU_ALL_COUNTER ist in Abbildung 6.4 ersichtlich. Dieser Zähler soll ja ab dem Konfigurationszeitpunkt dauernd zählen. Als Messzeitpunkt wird hier der erste Taskwechsel gewählt.

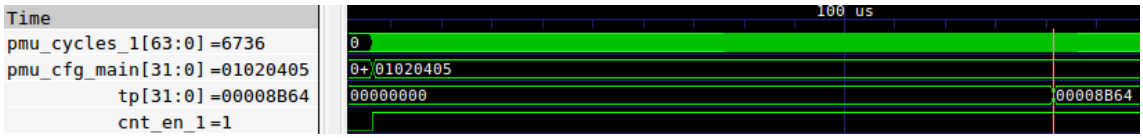


Abbildung 6.4: Die Darstellung des Testfalls PMU_ALL_COUNTER in der Simulation in GTKWave.

Der aus der Simulation ermittelte Laufzeitwert von 6736 entspricht nach (6.1) 134,72μs. Die korrespondierende Laufzeitmessung mit dem Oszilloskop ist in Abbildung 6.5 zu sehen. Der Messzeitraum ist – wie auch in allen weiteren Messungen – zwischen den gestrichelten Linien zu finden.

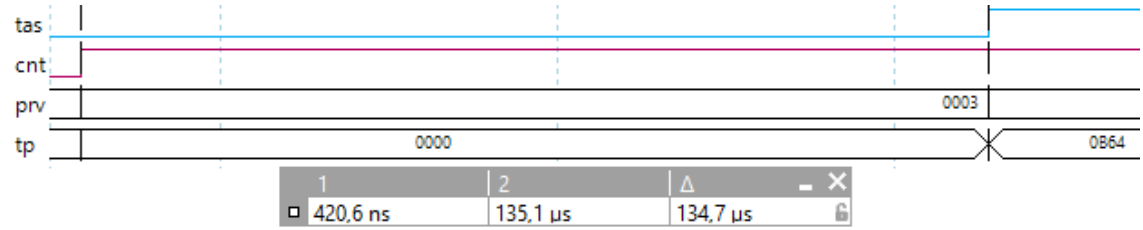


Abbildung 6.5: Die Messung des Testfalls PMU_ALL_COUNTER mit dem PicoScope.

Die Messung vom Start der Konfiguration bis zum ersten Taskwechsel ergibt 134,7μs, was sich sehr genau mit dem ermittelten Wert zuvor deckt. An dieser Stelle sei darauf hingewiesen, dass das PicoScope auf insgesamt vier Dezimalstellen misst.

In Abbildung 6.6 sieht man die Messung des PMU_USERMODE_OVERALL_COUNTER. Dieser misst genau dann, wenn der Privilege Mode der CPU dem Wert 0 entspricht.

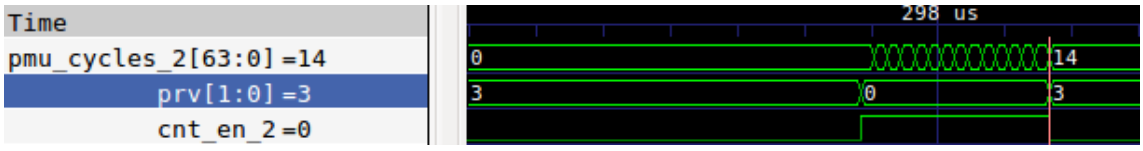


Abbildung 6.6: Die Darstellung des Testfalls PMU_USERMODE_OVERALL_COUNTER in der Simulation in GTKWave.

Die ersichtliche Laufzeit von 14 Zyklen entspricht 280ns. Die Zeitmessung ist in Abbildung 6.7 ersichtlich. Hier wird die Zeit gemessen, in der prv den Wert 0 hat.

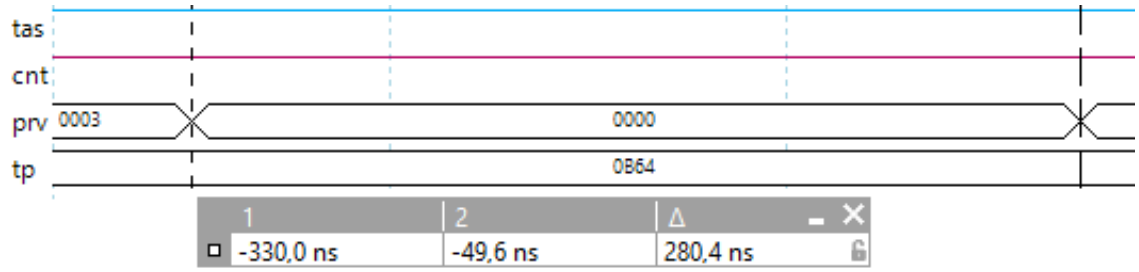


Abbildung 6.7: Die Messung des Testfalls `PMU_USERMODE_OVERALL_COUNTER` mit dem PicoScope.

Der Messwert beträgt hierbei 280,4ns.

Die Ausgabe der Simulation des `PMU_TASKTIME_COUNTER` ist in Abbildung 6.8 zu sehen. Dieser Zähler zählt die Laufzeit aller Tasks, demnach muss ein Messzeitpunkt gewählt werden. Hier bietet sich wieder ein Taskwechsel an.

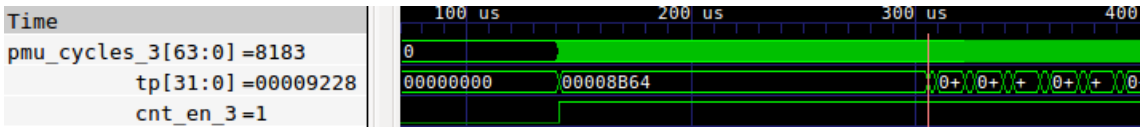


Abbildung 6.8: Die Darstellung des Testfalls `PMU_TASKTIME_COUNTER` in der Simulation in GTKWave.

Das Hardwaremodul ermittelt hier einen Wert von 8183, was einer Zeit von 163,66μs entspricht. Die zugehörige Zeitmessung ist in Abbildung 6.9 dargestellt.

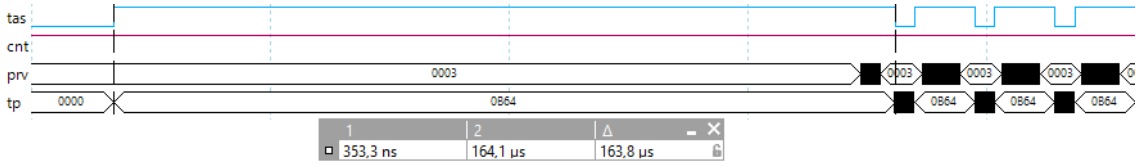


Abbildung 6.9: Die Messung des Testfalls `PMU_TASKTIME_COUNTER` mit dem PicoScope.

Die Messung ergibt hier einen Zeitwert von 163,8μs und bestätigt somit den ermittelten Wert.

Der PMU_TASKTIME_COUNTER_NO_IDLE wird in Abbildung 6.10 simuliert. Hierbei wird der PMU der Task 0x9228 als Idle-Task vorgetäuscht, was natürlich auch mit jedem anderen Task funktionieren würde.

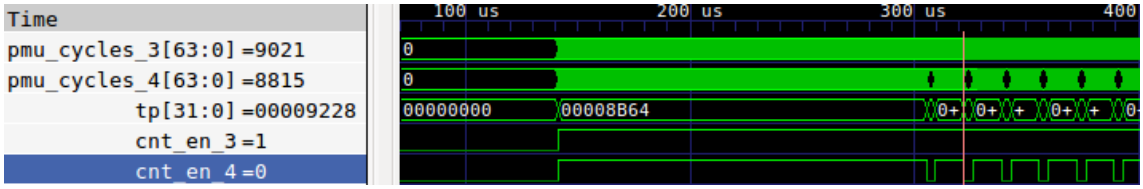


Abbildung 6.10: Die Darstellung des Testfalls PMU_TASKTIME_COUNTER_NO_IDLE in der Simulation in GTKWave.

Hier ermittelt die PMU einen Wert von 8815, was einem errechneten Zeitwert von 176,3μs entspricht. Zum selben Zeitpunkt ermittelt der PMU_TASKTIME_COUNTER einen Wert von 9021, was 180,42μs entspricht. Demnach lief der konfigurierte Idle-Task in diesem Zeitraum 4,12μs nicht. Die Überprüfung dieser Werte ist in Abbildung 6.11 gezeigt.

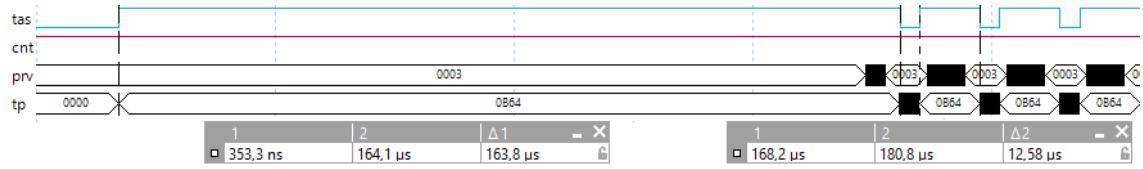


Abbildung 6.11: Die Messung des Testfalls PMU_TASKTIME_COUNTER_NO_IDLE mit dem PicoScope.

Hier ergeben die Messungen 163,8μs + 12,58μs = 176,38μs. Für die gesamte Laufzeit wird ein Wert von 180,4μs gemessen.

t_task.c

In diesem Test wird die Verwendung der Task-Zähler PMU_SINGLE_TASK_COUNTER, PMU_TASK_OVERALL_COUNTER und PMU_TASK_PART_COUNTER gezeigt.
Der PMU_SINGLE_TASK_COUNTER wird dabei auf Task 2 (0x9228) konfiguriert. Um auf vier Zähler zu kommen, wird zusätzlich ein PMU_ALL_COUNTER eingefügt.
Die PMU_TASK_PART_COUNTER sollen dabei den Bereich zwischen den GPIO_OUT-Zuweisungen der beiden Tasks messen. Die PC-Bereiche betragen dabei 0x358 bis 0x388 für den ersten und 0x3C8 bis 0x418 für den zweiten Task. Diese Konfigurationen müssen zu Beginn der jeweiligen Tasks geschehen.

Am Messausgang wird ein Signal für den Taskwechsel (`tas`), zwei Signale für die PC-Bereiche (`pc1` und `pc2`) und 13 Bit des aktuellen TP als `tp` angelegt.

Abbildung 6.12 zeigt die Ausgabe der Simulation des `PMU_SINGLE_TASK_COUNTER`. Als Messzeitpunkt wird der Zeitpunkt gewählt, zu dem der Task 2 mit dem TP 0x9228 zum ersten Mal angehalten und ein anderer Task zum Laufen gebracht wird.

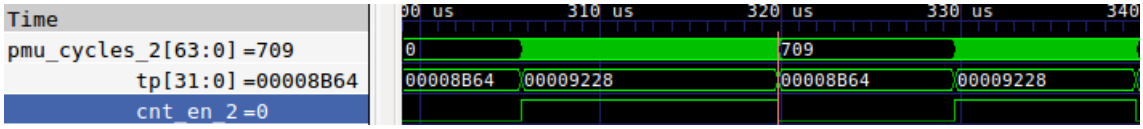


Abbildung 6.12: Die Darstellung des Testfalls `PMU_SINGLE_TASK_COUNTER` in der Simulation in GTKWave.

Die hier ermittelte Laufzeit von 709 entspricht 14,18μs. In Abbildung 6.13 ist die Kontrollmessung mit dem PicoScope dargestellt.

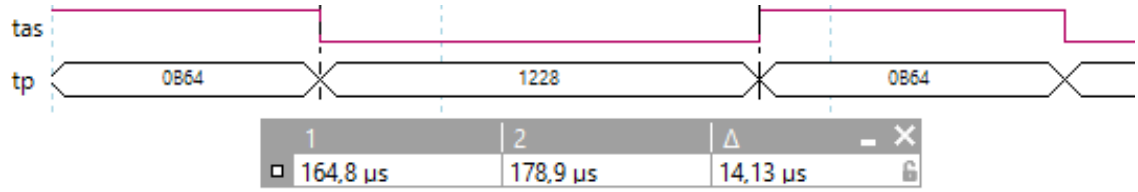


Abbildung 6.13: Die Messung des Testfalls `PMU_SINGLE_TASK_COUNTER` mit dem PicoScope.

Als Messwert ergibt sich für die gewählte Messzeitspanne ein Wert von 14,13μs.

Den `PMU_TASK_OVERALL_COUNTER` und seine Simulation zeigt Abbildung 6.14. Dabei wird nun auch kontrolliert, ob sich diese Zählerart beim Taskwechsel richtig kompensiert. Dazu sollte der vom Modul gemessene Wert immer noch mit dem Oszilloskopmesswert übereinstimmen. Gemessen wird vom Zeitpunkt des ersten Starts bis zur ersten Pausierung des Tasks 0x8B64.

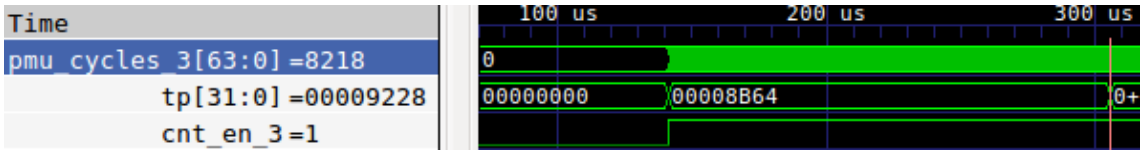


Abbildung 6.14: Die Darstellung des Testfalls `PMU_TASK_OVERALL_COUNTER` in der Simulation in GTKWave.

Die PMU ermittelt in dieser Messung einen Wert von 8218, was einem Zeitwert von 164,36μs entspricht. Dabei sollte die Kompensation bereits beinhaltet sein. Die Messung in Abbildung 6.15 zeigt dies nun.

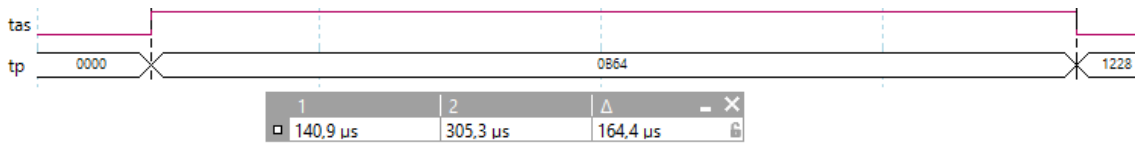


Abbildung 6.15: Die Messung des Testfalls `PMU_TASK_OVERALL_COUNTER` mit dem PicoScope.

Gemessen wird für den korrespondierenden Zeitraum eine Zeitspanne von 164,4μs. Demnach arbeitet die Kompensation richtig.

Die folgenden Abbildung 6.16 dient zur Validierung des `PMU_TASK_PART_COUNTER` für den Task 1 (0x8B64). Gemessen wird ab dem im Diagramm ersichtlichen Zeitpunkt.

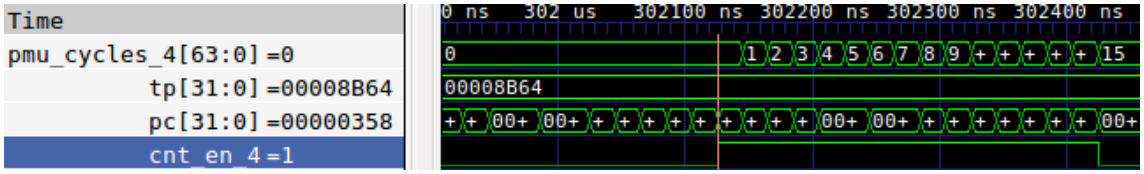


Abbildung 6.16: Die Darstellung des Testfalls 1 `PMU_TASK_PART_COUNTER` in der Simulation in GTKWave.

Die PMU schließt ihre Messung mit dem Wert 15 ab (rechts oben im Bild), was 300ns entspricht. Das Signal `pc1` liefert ein Hilfssignal, welches die Messung eines PC-Bereichs erleichtert. Die zugehörige Messung zeigt Abbildung 6.17.

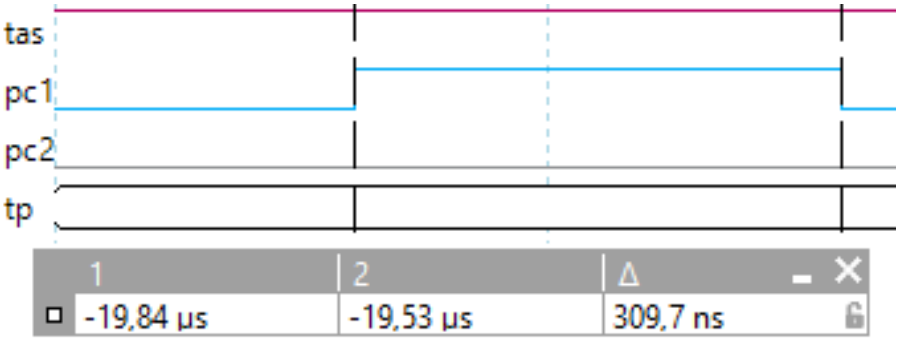


Abbildung 6.17: Die Messung des Testfalls 1 `PMU_TASK_PART_COUNTER` mit dem PicoScope.

Das Messergebnis ist hierbei 309,7ns. Die große Abweichung lässt sich erklären, da das Oszilloskop in dem zeitlichen Messbereich etwas ungenau misst. Bei 50 MHz ist die Periodendauer 20ns, wodurch man diese Abweichung kleiner als eine Zykluszeit ist.

Die Simulation der gleichen Messung für den zweiten Task (0x9228) ist in Abbildung 6.18 illustriert.

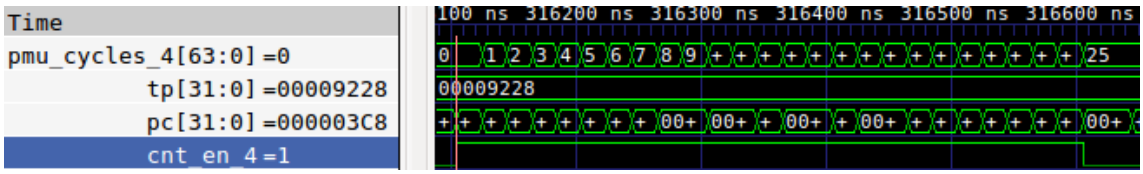


Abbildung 6.18: Die Darstellung des Testfalls 2 `PMU_TASK_PART_COUNTER` in der Simulation in GTKWave.

Als Ergebnis werden 25 Zyklen ermittelt, was einer Zeitspanne von 500ns entspricht. Validiert wird das Ganze in Abbildung 6.19.

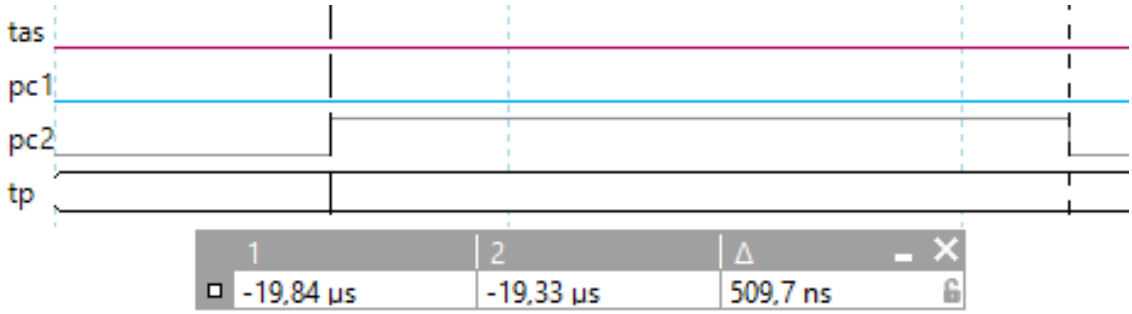


Abbildung 6.19: Die Messung des Testfalls 2 `PMU_TASK_PART_COUNTER` mit dem PicoScope.

Der Messwert ist in diesem Fall 509,7ns, wobei sich die Abweichung wie im vorigen Fall erklären lässt.

t_interrupt.c

Als nächstes wird die Verwendung der drei unterschiedlichen Interrupt-Zähler `PMU_INTERRUPT_OVERALL_COUNTER`, `PMU_INTERRUPT_MISSED` und `PMU_TASK_INTERRUPT_COUNTER` gezeigt. Da es sich hier nur um Ereigniszähler und nicht um Zeitzähler handelt, wird auf eine Darstellung eines Oszilloskopmesswertes verzichtet. Die Ausgangsdaten wurden jedoch trotzdem mit diesem validiert und entsprechen, sofern nicht anders angegeben, den Simulationswerten.
Die gewählte Maske ist jedes Mal 0x80, was einem Software-Interrupt entspricht. Der konfigurierte Task für den `PMU_INTERRUPT_MISSED` ist Task 1 (0x8B64). Dieser zählt also die Anzahl an Interrupts, während dieser Task nicht läuft, also welche er nicht mitbekommt.

Die gesamte Simulation aller drei Zähler ist in Abbildung 6.20 zu sehen.

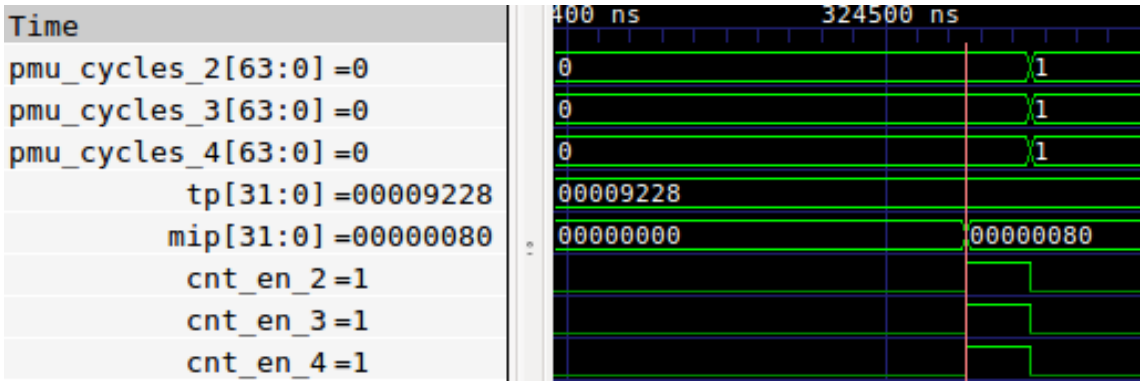


Abbildung 6.20: Die Darstellung der Testfälle `PMU_INTERRUPT_OVERALL_COUNTER`, `PMU_INTERRUPT_MISSED` und `PMU_TASK_INTERRUPT_COUNTER` in der Simulation in GTKWave.

Da alle drei Zähler auf dieselbe Maske konfiguriert wurden, zählen alle genau das eine Auftreten des Interrupts 0x80. Die gemessene Anzahl an Interrupts ist damit in jedem Fall 1.

t_ext.c

Dieser Test zeigt die Verwendung des externen Ports der PMU als Ereigniseingang. Verwendet werden dabei die beiden Zählerarten `PMU_EXT_MATCH_OVERALL` und `PMU_TASK_EXT_MATCH`. Im Zuge dieser Messung werden die GPIO-Ports des Systems direkt mit dem externen Port der PMU verbunden. Somit kann man in Software Werte auf diesen Eingang legen und muss diese nicht anderweitig generieren. Der erste Zähler wird auf ein Signal von 0xA, der zweite Zähler auf 0xB konfiguriert. Gezählt werden soll also, wie oft der externe Port seinen Wert auf diese Werte ändert.
Hier wird wie zuvor auf eine Oszilloskopdarstellung verzichtet.

Abbildung 6.21 veranschaulicht diesen Sachverhalt in einem Diagramm.

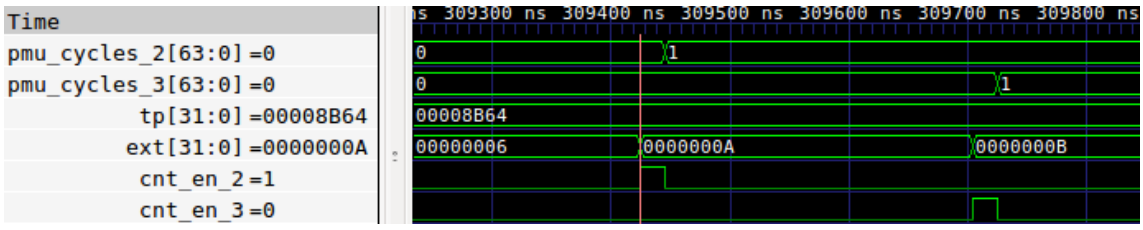


Abbildung 6.21: Die Darstellung der Testfälle `PMU_EXT_MATCH_OVERALL` und `PMU_TASK_EXT_MATCH` in der Simulation in GTKWave.

Wie zu erkennen ist, treten beide unterschiedlichen Werte im gezeigten Zeitraum je einmal auf, was sich auch in den Werten der Zähler niederschlägt.

t_taskcombined.c

Abschließend wird noch gezeigt, wie sich die globalen Messungen im Vergleich zu taskabhängigen Messungen verhalten. Dabei wird ein `PMU_TASKTIME_COUNTER`, zwei auf die beiden Tasks konfigurierte `PMU_SINGLE_TASK_COUNTER` zur globalen Messung der Tasks sowie ein `PMU_TASK_OVERALL_COUNTER` zur taskabhängigen Messung verwendet.
Es soll nun gezeigt werden, dass die globalen Zähler die gleichen Werte liefern, wie der taskabhängige Zähler. Hier soll ersichtlich gemacht werden, dass man sich mit diesem Hardwaremodul eine Vielzahl an einzelnen globalen Zählern sparen kann, indem man auf das Konzept der taskabhängigen Zähler setzt.
Der Messausgang der PMU wird hierbei mit einem Signal zum Taskwechsel (`tas`) sowie 15 Bit des aktuellen TP als `tp` beschalten, um Messungen mit dem Oszilloskop durchführen zu können.

In Abbildung 6.23 sieht man die Messung für den Task 2 (0x9228) und in Abbildung 6.22 ist die gleiche Messung für den Task 1 (0x8B64) dargestellt.

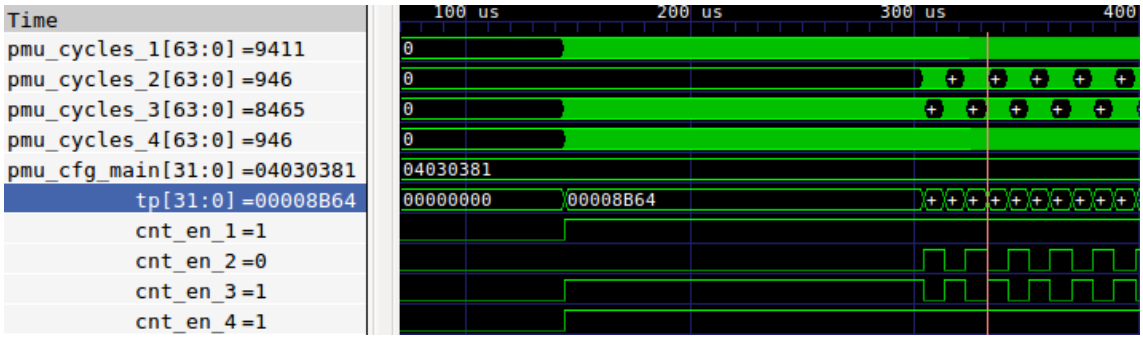


Abbildung 6.22: Die Darstellung des Testfalls 1 taskcombined in der Simulation in GTKWave.

Der erste Zähler (für die gesamte Tasklaufzeit, `_1`) ergibt einen Wert von 9411, was 188,22μs entspricht. Der erste globale Taskzähler (`_2`) ermittelt 946, was eine Zeit von 18,92μs bedeutet. Der taskabhängige Zähler (`_4`) liefert zudem das gleiche Ergebnis.
Sieht man sich die globalen Taskzähler (`_2` und `_3`) und den Zähler für die gesamte Taskzeit (`_1`) an, erkennt man, dass sich der Wert des letzteren aus der Addition der beiden vorherigen ergibt. Demnach kann das Ergebnis als richtig angesehen werden.

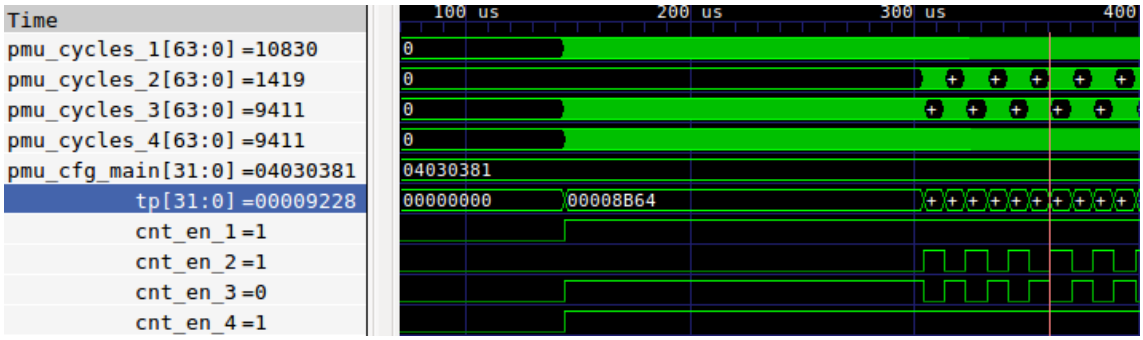


Abbildung 6.23: Die Darstellung des Testfalls 2 taskcombined in der Simulation in GTKWave.

Hier ist das Ergebnis für den `PMU_TASKTIME_COUNTER` 10830, was 216,6μs entspricht. Der globale Taskzähler für diesen Task liefert genau wie der taskabhängie Zähler einen Wert von 9411, was sich in einen Zeitwert von 188,22μs errechnen lässt.
Die Addition der beiden globalen Taskzähler zu diesem Zeitpunkt ergibt wieder die gesamte Tasklaufzeit.

Die oben ermittelten Werte werden nun durch Messung mit dem Oszilloskop validiert. Die zugehörige Messung ist in Abbildung 6.24 zu sehen.

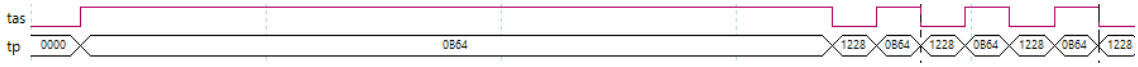


Abbildung 6.24: Die Messung des Testfalls taskcombined mit dem PicoScope.

Hier müssen nun die korrespondierenden TP-Phasen einzeln gemessen und richtig zusammenaddiert werden.
Die Messung für die Simulation in Abbildung 6.22 ergeben demnach 188,6μs für die gesamte Tasklaufzeit und 19,099μs für die Tasklaufzeit.
Für die Simulation in Abbildung 6.23 werden die Werte 216,8μs für die gesamte Tasklaufzeit und 188,614μs für die Tasklaufzeit beobachtet.

Abschließend kann man aus der Sammlung an Messwerten und deren korrespondierenden Messungen der PMU schließen, dass das Hardwaremodul die Laufzeiten sehr genau ermitteln kann und keine Taktzyklen ausgelassen werden. Gewisse Abweichungen lassen sich auf Messungenauigkeiten seitens des Oszilloskops und auch der PC-Software zurückführen. Außerdem erweisen sich diese Ungenauigkeiten als ausreichend klein um vernachlässigt werden zu können.

Kapitel 7

Vergleiche und Ausblick

Abschließend werden in diesem Kapitel noch Vergleiche mit anderen Arbeiten gezogen sowie ein Ausblick auf mögliche zukünftige Entwicklungen gegeben.

7.1 Vergleiche mit bestehenden Lösungen

Die hier vorliegende Arbeit unterscheidet sich in zwei Punkten maßgeblich von bislang vorgelegten Systemen:

- Die PMU wurde so konzipiert, dass sie sich in das Gesamtsystem von Hard- und Software einfügt und dieses versteht und
- sie setzt auf Wiederverwendung der Zählelemente um so den Hardwarebedarf minimal zu halten.

Weiters kann sie durch ihre Konfigurierbarkeit einfach auf andere Systeme übertragen werden.

7.2 Ausblick und zukünftige Entwicklungen

Einige interessante Features wurden in dieser Arbeit noch nicht berücksichtigt. Diese beinhalten unter anderem

- sequentielle Zähler, also Zähler, die durch ein Ereignis gestartet und durch ein anderes beendet werden und
- Selbstoptimierung des Speicherauslagerungsprozesses hinsichtlich der benötigten Laufzeit.

Des weiteren würde eine Integration in ein Multicore-System neue Erkenntnisse zur Optimierung bringen können.

7.3 Zusammenfassung

Die hier vorliegende Arbeit beschreibt die entwickelte Performance Monitoring Unit für die RISC-V-Architektur. Sie ist dabei voll in das Gesamtsystem aus Soft- und Hardware integriert und macht sich dessen Eigenheiten zu nutze. Dabei ist insbesondere die Task-Awareness hervorzuheben, welche es ermöglicht, taskabhängige Messungen von Laufzeiten und Ereignissen durchzuführen.
Weiters setzt das entwickelte Modul auf volle Skalierbarkeit im Bezug auf die Zähleinheiten sowie deren zugehörige Konfigurationsregister.

Eine Neuheit stellt die taskabhängige Wiederverwendung von Hardwarezählern mittels Zwischenspeicherung in einem RAM dar. Dabei werden die aktuellen Stände bestimmter Zählerarten beim Wechsel eines Tasks auf den Speicher ausgelagert und beim Fortsetzen wieder eingelagert. Aufgrund dieser Tatsache kann eine Hardwareeinheit von einer Vielzahl an Tasks verwendet werden, ohne dass es zu Messfehlern kommt.

Abschließend sei gesagt, dass die Arbeit eine neue Möglichkeit aufzeigt, wie man Tasks in einem eingebetteten System – hier auf RISC-V basiert – überwachen kann, ohne das System an sich zu beeinflussen. Dabei wird insbesondere darauf geachtet, wenige Hardwareressourcen zu benötigen sowie diese möglichst effizient nutzen zu können.

Appendix

A Konfigurationsregister

Registerübersicht

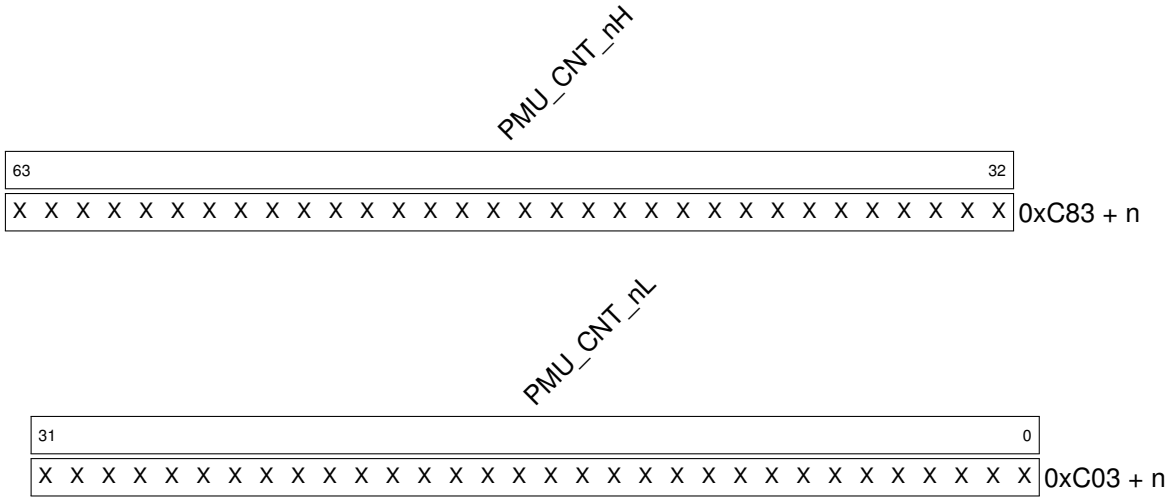
In diesem Kapitel werden die Konfigurations- und Zählregister für eine Performance Monitoring Unit (PMU) mit 4 Zählern zu je 64 Bit mit jeweils 2 Konfigurationsbytes beschrieben. Insgesamt kommt man also auf 17 32-Bit-Register:

- 8 Zählregister PMUCNTnH/L: 2 Stück 32-Bit-Register pro Zähler
- 1 Hauptkonfigurationsregister PMUCFGMAIN (4 Zähler, 1 Byte pro Zähler)
- 8 Konfigurationsregister PMUCFGnm: 2 Konfigurationsregister je Zähler

Des Weiteren werden die zugehörigen Control and Status Register (CSR)-Adressen angegeben.

Registerbeschreibung

PMUCNTnH und PMUCNTnL (0xC83 + n und 0xC03 + n)

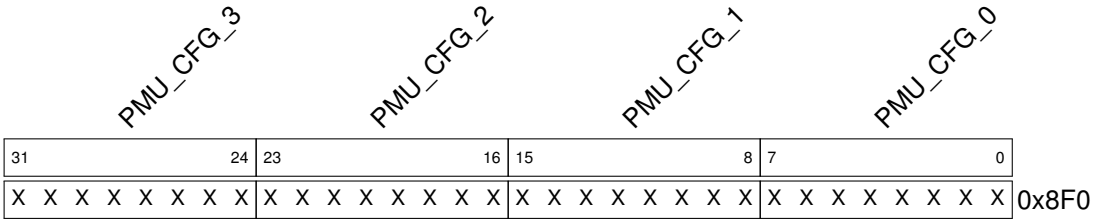


PMU_CNT_nH High-Byte des Zählers mit dem Index n.

PMU_CNT_nL Low-Byte des Zählers mit dem Index n.

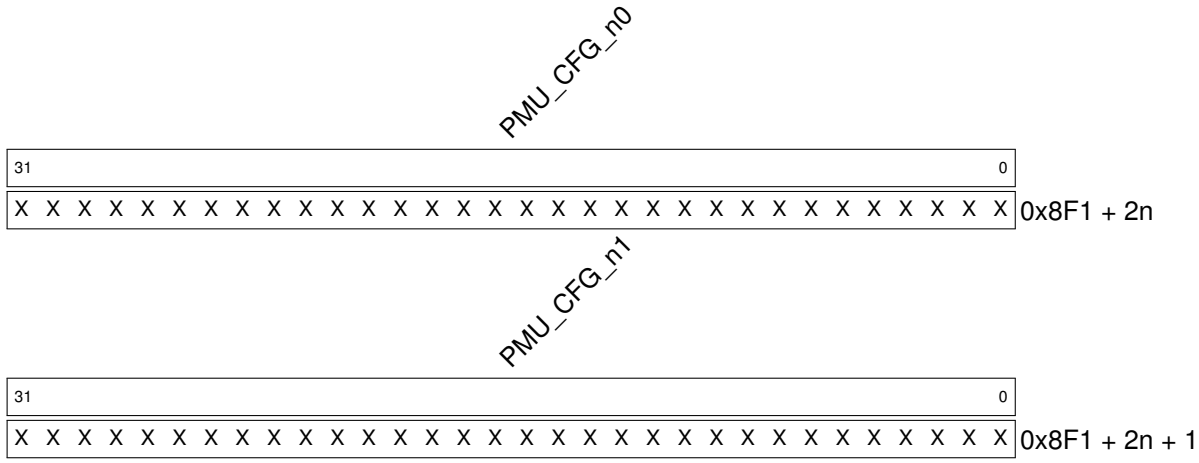
Als Registeradressen können auch die Alias-Bezeichnungen `hpmcounter3`, `hpmcounter4`, ... und `hpmcounter3h`, `hpmcounter4h`, ... im Assemblercode verwendet werden.

PMUCFGMAIN (0x8F0)



PMU_CFG_n	Konfiguriert die Art des Zählers mit dem Index n. Gültige Werte:
0x00	PMU_RESET
0x01	PMU_ALL_COUNTER
0x02	PMU_USERMODE_OVERALL_COUNTER
0x03	PMU_SINGLE_TASK_COUNTER
0x04	PMU_TASKTIME_COUNTER
0x05	PMU_TASKTIME_COUNTER_NO_IDLE
0x06	PMU_INTERRUPT_OVERALL_COUNTER
0x07	PMU_INTERRUPT_MISSED
0x08	PMU_EXT_MATCH_OVERALL
0x81	PMU_TASK_OVERALL_COUNTER
0x82	PMU_TASK_INTERRUPT_COUNTER
0x83	PMU_TASK_EXT_MATCH
0xC2	PMU_TASK_PART_COUNTER

PMUCFGnm (0x8F1 + 2n + m)



PMU_CFG_n0 Konfigurationsregister 0 des Zählers mit dem Index n.

PMU_CFG_n1 Konfigurationsregister 1 des Zählers mit dem Index n.

Verwendung bei folgenden Zählern:

Zählerart	*PMU_CFG_n0*	*PMU_CFG_n1*
PMU_SINGLE_TASK_COUNTER	Taskpointer	-
PMU_TASKTIME_COUNTER_NO_IDLE	Taskpointer Idle-Task	-
PMU_INTERRUPT_OVERALL_COUNTER	Interrupt-Maske	-
PMU_INTERRUPT_MISSED	Interrupt-Maske	Taskpointer
PMU_EXT_MATCH_OVERALL	Vergleichswert	-
PMU_TASK_INTERRUPT_COUNTER	Interrupt-Maske	-
PMU_TASK_EXT_MATCH	Vergleichswert	-
PMU_TASK_PART_COUNTER	PC Startwert	PC Endwert

Tabelle A.1: Verwendung der PMU_CFG_n0 und PMU_CFG_n1 für die verschiedenen Zählerarten.

B Verilog

Next-State-Logik

```
1  reg [`XPR_LEN-1:0]  osmem_wdata;
2  reg [`XPR_LEN-1:0]  state;
3  reg [`XPR_LEN-1:0]  state_next;
4  reg [`XPR_LEN-1:0]  index;
5  reg [`XPR_LEN-1:0]  index_next;
6  localparam  PMU_IDLE=0,     PMU_SWAP_OUT_CFG=1,
7        PMU_SWAP_OUT_CNT_L=2, PMU_SWAP_OUT_CNT_H=3,
8        PMU_SWAP_IN_CFG=4,    PMU_SWAP_IN_CNT_L=5,
9        PMU_SWAP_IN_CNT_H=6;
10 always @(*) begin
11   osmem_en = 0;
12   osmem_wen = 0;
13   osmem_size = 2'b10;
14   osmem_addr = `XPR_LEN'h0;
15   osmem_wdata = `XPR_LEN'h0;
16   state_next = state; index_next = index;
17   case(state)
18     PMU_IDLE: begin
19         if(tp_changed && tp != 0) begin
20           osmem_write(tp_old + `OS_OFFSET_PMU_CFG,
21                                pmu_cfg[index]);
22           state_next = PMU_SWAP_OUT_CFG;
23           index_next = index + 1;
24         end
25       end
26     PMU_SWAP_OUT_CFG: begin
27         osmem_write(tp_old + `OS_OFFSET_PMU_CFG + (4*index),
28                                pmu_cfg[index]);
29         if(index != (`CSR_PMU_N * `CSR_PMU_CFG_N +
30                                `CSR_PMU_N/4 - 1)) begin
31           index_next = index + 1;
32         end else begin
33           index_next = 0;
34           state_next = PMU_SWAP_OUT_CNT_L;
35         end
36       end
37     PMU_SWAP_OUT_CNT_L: begin
38         osmem_write(tp_old + `OS_OFFSET_PMU_CNT + (8*index),
39                                pmu_cycles[index][0+:`XPR_LEN]);
40         if(index != (`CSR_PMU_N-1)) begin
41           index_next = index + 1;
```

```
42          end else begin
43            index_next = 0;
44            state_next = PMU_SWAP_OUT_CNT_H;
45          end
46        end
47      PMU_SWAP_OUT_CNT_H: begin
48          osmem_write(tp_old + `OS_OFFSET_PMU_CNT + (8*index)+4,
49                                pmu_cycles[index][`XPR_LEN+:`XPR_LEN]);
50          if(index != (`CSR_PMU_N-1)) begin
51            index_next = index + 1;
52          end else begin
53            index_next = 0;
54            state_next = PMU_SWAP_IN_CFG;
55          end
56        end
57      PMU_SWAP_IN_CFG: begin
58          osmem_read(tp + `OS_OFFSET_PMU_CFG + (4*index));
59          if(index != (`CSR_PMU_N * `CSR_PMU_CFG_N +
60                                  `CSR_PMU_N/4)) begin
61            index_next = index + 1;
62          end else begin
63            index_next = 0;
64            state_next = PMU_SWAP_IN_CNT_L;
65          end
66        end
67      PMU_SWAP_IN_CNT_L: begin
68          osmem_read(tp + `OS_OFFSET_PMU_CNT + (8*index));
69          if(index != (`CSR_PMU_N)) begin
70            index_next = index + 1;
71          end else begin
72            index_next = 0;
73            state_next = PMU_SWAP_IN_CNT_H;
74          end
75        end
76      PMU_SWAP_IN_CNT_H: begin
77          osmem_read(tp + `OS_OFFSET_PMU_CNT + (8*index)+4);
78          if(index != (`CSR_PMU_N)) begin
79            index_next = index + 1;
80          end else begin
81            index_next = 0;
82            state_next = PMU_IDLE;
83          end
84        end
85    endcase
86  end
```

C Syscalls und C-Funktionen

Makros und Konstanten

Die Konstanten für die Zählerarten sind gleich wie in der Hardware definiert (siehe Anhang A).
Zum Generieren eines 32-Bit-Wertes für ein Hauptkonfigurationsregister kann das Makro `CFG(..)` verwendet werden:

```
1 #define CFG(a, b, c, d) ((((a) & 0xFF) << 24) |
2                          (((b) & 0xFF) << 16) |
3                          (((c) & 0xFF) <<  8) |
4                           ((d) & 0xFF))
```

Für `a`, `b`, `c` und `d` können dann die Konstanten `PMU_*` verwendet werden.

Syscalls

`void pmu_cfg_main(uint32_t cfg)`

Setzt das Hauptkonfigurationsregister.

`cfg` 32-Bit-Wert aller vier 8-Bit-Zählerarten

`void pmu_cfg(uint32_t offset, uint32_t cfg_1, uint32_t cfg_2)`

Setzt die zwei Konfigurationsregister eines Zählers.

`index` Index des Zählers

`cfg_1` Erstes Konfigurationsregister des indizierten Zählers

`cfg_2` Zweites Konfigurationsregister des indizierten Zählers

`os_time_t pmu_get_cycles(uint32_t index)`

Liest den aktuellen Wert eines Zählers aus.

`index`	Index des Zählers
`return`	Aktueller Zählerstand als 64-Bit-Wert

`uint32_t* pmu_get_task_cycles(os_id_t task)`

Liefert die gespeicherten Zählerstände eines Tasks.

`task`	Task Pointer des gewünschten Tasks
`return`	Pointer auf ein Array aus 32-Bit-Zählerstandswerten (wie im Speicher abgelegt)

D Testfälle

```
1  // t_overall.c
2  #include <mosart_os.h>
3  #include <pmu.h>
4
5  int task1(void);
6  int task2(void);
7  os_resource_t r;
8
9  void main(void)
10 {
11     pmu_cfg_main(PMU_RESET);
12     pmu_cfg(3, 0x9228, 0x0);
13     pmu_cfg_main(CFG(PMU_ALL_COUNTER,
14                      PMU_USERMODE_OVERALL_COUNTER,
15                      PMU_TASKTIME_COUNTER,
16                      PMU_TASKTIME_COUNTER_NO_IDLE));
17
18     os_register_task(task1, 16*80, 100);
19     os_register_task(task2, 16*80, 10);
20     os_run();
21
22     while(1);
23 }
24 int task1(void)
25 {
26     while(1)
27     {
28         get_resource_until(&r, DEADLINE_INFINITE);
29         sleep(5);
30         release_resource(&r);
31     }
32     return 0;
33 }
34 int task2(void)
35 {
36     while(1)
37     {
38         get_resource_until(&r, DEADLINE_INFINITE);
39         release_resource(&r);
40     }
41     return 0;
42 }
```

```c
// t_task.c
#include <mosart_os.h>
#include <pmu.h>
#define GPIO_OUT *((volatile uint32_t *)0x80001004ul)
int task1(void);
int task2(void);

void main(void)
{
    pmu_cfg_main(PMU_RESET);
    pmu_cfg(1, 0x9228, 0x0);
    pmu_cfg_main(CFG(PMU_ALL_COUNTER, PMU_SINGLE_TASK_COUNTER,
             PMU_TASK_OVERALL_COUNTER, PMU_TASK_PART_COUNTER));
    os_register_task(task1, 16*80, 100);
    os_register_task(task2, 16*80, 100);
    os_run();
    while(1);
}
int task1(void)
{
    pmu_cfg(3, 0x358, 0x388);
    while(1)
    {
        GPIO_OUT = 0xA;
        uint32_t a, b, c;
        a = 1; b = 2; c = a + b;
        GPIO_OUT = 0xB;
        sleep(10);
    }
    return 0;
}
int task2(void)
{
    pmu_cfg(3, 0x3C8, 0x418);
    while(1)
    {
        GPIO_OUT = 0xC;
        uint32_t a, b, c, d, e;
        a = 6; b = 1; c = 12; d = 3;
        e = a - b + c - d;
        GPIO_OUT = 0xD;
        sleep(10);
    }
    return 0;
}
```

```c
// t_interrupt.c
#include <mosart_os.h>
#include <pmu.h>
#define GPIO_OUT *((volatile uint32_t *)0x80001004ul)
int task1(void);
int task2(void);
os_resource_t r;

void main(void)
{
    pmu_cfg_main(PMU_RESET);
    pmu_cfg(1, 0x80, 0x0);
    pmu_cfg(2, 0x80, 0x8B64);
    pmu_cfg(3, 0x80, 0x0);
    pmu_cfg_main(CFG(PMU_ALL_COUNTER,
        PMU_INTERRUPT_OVERALL_COUNTER, PMU_INTERRUPT_MISSED,
        PMU_TASK_INTERRUPT_COUNTER));
    os_register_task(task1, 16*80, 100);
    os_register_task(task2, 16*80, 10);
    os_run();
    while(1);
}
int task1(void)
{
    sleep(1000);
    while(1)
    {
        GPIO_OUT |= 0x80;
        get_resource_until(&r, DEADLINE_INFINITE);
        sleep(5);
        release_resource(&r);
    }
    return 0;
}
int task2(void)
{
    while(1)
    {
        GPIO_OUT |= 0x40;
        get_resource_until(&r, DEADLINE_INFINITE);
        release_resource(&r);
    }
    return 0;
}
```

```c
// t_ext.c
#include <mosart_os.h>
#include <pmu.h>
#define GPIO_OUT *((volatile uint32_t *)0x80001004ul)
int task1(void);
int task2(void);
void main(void)
{
    pmu_cfg_main(PMU_RESET);
    pmu_cfg(0, 0x0, 0x0);
    pmu_cfg(1, 0xA, 0x0);
    pmu_cfg(2, 0xB, 0x0);
    pmu_cfg(3, 0x0, 0x0);
    pmu_cfg_main(CFG(PMU_ALL_COUNTER, PMU_EXT_MATCH_OVERALL,
            PMU_TASK_EXT_MATCH, PMU_RESET));
    os_register_task(task1, 16*80, 100);
    os_register_task(task2, 16*80, 10);
    os_run();
    while(1);
}
int task1(void)
{
    pmu_cfg(3, 0x358, 0x388);
    while(1)
    {
        GPIO_OUT = 0xA;
        uint32_t a, b, c;
        a = 1; b = 2; c = a + b;
        GPIO_OUT = 0xB;
        sleep(10);
    } return 0;
}
int task2(void)
{
    pmu_cfg(3, 0x3C8, 0x418);
    while(1)
    {
        GPIO_OUT = 0xC;
        uint32_t a, b, c, d, e;
        a = 6; b = 1; c = 12; d = 3; e = a - b + c - d;
        GPIO_OUT = 0xD;
        sleep(10);
    } return 0;
}
```

```c
// t_taskcombined.c
#include <mosart_os.h>
#include <pmu.h>

int task1(void);
int task2(void);

void main(void)
{
    pmu_cfg_main(PMU_RESET);
    pmu_cfg(1, 0x9228, 0x0);
    pmu_cfg(2, 0x8B64, 0x0);
    pmu_cfg_main(CFG(PMU_TASKTIME_COUNTER,
            PMU_SINGLE_TASK_COUNTER, PMU_SINGLE_TASK_COUNTER,
            PMU_TASK_OVERALL_COUNTER));

    os_register_task(task1, 16*80, 100);
    os_register_task(task2, 16*80, 100);
    os_run();

    while(1);
}
int task1(void)
{
    while(1)
    {
        sleep(10);
    }
    return 0;
}
int task2(void)
{
    while(1)
    {
        sleep(20);
    }
    return 0;
}
```

Literaturverzeichnis

[1] Jude Angelo Ambrose, Vito Cassisi, Daniel Murphy, Tuo Li, Darshana Jayasinghe, and Sri Parameswaran. Scalable Performance Monitoring of Application Specific Multiprocessor Systems-on-Chip. *2013 IEEE 8th International Conference on Industrial and Information Systems*, August 2013.

[2] Nam Ho, Paul Kaufmann, and Marco Platzner. A hardware/software infrastructure for performance monitoring on LEON3 multicore platforms. *24th International Conference on Field Programmable Logic and Applications*, 2014.

[3] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A. Patterson, and Krste Asanovic. The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9.1. Technical Report UCB/EECS-2016-161, EECS Department, University of California, Berkeley, Nov 2016.

[4] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. Technical Report 253669-061US, Intel, December 2016.

[5] ARM. Cortex-A5 Technical Reference Manual. Technical report, ARM, 2016.

[6] VectorBlox. ORCA FPGA-Optimized RISC-V. http://riscv.org/wp-content/uploads/2016/01/Wed1200-2016-01-05-VectorBlox-ORCA-RISC-V-DEMO.pdf, 2016.

[7] Yunsup Lee, Albert Ou, and Albert Magyar. Z-scale: Tiny 32-bit RISC-V Systems. https://riscv.org/wp-content/uploads/2015/06/riscv-zscale-workshop-june2015.pdf, 2015.

[8] Andreas Traber, Florian Zaruba, Sven Stucki, Antonio Pullini, Germain Haugou, Eric Flamand, Frank K. Gürkayank, and Luca Benini. PULPino: A small single-core RISC-V SoC. http://iis-projects.ee.ethz.ch/images/d/d0/Pulpino_poster_riscv2015.pdf, 2015.

[9] Brinkley Sprunt. The Basics of Performance-Monitoring Hardware. *IEEE Micro (Volume: 22, Issue: 4, Jul/Aug 2002)*, 2002.

[10] Dipak Patil, Prashant Kharat, and Anil Kumar Gupta. Study of Performance Counters and Profiling Tools to Monitor Performance of Application. *21st IRF International Conference*, 2015.

[11] Aman Singh, Anup Buchke, and Yan-Hang Lee. A Study of Performance Monitoring Unit, perf and perf_events subsystem. -, 2012.

[12] Intel. Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide. Technical Report 253669-061US, Intel, December 2010.

[13] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1. Technical Report UCB/EECS-2016-118, EECS Department, University of California, Berkeley, May 2016.

[14] ARM Ltd. *ARM Architecture Reference Manual*, 2005.

[15] Hitachi. *SH-4 CPU Core Architecture*, 2002.

[16] Hewlett-Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1994.

[17] SPARC International, Inc. *The SPARC Architecture Manual Version 8*, 1992.

[18] Tony Chen and David A. Patterson. RISC-V Geneology. Technical Report UCB/EECS-2016-6, EECS Department, University of California, Berkeley, Jan 2016.

[19] Wade D. Peterson. *Wishbone B4, WISHBONE System-on-Chip (SoC)Interconnection Architecturefor Portable IP Cores*. Silicore Corp., 2010.

[20] ARM Ltd. *AMBA AXI Protocol v1.0*, 2004.

[21] Altera. *Avalon Interface Specifications*, 2015.

[22] Andreas Traber. RI5CY Core: Datasheet. Technical report, ETH Zürich, February 2016.

[23] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1216–1225, New York, NY, USA, 2012. ACM.

[24] ARM Ltd. *AMBA 3 AHB-Lite Protocol v1.0*, 2006.

[25] Florian Dittman and Stefan Frank. Hard real-time reconfiguration port scheduling. *Design, Automation and Test in Europe Conference and Exhibition*, April 2007.

[26] Fabian Mauroner and Marcel Carsten Baunach. Event based and Priority aware IRQ handling for Multi-Tasking Environments. unpublished, 2016.

[27] Xilinx. *7 Series FPGAs Memory Resources*. Xilinx, ug473 (v1.12) edition, September 2016.

[28] Pico Technology. *PicoScope 2205 MSO Mixed Signal Oscilloscope*, 2016.