ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

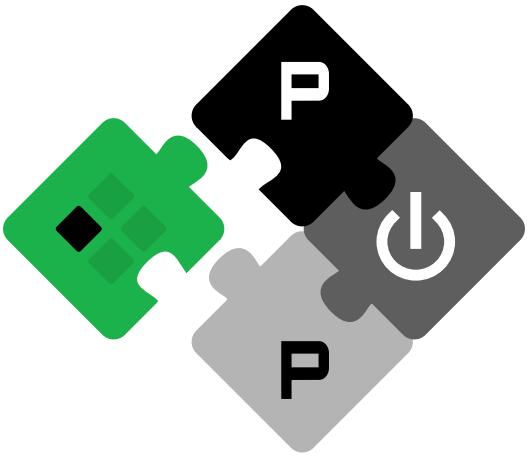
Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and
Electrical Engineering

Autumn Semester 2015

PULPino implementation in 65nm CMOS

Semester Project



Florian Zaruba
zarubaf@student.ethz.ch

December 2015

Supervisors: Frank K. Gürkaynak (DZ), kgf@ee.ethz.ch
Michael Gautschi (IIS), gautschi@iis.ee.ethz.ch
Professor: Prof. Dr. Luca Benini, lbenini@iis.ee.ethz.ch

Acknowledgements

This work would not have been possible without the great help of my supervisors Frank Gürkaynak and Michael Gautschi whom I want to thank for all the support they provided me. In addition I want to express my gratitude to Professor Luca Benini for always keeping track with my project and giving valuable input when I needed it. Last but not least I want to give my deep appreciativeness to the entire PULP group and especially to Andreas Traber, the "father of PULPino", who has been always more than willing to answer all my questions and has been a great source of help and advise when I needed it.

Abstract

PULPino is an open-source microcontroller like system, based on a small 32-bit RISC-V core that was developed at ETH Zurich. The core has an IPC close to 1, full support for the base integer instruction set (RV32I), compressed instructions (RV32C) and partial support for the multiplication instruction set extension (RV32M). It implements our non-standard extensions for hardware loops, post-incrementing load and store instructions, ALU and MAC operations. To allow embedded operating systems such as FreeRTOS to run, a subset of the privileged specification is supported. When the core is idle, the platform can be put into a low power mode, where only a simple event unit is active and wakes up the core in case an event/interrupt arrives.

The PULPino platform is available for RTL simulation, FPGA and as an ASIC in UMC 65nm (Imperio). It has full debug support on all targets. In addition we support extended profiling with source code annotated execution times through KCacheGrind in RTL simulations.

PULPino is based on IP blocks from the PULP project, the Parallel Ultra-Low-Power Processor that is developed as a collaboration between multiple universities in Europe, including the Swiss Federal Institute of Technology Zurich (ETHZ), University of Bologna, Politecnico di Milano, Swiss Federal Institute of Technology Lausanne (EPFL) and the Laboratory for Electronics and Information Technology of Atomic Energy and Alternative Energies Commission (CEA-LETI).

Declaration of Originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor. For a detailed version of the declaration of originality, please refer to Appendix B

Florian Zaruba,
Zurich, December 2015

Contents

I. PULPino **1**

1. Introduction **2**
1.1. Scope of this thesis . . . 3
1.2. General Overview . . . 3
1.3. Document Structure . . . 4
1.4. Related Work . . . 4
1.4.1. OpenRISC . . . 4
1.4.2. RISC-V . . . 5
1.4.3. Spain University . . . 5
1.4.4. lowRISC . . . 5

2. Preliminaries / Background **7**
2.1. Getting Started . . . 7

3. Theory / Algorithms **8**
3.1. FreeRTOS . . . 8
3.1.1. Purpose of an Operating System . . . 9
3.1.2. Directory structure . . . 10
3.1.3. Portable layer . . . 12
3.1.4. Context Switch . . . 12
3.1.5. Stack initialization . . . 15

II. Imperio **16**

4. Hardware Architecture **17**
4.1. Core . . . 17
4.1.1. OpenRISC - OR10N . . . 17
4.1.2. RISC-V - RI5CY . . . 19

4.2. AXI Interconnect . . . 19
4.2.1. AXI Slices . . . 20
4.2.2. SPI Slave . . . 20
4.2.3. Memory and Core adapters . . . 20
4.3. Advanced Debug Unit . . . 20
4.4. APB Peripherals . . . 21
4.4.1. Interrupt and Sleep Unit . . . 21
4.4.2. Timer Unit . . . 22
4.4.3. PULPino peripheral . . . 22
4.4.4. GPIO . . . 23
4.4.5. I2C . . . 23
4.4.6. UART (Universal Asynchronous Receiver Transmitter) . . . 24
4.4.7. SPI Master . . . 25
4.4.8. FLL Interface . . . 25
4.5. FLL . . . 25

5. Design Implementation and Results **26**
5.1. Implementation . . . 26
5.1.1. Multiplexed Pads . . . 26
5.1.2. Register file . . . 27
5.1.3. RAM banks . . . 28
5.2. Booting . . . 28
5.2.1. Booting from ROM . . . 28
5.2.2. Booting over SPI or JTAG . . . 29
5.3. Verification . . . 29
5.3.1. Functional . . . 29
5.3.2. Design for Testability (DFT) . . . 31
5.4. Timing . . . 33
5.5. Power . . . 33
5.6. Results . . . 34
5.6.1. Power . . . 34
5.7. Area . . . 35

6. Conclusion and Future Work **38**
6.1. Conclusion . . . 38
6.2. Future Work . . . 38

A. Task Description **40**

B. Declaration of Originality **45**

C. Datasets **47**
C.1. More Evaluation Results . . . 47
C.2. Algorithms / Tables . . . 47

C.3. Boot code 47

D. ASIC Datasheet (Imperio) 53
D.1. Features 53
D.2. Description 54
D.3. Packaging 54
D.4. Bonding Diagram 54
D.5. Pin Map 56
D.6. Pin Description 56
D.7. Floorplan 57
D.8. Pad Configuration 59
D.9. Interface Description 59
D.10. Register Map 59
D.11. Operation Modes 59
D.11.1. Functional Modes 59
D.11.2. Test Modes 59
D.12. Electrical Specifications 61
D.12.1. Recommended Operating Regions 61

E. PULPino environment 62
E.0.1. Requirements 62
E.0.2. Directory Structure 62
E.1. Getting started 64
E.1.1. Utilities 65
E.1.2. Imperio related CMake targets 66

F. Memory and Register Map 67
F.1. Memory Map 68
F.2. Register Map 69
F.2.1. UART 69
F.2.2. GPIO 69
F.2.3. SPI Master 69
F.2.4. Timer 69
F.2.5. Event/Interrupt Unit 69
F.2.6. I2C 69
F.2.7. FLL 70
F.2.8. SoC Control 70

List of Figures

3.1. FreeRTOS task state diagram. TODO: list resource 10
3.2. Preemptive and cooperative context switch 11
3.3. FreeRTOS memory management and stack content of a switched out task (Task 2) 12
3.4. FreeRTOS context switch 13

4.1. PULPino block diagram 18

5.1. Multiplexed Pads 27
5.2. Comparison of latch based and flip-flop based register file (test coverage vs. area) 28
5.3. Functional verification setup. 32
5.4. Bonding diagram 36
5.5. Area estimates (UMC65 - LVT 1.2V) @ 1.3ns - 1 GE = 1.44 µm^2 37

D.1. Bonding diagram. 55
D.2. Imperio pinout (QFN40) 56
D.3. Floorplan with macro block positioning 57

F.1. I2C Registers 69

List of Tables

5.1. Power Results (UMC65 - LVT 1.2V) @ 2ns, 25 °C 35

D.1. Pin Description. (Power, IO, Clock, Reset, Test) 58
D.2. Pad configuration and corresponding reset values. 60
D.3. DC characteristics 61
D.4. Recommended Operating Conditions 61

Part I.

PULPino

Chapter 1

Introduction

PULPino is a open-source 32-bit micro-controller like system based on IPs mostly taken from its bigger brother the Open Parallel Ultra-Low-Power Processing-Platform (PULP) project. It is a single core system built for the OpenRISC and RISC-V cores developed here at IIS. PULPino uses two seperate single-port data and instruction RAMs and also includes a boot ROM that contains a boot loader which retrieves data and instructions from an external SPI flash during startup. PULPino is entirely open source and licensed under Solderpad Hardware License.

In fact this project mostly origins from the idea to open-source the PULP project. Since PULP is a huge project PULPino is the first effort in doing so. The direct relation to the PULP project is even expressed in the name chosen for the project: In Italian, adding an "-ino" at the end of a word usually means that word corresponds to a minimized version. It has been always one of the projects main aims to provide a simple and easy to use computing platform with extensive possibilities to communicate with the outside world.

Apart from the open source release, having a smaller platform has some tremendous advantages for the PULP project as well. The PULPino platform easily allows us to evaluate new features without considering the overhead of the whole PULP platform in the first place. This is true regarding simple RTL simulation as well as for Synthesis estimates.

In addition to the opportunity stated above there is still the educational aspect of the project. Due to its simplicity it can be of great value for students who want to gather deep understanding of the basic building blocks of a micro-controller like system. This relates to the SoC architecture as well as the idea and construction of a RISC core. It is often useful for ones understanding of a concept to have the possibility to observe a working implementation.

Last but not least, we hope that the open sourcing of PULPino helps the open source hardware community to gather more momentum in the development of software tools necessary to simulate, synthesize and manufacturing open ASIC designs.

PULPino is mainly targeted at RTL simulation and ASIC's, although there is also an FPGA version available.

1.1. Scope of this thesis

Part my work on the project has been on the software side as well, namely the implementation of an operating system for PULPino. As the PULP project runs its programs solely on bare metal, porting an OS is something that has not been done before as of several reasons: Running an operating system on a multi-core platform is not such an easy task. Especially most of the operating systems targeted to micro controller do not support any kind of multi-core environment. Although there have been some people that tried porting it it never really established a reasonable stable state. As a matter of fact porting it PULP would be a very time consuming process that has not been done yet by anybody from the group.

Things look entirely different for PULPino. Since we are having a strong focus on simplicity and are only confronted with a single core environment this task gets considerably easier. The exact implementation of freeRTOS is discussed in sec (TODO: find section).

1.2. General Overview

PULPino provides you with a 32-bit Harvard architecture (e.g. it has physically and logically separated instruction and data RAMs). At its heart it has a Reduced Instruction Set Computer (RISC) core operating. We currently support two different instruction sets (ISAs) for two distinct cores. This can either be our OpenRISC core OR1ON or our RISC-V implementation RI5CY. The cores are pin compatible and can therefore be swapped at one's convenience. Both cores process instructions within a four stage in-order pipeline.

The core has debug support enabled through the Advanced Debug Unit (ADB) partially adapted from the OpenCores project. The debug unit provides outside world communication via a standardized JTAG TAP. The core region (including the core, the debug unit and the RAMs) is communicating over a standard Advanced eXtensible Interface Bus (AXI). A dedicated AXI to APB bridge connects the internal AXI bus to the (slower) Advanced Peripheral Bus (APB). Both bus specifications are part of theAdvanced Microcontroller Bus Architecture (AMBA) specification.

1.3. Document Structure

This document is separated into two different parts. Part I deals with the general concept behind PULPino and everything that is needed to start developing programs and/or specialized IPs easily. It starts with an introduction to the build framework and the project structure so that the reader can easily follow along. If you want to have the sources while reading the chapters I would highly recommend to start with reading the appendix on how to check out the project and get everything up and running.

I then aim to give a more detailed description of the overall architecture, the different IP cores and their peculiarities.This section concludes in a explanation of the functional verification framework that is shipped alongside PULPino.

The second part contains ASIC (Imperio) specific information. It gives insight on the measures taken for the tape-out as well as chip related information and concludes with a chip data sheet. Since Imperio is the ASIC of PULPino everything explained in the first part of the document is directly applicable to Imperio as well.

In the appendix you can find a summary of details needed to start developing for PULPino. This includes a register description anda API description amongst others and it is supposed to act as a quick reference card for application developers.

1.4. Related Work

At the moment one can see a landsliding transition happening in the open-source hardware community. This began with the effort of the OpenRISC project in (?) which was the first open-source release of a micro-controller like architecture. At the moment this is currently climaxing with Berkely's RISCV project and conferences on a regular basis that focus entirely on open source hardware development (e.g.: ORCONF and RISCV Workshops).

Although there are several implementations of openRISC and RISC-V cores freely available the PULP project group has always been one of the early adaptors of both ISAs, hence we developed and maintain our own cores from the very beginning. This has the advantage of being leading edge from the very beginning and being able to improve our cores in terms of area and power.One principle idea of PULPino is to give a well established and silicon proven design back to the community on which they can build their implementations and/or extensions on.

1.4.1. OpenRISC

OpenRISC is a project started by the OpenCores community in order to establish an open source ISA and provide a reference implementation in Verilog. The first core design

was called OpenRISC 1200 (OR1200) and has been published under GNU General Public License (GNU GPL). Based on that core design they have implemented a SoC variant called ORPSoC (OpenRISC Reference Platform System-on-Chip). Unfortunately until the time of this writing there has been no open source ASIC implementation of ORPSoC but the project is still under maintenance and can be run on FPGAs as well.

Nevertheless there are numerous commercial ASIC implementations based on OpenRISC 1200 and ORPSoC, for example BA12 from Beyond Semiconductor or SD1106 E by Samsung used as a main processing unit in there Digitial Television (DTV) devices.

1.4.2. RISC-V

RISC-V is a project initiated by the electrical engineering department of University of California Berkely (UCB). It aims to create an open and freely available Instruction Set Architecture (ISA) standard. The design of the ISA aims satisfy a very broad field of application purposes. Ranging from small scale micro-controllers to full-blown out-of-order many core architectures.

Specifically interesting in relation to the present work is their Z-scale implementation. It features a 32-bit 3-stage single-issue in-order pipeline with support for the RV32IM ISA (integer base arithmetics and multiplication) and M/U privilege modes. Communication with the memories takes place over a 32bit AHB-lite bus.

Currently UCB provides to versions of the z-scale core. One is written in their own hardware description language (HDL) called chisel while the other implementation is conducted entirely in Verilog. Both are distributed under a 3-clause BSD license.

The emerging ecosystem that comes along with the growing popularity of the RISC-V ISA comes in handy for the PULPino project as well. The various virtual platforms provided by UCB can emulate code that will natively run on PULPino. In addition it provides us with a tool-chain that supports the official RISC-V ISA.

1.4.3. Spain University

1.4.4. lowRISC

lowRISC is a community project that aims to create a fully open hardware system. This also includes the processor design which they intend to base on the RISC-V ISA. Their final aim is to start volume production of open silicon and open source PCB designs.

They plan to have specialized cores entirely dedicated to external I/O (so called minion cores). This will give hardware support for basic tasks such as shifting data in or out more efficiently. In a more final implementation minion cores should also be able to handle more complex communication protocols like Ethernet or USB for example. lowRISC's

current idea is to use the RISC-V core developed at ETH Zurich for their minion cores implementation. The very same core is used in for PULPino's ASIC implementation described in the second part of this document.

Chapter 2

Preliminaries / Background

2.1. Getting Started

Chapter 3

Theory / Algorithms

3.1. FreeRTOS

FreeRTOS is a popular, well-supported, open-source real time operating system published under GPL. It is widely used throughout the industry for various projects and it sees itself as the de-facto standard solution for microcontroller and small microprocessors. It underlies strong peer reviewed quality control and provides the developer with well documented sources.

With its highly configurable nature it allows for preemptive scheduling as well as cooperative scheduling. Additionally it offers queues for inter process communication (IPC) and synchronization constructs through critical sections and mutual exclusion (mutex). It can also make use of more advanced hardware features like a memory protection unit (MPU) and different privilege levels.

The hardware dependent code is cleanly separated through a distinct portable layer that abstracts all hardware related features. This makes it easily possible to port application code between two different devices albeit their architectural differences. The operating system (OS) specific implementations like the scheduler and synchronization constructs are entirely written in C and therefore architecture independent. Depending on your resources and requirements it is possible to employ different memory allocations schemes. Several implementations are shipped with freeRTOS itself but you are free to implement your own allocation scheme. The shipped allocation schemes start from simple block based array allocation up to calls to standard (newlib) C library functions `malloc()` and `free()`.

To my extent this is the first implementation of a RISC-V portable layer. Although the implementation makes use of non (yet) standardized features like interrupts and timers it should be no big problem to adapt it for feature use of the standardized facilities.

3.1.1. Purpose of an Operating System

The purpose of an OS can be manifold. But what unites them all is the purpose to share computational resources between different tasks (programs, processes). Essentially giving the application user the feeling as if the programs are executed in parallel. It therefore distributes the processing time amongst each of the programs. Furthermore it should be possible for the tasks to communicate with each other and to acquire shared resources (for example I/O or a shared variable). This is especially true for the case of freeRTOS.

In freeRTOS scheduling is provided in two different ways. On the one hand it maintains a preemptive scheduler that can preempt the currently running task and give execution time to another task. On the other hand freeRTOS has a cooperative scheduling algorithm as well. This means that the program given control to, essentially runs as long as it returns control back to the OS. There is no way for the OS to interrupt the currently running task by any means. The process of switching between tasks is called context switch. Nevertheless freeRTOS uses the tick interrupt in this setting as well in order to figure out how long the task has been running and to eventually schedule a different task accordingly.

Regardless of the scheduling scheme it is crucial for the OS to be able to track whether a task is ready to run or e.g.: has previously been suspended. It therefore assigns it to different states depending on the reason the task was swapped out beforehand. In freeRTOS a task can be in one of the following four states:

- Running: When a task is actually utilizing the processor it is said to be in the running state.
- Ready: A task is said to be ready if it is not utilizing the core at the moment (it is not in the running state) but is ready to run as soon as the OS decides to switch it in.
- Blocked: A process is blocked if it currently waiting for a temporal (e.g.: a timed delay) or an external event (e.g.: release of a shared resource).
- Suspended: A task is suspended if it has been told so through an explicit function call to `vTaskSuspend()`.

The whole state diagram is depicted in figure 3.1. When a task gets switched in the scheduling algorithm has to decided which new task is given processor time. There are different approaches to that. For example a round robin scheduler assigns fixed time slots to every process and the different tasks are therefore provided with equal computation time. FreeRTOS pursues a slightly different approach. At the creation of each task the programmer can assign a priority to the task he is creating. Based on the priority freeRTOS schedules the next task accordingly, beginning with the highest priority task either in the ready or running state. Least precedence is given to lower priority tasks.

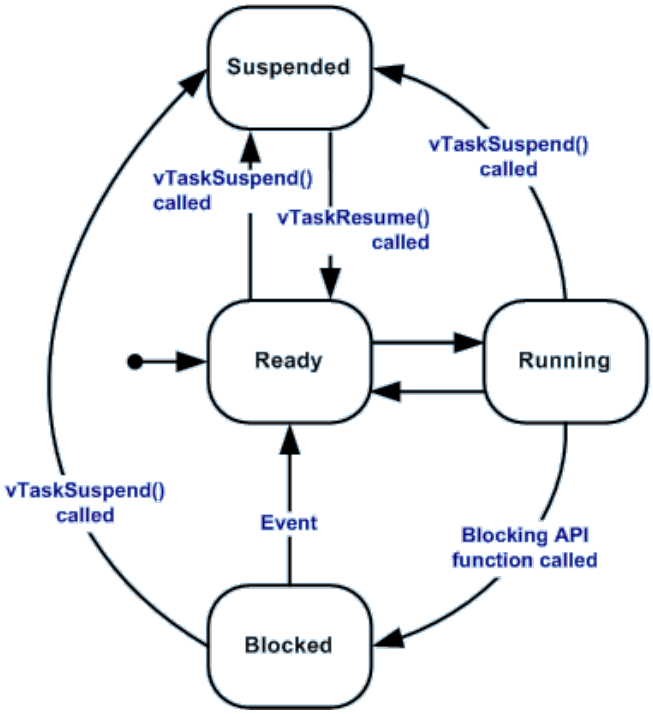


Figure 3.1.: FreeRTOS task state diagram. TODO: list resource

The idle task has priority zero. If two tasks share the same priority freeRTOS schedules them in a round robin fashion.

In the preemptive case the OS needs a way to preempt a currently running task. It does this by registering a interrupt service routine (ISR) triggered by a timer that is configured by the OS according to the users specification. The interrupt triggers an asynchronous change in the control flow and directs control back to the OS that then decides whether the current task shall be switch out or not. Figure 3.4 depicted the difference between a preemptive and cooperative context switch.

3.1.2. Directory structure

Since freeRTOS aims high portability, the directory structure directly reflects this property. Everything that is architectural specific resides in the `portable` folder. This begins with basic configuration macros and continues with implementation of the actual context switch in `port.c`.

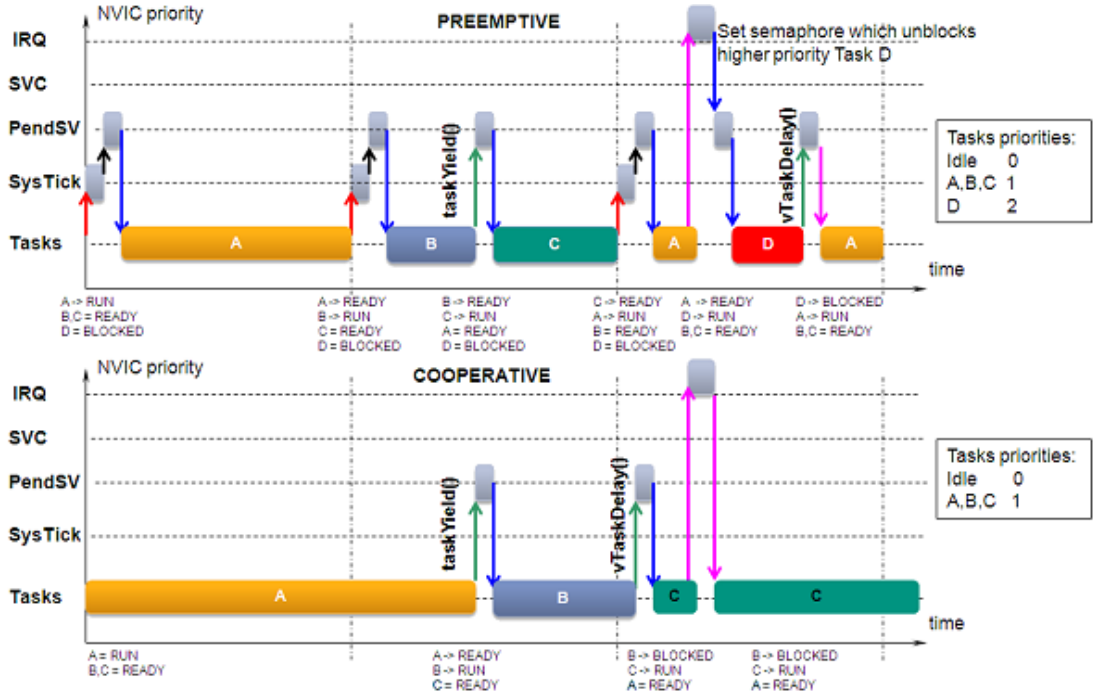


Figure 3.2.: Preemptive and cooperative context switch

/
- `main.c` ..... Entry point of user program
- `FreeRTOSConfig.h` User defined and application specific freeRTOS configuration
- `update-ips.py` . This script reads the ip_list.txt and pulls all ips from the GIT server
- `include` .....Folder containing scripts used by continuous integration
- `portable` .....Contains the actual source files of your report.
  - `GCC` ..... Portable layer specific to GCC
  - `RI5CY` ..... Portable layer specific to the microcontroller's architecture
    - `portmacro.h` ..... architecture related configuration file
    - `port.c` ..... Actual portable layer implementation
  - `MemMang` .....Memory management schemes.
- `event_groups.c` ..... freeRTOS eventing scheme.
- `list.c` .....List implementation - Task utility
- `tasks.c` .....Task implementation.
- `timers.c` .....Software timers.

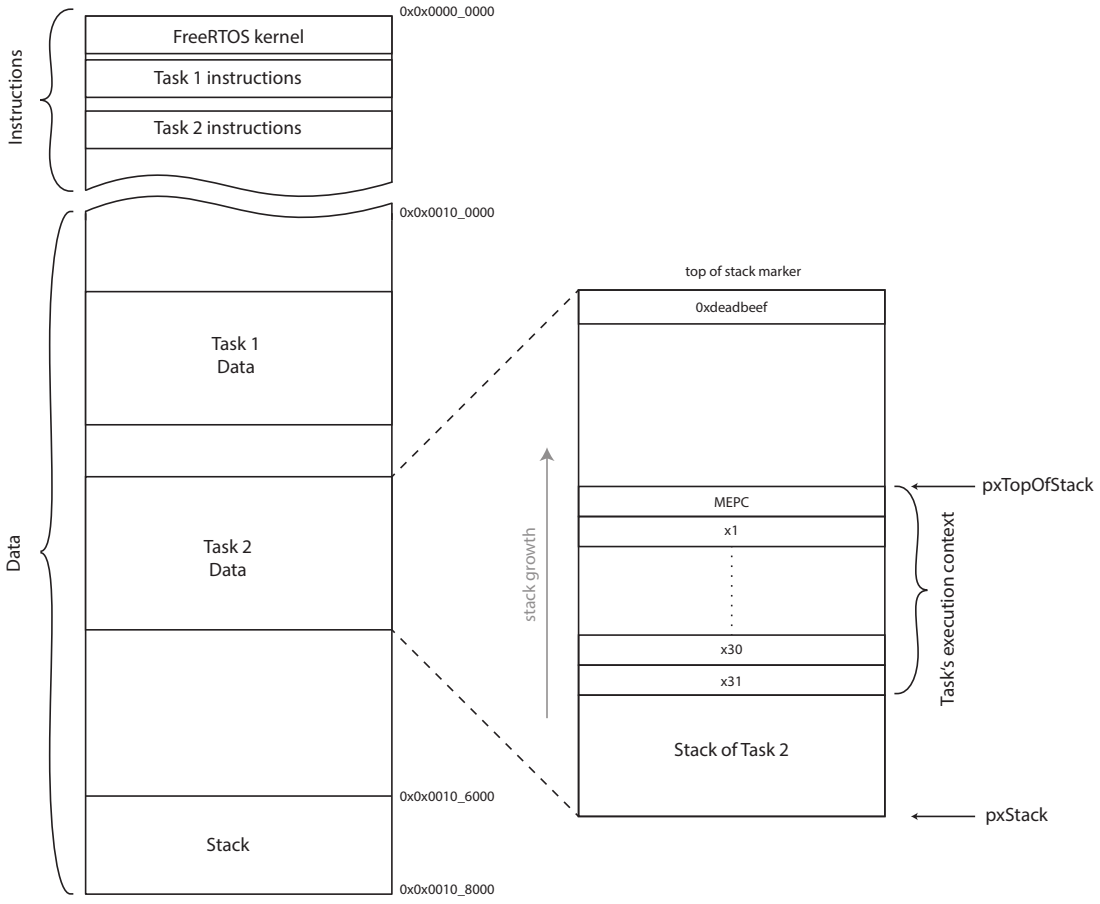


Figure 3.3.: FreeRTOS memory management and stack content of a switched out task (Task 2)

3.1.3. Portable layer

The portable layers purpose is to abstract all device specific information to the OS. In particular this means implementation of the actual context switch, interrupt service routines (ISRs) that trigger the context switch and stack initialization for task creation. All device specific codes resides here.

3.1.4. Context Switch

Context switches can occur in two different settings depending on the scheduling algorithm in use. According to that, there are differences in the program control flow. Nevertheless the main idea of the context switch stays the same.

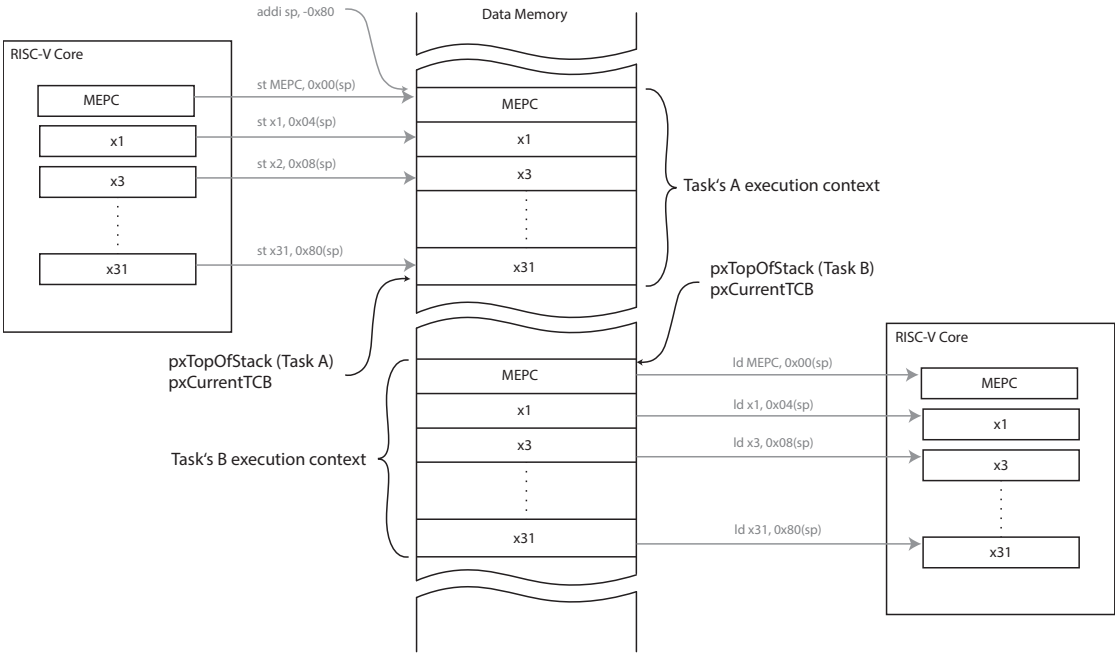


Figure 3.4.: FreeRTOS context switch

Each task in freeRTOS manages its own stack (figure 3.3). The task's memory region is allocated when the tasks gets registered for the first time and reside entirely in the heap regardless of the global linker settings. In order for FreeRTOS to manage the task's memories it stores task related information in its own task structure called Task Control Block (TCB). The TCB has essentially the following structure:

```
typedef struct tskTaskControlBlock
{
  volatile portSTACK_TYPE *pxTopOfStack;
  xListItem       xGenericListItem;
  xListItem       xEventListItem;
  unsigned portBASE_TYPE uxPriority;
  portSTACK_TYPE *pxStack;
  ...
} tskTCB;
```

The `pxTopOfStack` variable points to the last element put on the stack of the task and must be the first element of the TCB structure. `pxStack` on the contrary points to the stack's base address.

If a task is switched out its current state is going to be saved into memory. For the

RISC-V RV32I this implies at least all general purpose registers and the MEPC register that has the program counter stored from which the interrupt service unit may return to. It does so by calling the `portSAVE_CONTEXT()` macro defined in `port.c`. This macro allocates some space on the stack and stores every register in a predefined order, starting with the MEPC. Finally it updates the `pxCurrenTCB` pointer that points to the TCB of the task switch out and updates the stack pointer. The stack pointer is therefore saved implicitly in the TCB.

The OS then figures out which task that is going to run next (based on the principles mentioned beforehand) and afterwards calls the `portRESTORE_CONTEXT()` macro. This essentially performs the same function as the save routine but in reverse order, e.g. it loads the stack pointer of the task that is going to be switched in from the pxCurrentTCB and restores all registers from memory in the same order as they have been saved.

For the cooperative case this happens within a normal function call, e.g.: a task decides that it may wants to yield and therefore gives a call to `vPortYield()`. The scheduler saves the current context and restores it after it has figured out which task is going to be switched in. It is crucial that the save and restore sequence does not get interrupted therefore interrupts are disabled throughout the whole process.

For the preemptive case this happens within an ISR. The only difference to the cooperative case is that the return from the interrupt (ERET) will use the MEPC register to find out where the current task has been interrupted from.

Context Switch Overhead

It is crucial for the property as a real time operating system to know exactly how long (in terms of processing time) certain sequences of code need. While this is often very difficult to state for more complex systems this is quite feasible in the context of a single core microcontroller and especially for PULPino. For the current implementation of freeRTOS I can give this exact cycle time break-down:

1. 43 cycles for storing of all registers
2. 64 cycles for scheduling the next task.
3. 43 cycles for the restore sequence of all registers
4. 68 cycles overhead for the interrupt call and all subsequent function calls and returns. Interrupt and function calls need a certain amount of time imposed by the compiler to obey the callee/caller convention (e.g. certain registers need to be saved by the callee and some by the caller of a function).

This makes a total of 218 cycles for the overall context switch e.g. from the time of one task becoming suspended to the time another tasks starts executing.

3.1.5. Stack initialization

The purpose of the stack initialization is to setup a task structure in the data RAM to look like it has been already running and was only switched out by the scheduler. To this point the OS has already allocated some space on the stack. What remains for the initialization to do is to store the start address of the task on the same place we would have stored the return address (register x1) and the MEPC if the task has been switched out. It therefore can simply restore the context of the task albeit it has never been running before.

Remaining functionality that is architecture depended is the configuration of the timer interrupt and a call to return to directly jump to the tasks code. We have to manually call the `ret` directive since we do not want to return to the function that has called `xPortStartScheduler` but to the program.

Part II.

Imperio

Chapter 4

Hardware Architecture

In this chapter I will give an architectural overview of the PULPino SoC. As described in the instructional chapter both cores available for PULPino feature a 32-bit 4-stage in order pipeline with distinct ports to the instruction and data RAMs. Since the cores are totally pin compatible one can swap them as needed without modifying any RTL.

PULPino uses a 32-bit wide AXI as its main interconnect. The memories and the core itself are connected to the AXI Bus via dedicated bus adapters. The APB peripherals are connected to the AXI bus through a AXI2APB adapter. As all components withing the system have access to the AXI bus they share a common memory map. This makes it particularly easy to write and read registers from each peripheral, the core and memories' content. The overall device architecture is depicted in figure 4.1.

4.1. Core

PULPino comes with two cores available. Both cores have been developed by the PULP group. This can either be the RISC-V core RI5CY or the pin compatible OpenRISC core OR10N.

4.1.1. OpenRISC - OR10N

OR10N was developed as part of a semester thesis here at IIS by Renzo Andri and Matthias Baer in 2014. It was meant to replace the former OpenRISC 1200 core used for the PULP project. The core employs a 32-bit 4 Stage in-order pipeline and has shown to be significantly faster then the reference implementation of the OpenCores community called OpenRISC 1200 [1].

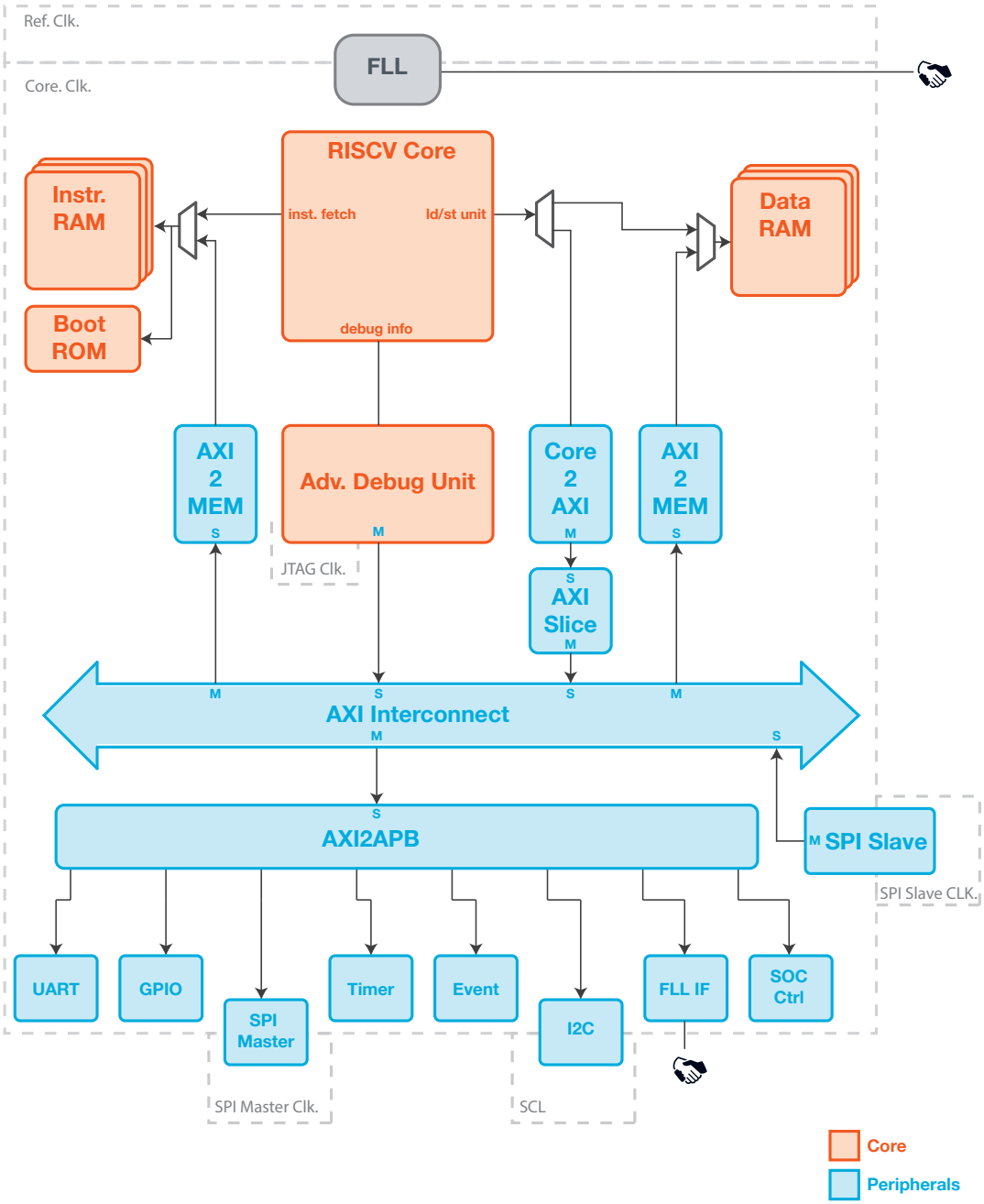


Figure 4.1.: PULPino block diagram

4.1.2. RISC-V - RI5CY

RI5CY is a 4-stage in-order CPU based on the freely available RISC-V instruction set developed at UCB. The core has been mainly developed by Sven Stucki as part of his master's thesis (TODO: Citiation??). RI5CY is loosely based on OR10N.

The RISC-V instruction set comes in a very modular and extendable flavor [2]:

- **RV32I**, **RV64I**: Is the 32-bit or 64-bit respectively, base integer instruction set. It describes general instruction formats and includes all operations to modify (integer) data and control flow. We fully support the bas integer instruction set with our RI5CY core.
- **M Standard Extension**: The M standard extension describes multiplication and division operations that multiply or divide values held in two integer registers. We provide only partial support for the M extension since our current multiplier implementation uses a single-cycle 32 bit lower result multiplier. As a matter of fact we do not support divisions and multiplications that return the upper half of the result.
- **A** (Atomic), **F** (single precision floating point) and **D** (double precision floating point) **standard extension**: We currently do not support any of these extensions.
- **C standard compressed ISA**: The compressed ISA specification is currently a proposal but will likely be frozen in the near future. The compressed ISA aims to reduce static and dynamic code size by adopting a simple compression scheme (i.e. small immediate values, one of the registers is the zero register $x0$,...). According to the current compressed ISA draft specification typically 50%-60% of the RISC-V instructions in a program can be replaced with RVC instructions, resulting in a 25%-30% code-size reduction [3].

In addition the RISC-V ISA provides the opportunity to define instruction set extensions that are not currently officially supported. We do make particular use of this with support of post-increment load and store instructions, hardware loops and packed-SIMD instructions.

4.2. AXI Interconnect

PULPino features an AXI bus as its main interconnect. Memories and the core are connected via dedicated bus adapters developed by the PULP group. The main advantage of having everything sharing one interconnect is the fact that the whole architecture becomes memory mapped.

4.2.1. AXI Slices

The AXI interconnect can be sliced when becoming necessary. This means that registers are getting inserted at the specified position. This can be of help if timing on certain parts of the interconnect becomes tight. Slicing the interconnect comes at the price of loosing one clock cycle (of course depending on the number of slices inserted) upon access to this certain part of the interconnect. In the present PULPino design this was necessary between the core bus adapter and the bus itself due the criticality of that path.

4.2.2. SPI Slave

The SPI Slave is a special peripheral in the sense that it allows to receive and send data without any interaction of the core itself. This allows us to treat PULPino as a normal SPI Slave that can get its whole memory region written from an external SPI Master (for example another microcontroller). This has the advantage that we can pre-load its entire memorie region (therefore writing all RAMS and arbitrary configuration registers). This features is inherited from PULP that was designed with the goal in mind that it will almost always be used in the context of an external host environment. Hence this provides us with another way of booting PULPino.

The SPI Slave has an AXI master through which it can access the whole memory region. It can be either used as a standard SPI Slave or as an Quad SPI which takes four bi-directional data lanes and provides four times the data throughput compared to normal SPI.

If you plan for an ASIC implementation you may want to reduce the pin count by only using standard SPI.

4.2.3. Memory and Core adapters

We provide special memory and core adapters which are hooked up to the AXI bus. This allows us to read and write all memories through AXI. In particular this can be useful when loading a program (see 4.2.2) or testing the memories.

4.3. Advanced Debug Unit

The advanced debug unit (ADU) originates from an OpenCores project. It has been adapted by us for the use in the PULP project. On the one side the ADU interfaces the core with a custom made debug interface that is the same for the OR1ON and RI5CY core and an interface to the AXI Bus. On the other hand it provides a JTAG TAP (Test Access Port) that can be interfaced by the standard JTAG protocol.

With the ADU it is possible to debug the core by a software debugger such as GDB running on a separate host PC.

The ADU is designed with different modules in place. On the one hand side it the JTAG TAP module that implements the JTAG protocol and on the other hand we have a module for the AXI and one module for the core. Each module must be dedicatedly activated before it can be used.

Sub-modules generally consist of two parts: internal registers, and a bus interface. Internal module registers may contain information about the status of the module (such as the error register in the AXI4 module), or they may control external I/O lines (such as the reset and stall lines from the OR1ON module). Each sub-module using one or more registers contains an index register, which enables one internal register at a time for reading or writing. Internal registers are selected, read, and written by sending commands to a sub-module through the TAP.

4.4. APB Peripherals

4.4.1. Interrupt and Sleep Unit

The unit is conducted as an APB peripheral. Each interrupt can be enabled or disabled and can have its pending status set or cleared by software (resulting in the fact that we support "soft interrupts"). The units main purpose is to keep track of all arriving interrupts and outputting the interrupt with the highest priority (starting from 0 up to 31) - iff it is enabled - to the core in a one-hot encoded fashion.

The interrupt unit can basically handle two types of interrupt sources:

- **Pulsed interrupt request** - the interrupt request is at least one clock cycle long. When the interrupt controller receives a pulse at its interrupt input, the pending status is set and held until the interrupt gets served.
- L**evel triggered interrupt request** - the interrupt source holds the request high until the interrupt is serviced.

The interrupt output to the core is level sensitive. As long as there is a pending interrupt the corresponding interrupt request line (`irq_o`) gets asserted. If the core wants to acknowledge an external interrupt it needs to clear the corresponding pending interrupt by writing 1 to the `clear_pending` (ICP) register. The interrupt is then de-asserted.

The event unit has exactly the same RTL layout as the interrupt unit. The only difference is that its `signal_o` is not forwarded to the processor core. An event can wake the core from sleep state. (TODO: provide a facility to get the currently pending event).

The sleep unit's task is to put the core into a low power mode by disabling its clock (through a clock gate). It therefore needs to track the sleep status of the core through a simple FSM (see figure **??**). The sleep unit is tightly coupled to the interrupt unit since it needs to wake the processor if an enabled interrupt arrives. The sleep controller is as well conducted as an APB peripheral with two registers (see figure **??**):

1. **Sleep Controll Register:** By writing the lowest bit (SE - Sleep Enable) to 1 the core requests to be put to sleep. Upon wakeup this bit is cleared by the controller.
2. **Sleep Status Register:** This register contains the current sleep status of the core. This is helpful if for example you want to get the sleep status through the debug interface. Writes to this location are ignored.

If the core signals that it is idle and wants to be put to sleep it writes the SE register. The controller then enters the `SHUTDOWN` state where the fetch enable signal of the core is de-asserted and no more instructions are fetched. After the processor has flushed its pipeline (signaled by a low `core_busy` signal) its clock is gated and the `SLEEP` state is entered. It now resides in a lower power state and can only be waked up by an external interrupt event (`signal_i` asserted).

Because the interrupt unit has control over the fetch enable signal of the core it has been immediately suggested itself to put the fetch enable synchronizer into the event unit as well. Because the event unit is clock gated after reset by PULPino peripheral (4.4.3) it was necessary to also include a clock signal that is not clock gated.

4.4.2. Timer Unit

The timer provides a facility to cycle accurately time certain events and/or trigger interrupts or events respectively. In detail it should be possible to generate a timer overflow interrupt/event if the timer reaches its highest value and a timer output compare event when the timer is equal to a certain value set in the timer output compare register.

Additionally a timer control register makes it possible to set a pre-scaler, enable/disable timer overflow and timer output compare interrupts/events and to start and stop the timer respectively.

4.4.3. PULPino peripheral

In order to control certain features of the SoC a special set of control registers is needed to do so. Since this is highly application specific (e.g. it only applies to this distinct SoC configuration) it has been my indention to logically and physically separate this sort of functionality into an independent peripheral.

This peripheral has control over the following registers:

- Control of all pads with double functionality: Whether they should be used as GPIO or perform there special function like SPI, UART, etc. The default value is set so that the special function is activated. For further information on how the pads are configured refer to table D.2.
- Control of peripheral clock gating: In order to save energy most of the peripherals are clock gated by default. Actually the only exception to this is the PULPino peripheral itself. To activate the desired peripheral you have to set the corresponding bit in the clock gating register.
- Boot address: The register manages the boot address from which the core start fetching its instructions once the fetch enable signal has become asserted.
- Version String: This peripheral provides a hard coded version string that represents part of the current configuration which is especially crucial for the ASIC implementation. The version string contains basic information about the system's configuration like RAM and ROM sizes and whether there are instruction and data caches in place:

Version Register

For a detailed explanation about the single bitfields please refer to register description in F.2.

- Pad configuration: For the pad's primary function the peripheral controls the configuration parameters like slew rate, drive strength and pull-ups. For a list of all reset values refer to D.2.

4.4.4. GPIO

General prupose input output is managed through this dedicated peripheral. It controls whether the signal is an input or an output. In addition it allows you to configure the pads according to your needs, similar to the PULPino peripheral does for specialized I/O. Furthermore the GPIO peripheral can trigger interrupts when the level of a pad is changing.

4.4.5. I2C

I2C (Inter-Integrated Circuit) is a serial bus interface invented by Philips Seminconductor. It is typically used for communication with lower-speed ICs like for example an EEPROM on a printed circuit board (PCB).

I2C uses two bidirectional open-drain lines called Serial Data Line (SDA) and Serial Clock Line (SCL) pulled up. It supports multi-master and multi-slave bus systems. Arbitration

in multi-master clock gets done by specifiying bus access schemes so that no more than one master is accessing the bus at once.

In our I2C implementation the clock is derived from the main system clock by setting an appropriate prescaler value. The current status of the I2C peripheral is held in the corresponding status register. Dpending on the value of the command register the I2C performs a different operation. This can either be a start, stop, read and write operation depending on the bit set in the command register.

The control register allows you to enable interrupts. The interrupt flag can also be polled from the status register in case interrupts are not enabled. In addition interrupts get generated when either the transmission has been successful or bus arbitration has been lost. The interrupt needs to be cleared in the event unit as well as in the I2C peripheral by writing 0x01 to the command register. For a detailed explanation of the registers refer to the register description in the appendix F.2.

4.4.6. UART (Universal Asynchronous Receiver Transmitter)

A universal asynchronous receiver/transmitter (UART) is piece of hardware that translates data between parallel and serial forms. It was frequently found in former times as part of the RS-232 interface of personal computers but is still widely incorporated on micro controller platforms. This UART is mostly based on the 16750 UART developed by Texas Instruments (TI) and was written by Sebastian Witt for the OpenCores project. It is distributed under a LGPL license.

The design is pin and register compatible to TI's 16550/16750 UART. It allows for variable baud generation based on the main system clock. It supports the following features:

- Full synchronous design
- Pin compatible to 16550/16750
- Register compatible to 16550/16750
- Baudrate generator with clock enable
- Supports 5/6/7/8 bit characters
- None/Even/Odd parity bit generation and detection
- Supports 1/1.5/2 stop bit generation
- 16/64 byte FIFO mode
- Receiver FIFO trigger levels 1/4/8/14/16/32/56
- Control lines RTS/CTS/DTR/DSR/DCD/RI/OUT1/OUT2

- Automatic flow control with RTS/CTS
- All interrupts sources/modes

4.4.7. SPI Master

4.4.8. FLL Interface

4.5. FLL

Chapter 5

Design Implementation and Results

This chapter is about the implemented functionality and ASIC key design data (timing, area and power) and implementation decisions made.

5.1. Implementation

5.1.1. Multiplexed Pads

Since pads are often a quite limiting factor in the overall design it can be particularly useful to reuse them depending on the functionality required at a certain point in time.

The pads should remain highly configurable. Therefore it was necessary to multiplex all inputs and outputs of every pad instance that should perform a secondary function (for example function as an UART or GPIO). Depending on a configuration register in the APB PULPino Peripheral (4.4.3) the pad should switch functionality. In addition all pads need to be in a fixed configuration when the chip is on the tester (testmode is enabled).

The block diagram of the pad multiplexers is depicted in figure 5.1. In order to not clutter the circuit diagram too much output enable, input, output and configuration multiplexers are drawn for different functionality of the circuit. In fact all pins that share functionality feature all multiplexers depicted.

- **Output Enable (OE)**: For the pads primary function output is either tied to low or high depending on the main functionality. If the pad is configured as general purpose I/O it is connected to the configuration register of the APB GPIO module (see 4.4.4).

- **Input**: If the pad serves double functionality, in terms of that its primary function serves as input, its input pin is multiplexed to the main entity. Depending on how the pad is currently configured the the non-active input is silenced.
- **Output**: If the pad serves as an output in its primary function the output to the pad is selected in terms of its current configuration.
- **Pad Config**: Pad configuration is a 6 bit wide bus that sets the pads configuration, depending on its current functionality, accordingly (like Schmitt Trigger, Drive Strength, etc.).

An exhaustive pad list along with reset configuration values is depicted in the chip's data-sheet (D.2)).

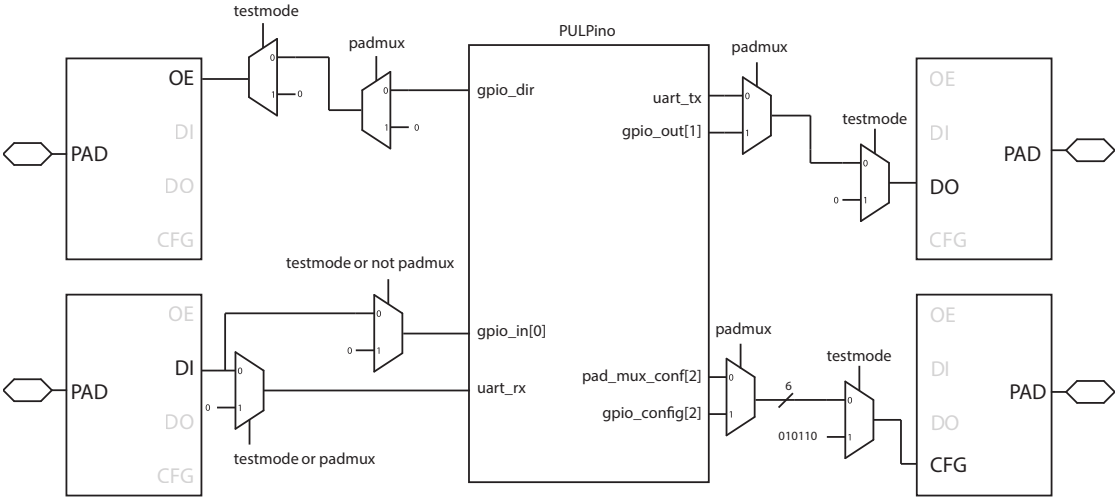


Figure 5.1.: Multiplexed Pads

5.1.2. Register file

PULPino provides to different implementation for its register file. One is standard array of flip-flops while the other makes use of clock gated latches. The first one has the advantage of being a very simple design choice that performs well in classical verification methods like Scan-Chain insertion but has the disadvantage of having an area overhead compared to the latch based register file.

As mentioned above the latch base register file is smaller and dissipates less power than the flip-flop based approach but isn't as, at least with classic full scan insertion, well testable as the flip-flop version.

To have a comparison of the two approaches I synthesized both versions and did automatic test pattern generation (ATPG) with TetraMax.

Type	**Area** (μm^2)	**Test Coverage** (%)
latch based register file	8541	86.2
flip-flop based register file	12278	97.2

Figure 5.2.: Comparison of latch based and flip-flop based register file (test coverage vs. area)

As you can see in table 5.2 swapping the latch based register file for a flip-flop version results in an area overhead of about 1.44 opposed to the latch based version, but results in a significant better test coverage.

5.1.3. RAM banks

5.2. Booting

Booting is an essential part of the ASIC implementation. Unless you are really accounting for loading program data into the microcontroller you are left with a worthless piece of silicon.

5.2.1. Booting from ROM

The boot process on Imperio is achieved by a 2 kByte of boot ROM that has instructions that read a SPI Flash and load the program's instructions and data into memories.

The compiled program is stored in the SPI Flash. It starts at address zero with 32 bytes of meta data that tell the bootcode where the instruction section begins, how many bytes are used to encode each instruction, where it should start fetching and how may blocks of fixed length it has to expect. The same is done for the data.

A special script takes care of preparing the flash content. This means calculating the headers and formatting the instructions.

The core starts executing boot code unless otherwise explicitly told so by writing the start address registers in the PULPino APB peripheral (see 4.4.3)).

The boot sequence starts by reading the vendor ID of the connected SPI Flash (we are only supporting a certain type of SPI Flash, where we definitely know all timing parameters). Next the SPI Master switches the communication to Quad SPI which is an extension to standard SPI but with 4 bidirectional data lanes. This has the advantage of providing a speedup of four times.

Next, the processor reads the 32 byte of header information and starts by loading instruction data into its instruction RAM. Afterwards the same is done for the data section of

the program. Finally it jumps to the base of the instruction RAM and starts executing the main program.

During the boot process interrupts are disabled and all peripherals except the SPI Master are in reset configuration. For the full boot code refer to C.3.

5.2.2. Booting over SPI or JTAG

Since the SPI Slave and the JTAG port of the ADU have access to the bus it is possible to store external data into the RAMs. This in fact can be used and is actually intended to give a way of loading an entrire program into Imperio's RAMs.

5.3. Verification

This section deals with functional and circuit level verification. In the functional section I will give detailed explanation on how PULPino and Imperio are tested.

5.3.1. Functional

For functional verification we mainly distinguish two test settings: PULPino related functional verification and imperio related verification. One can see Imperio's test setting as superset of PULPino's e.g. every test that can be run on PULPino can also be conducted on Imperio but not vice-verse.

PULPino testbench

PULPino tests are performed in a platform setting. This means that the core gets instantiated in the target platform setting and is then tested exclusively in this environment. This provides the advantage of having a very generic test framework which allows us to test (high level) assembly and C code. We have a specialized test library in place that performs checks for errors and generates reports that are output over UART. Currently we are supporting two different kind of tests:

1. **Sequential Tests:** Pure C code tests that perform common micro-benchmarks like sorting, matrix multiplication.
2. **RISC-V Tests:** Test that are exercising certain features of the core. This group also contains Berkely's official RISC-V tests that have been ported in the course of RI5CY's development. Additionally we are testing load/stores, control flow instructions and our instruction set extensions amongst others.

The testbench is highly configurable through parameters that are set when the corresponding testbench script is invoked (see the `vsim/scripts` folder for a list of available configurations see E.0.2). I want to emphasize two particular configuration parameters namely clock select and memload.

The first one selects whether the core should be clocked by an external reference clock or by its internal FLL. In terms of RTL simulation this does not matter too much since we do not have any timing parameters that may prevent us to use higher clock speeds through the external clock generation. But it allows for functional verifying the FLL.

The memload parameters specifies how the memories of the DUT should be filled with instructions and data. For PULPino we support:

- **Memory pre-loading**: In this setting we make use of the fact that we can directly pre-load the memories' functional model. Prior to starting the core all memories are filled with the content of `.slm` files that have been created of the corresponding object file with special conversion scripts (located in `sw/utis/`).
- **SPI Slave**: As discussed in the SPI Slave Section (4.2.2) PULPino's entire memory region can be written from outside via SPI. This makes it possible to load instructions and data through SPI.
- **JTAG**: Since the whole memory region is mapped to a common address space it is possible to write the RAMs through JTAG as well. Therefore it is possible to load a program via JTAG.

After the program has been loaded by either of the three possibilties specified above the fetch enable signal gets asserted and the core start computation. Depending on the test currently running the results are checked against a different golden model. This can either be some constants that where computed prior to compilation or other entities that get checked, for example a certain register's content.

Once the test passes this check, the result is reported back to the test bench library that outputs a predefined string over UART and signals the end of computation (EOC) event by pulling GPIO 8 high. Depending on the environment the test is running this string either gets checked manually or automatically by the `run_and_check.py` python script.

One particular point I'd like to stress is that it is necessary to set the boot address to the bottom of the instruction memory for all three options mentioned above. Since Imperio is designed to run as a standalone module on a PCB the default boot address points to the start of the boot ROM. Now if we pre-load the memories which is the case for PULPino's testbench we need to re-set the boot address accordingly.

Imperio testbench

For Imperio's testbench the approach is very similar. Actually the only reason why we employ a different test bench for ASIC related tests is, on the one hand, due to legal issues we are not allowed to open-source the models like that of the SPI Flash or I2C EEPROM for example. We therefore need a clean separation between those two testbenches. On the other hand the top instances of PULPino and Imperio are different in regard of inputs and outputs. While for PULPino there is only single-directional I/O this is no longer the case for Imperio since we have the pads instantiated in between.

Imperio RTL Simulation

As mentioned above Imperio's testbench extends the environment of PULPino's testbench in terms of that it instantiates behavioral models of two I2C EEPROMs and one SPI Flash. This has two advantages:

1. Using models in the testbench better simulates the PCB environment the ASIC is finally going to be employed.
2. It allows us to use a special boot sequence to load instructions and data from an external memory. In the case of Imperio this is going to be a Spansion SPI Flash. For the detailed boot code refer to the appendix C.3 and for an explanation of the boot process to section 5.2.

We are providing different CMake targets dependent on the way we want to load the program.

Post Synthesis Simulation

Post Layout Simulation

Continuous Integration

5.3.2. Design for Testability (DFT)

The RAM's input is bypassed in test mode in order to enable scan testing the combinatorial logic around the RAM. This is especially important for the RAM2AXI interfaces which allow operations on the bare memory. Furthermore, in order to observe the address pins of the RAM observation registers for the RAM address port have been added. The area overhead is small since Synopsys uses some sophisticated techniques to, on the one hand reuse the observation registers for both the instruction and the data RAM and on the other hand, only a few registers are necessary to make the signals of interest observable.

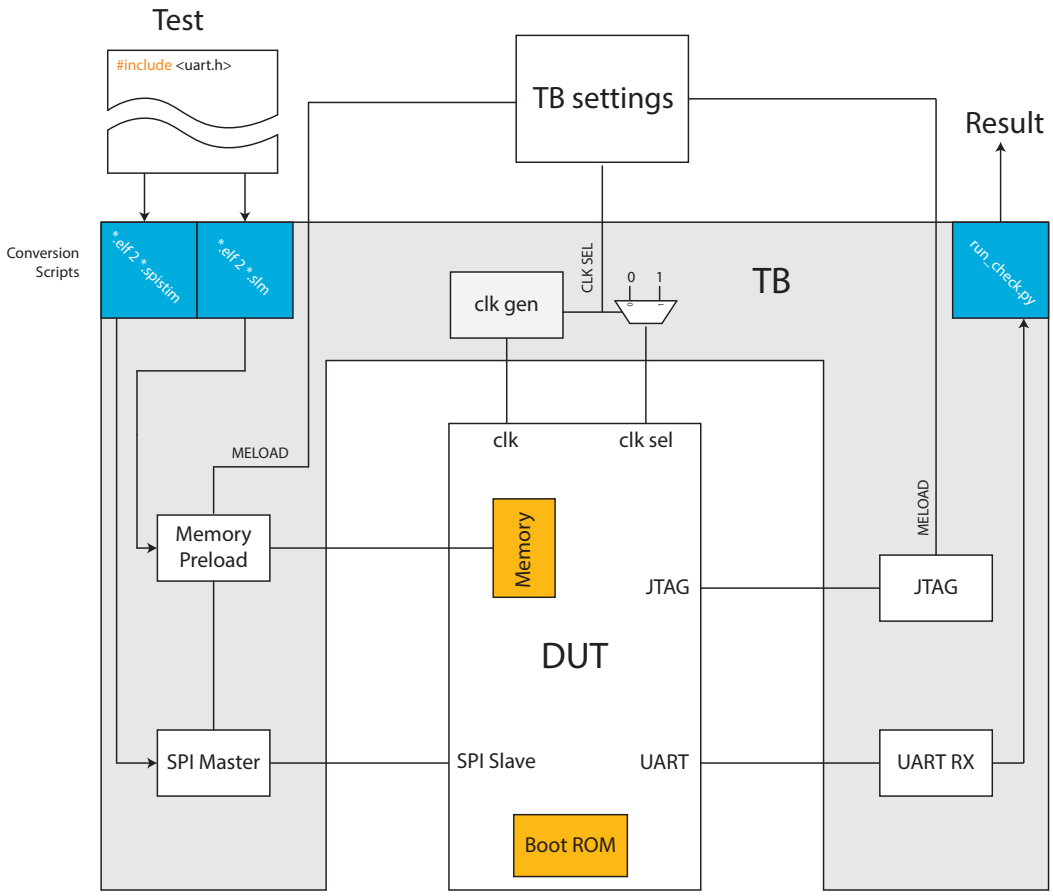


Figure 5.3.: Functional verification setup.

The memories itself will be tested with direct access through the JTAG interface.
The FLL has a dedicated DFT structure featuring a ScanEnable, ScanIn and ScanOut port. Until now the FLL is not part of one of the scan chains.
The current DFT numbers are related to post synthesis netlists. In a first run, a couple of weeks ago, I was able to achieve a bit more than 99 % test coverage. While in the latest runs, most probably related to the newly instantiated pad frame and the FLL, test coverage deteriorated a bit and I am at 97% for the current design.

Uncollapsed Stuck Fault Summary Report

fault class	code	#faults
Detected	DT	351339
Possibly detected	PT	68
Undetectable	UD	1267

ATPG untestable	AU	8065
Not detected	ND	99
total faults		360838
test coverage		97.72%
fault coverage		97.38%

Automated Testpattern Generation

5.4. Timing

5.5. Power

Lowering power has always been a key driving factor for the PULP group. As a matter of fact power consumption has been a point of interest for PULPino as well. Nevertheless due to PULPino's simplicity we spare most of the power saving tricks used by PULP.

In order to estimate power consumption I extended the current testbench to automatically trigger ModelSim's VCD dumps. Additionally I configured the CMake environment (for more information you may want to check the Getting Started Guide E.1) to have a distinct power target. This makes it particularly easy to run power estimations.

Although the whole design is getting compiled with clock gating enabled it turned out that the peripherals were dissipating a lot of power. Considering that I decided to implement more extensive manual clock gating. Concretely I employed a distinct clock gating control register into the APB PULPino peripheral. This register allows for clock gating every peripheral separately without relying on Synopsis automatic clock gating insertion alone.

Another measure I took, in order to lower power, was the usage of standard cells with different voltage threshold. The standard cells in UMC 65 technology come in three different flavors in terms of threshold voltage (VT), depending on the needs of the designer. Cells that have a high threshold voltage have slower transition times (the time needed to switch from one voltage level of logic 0/1 to the voltage level of 1/0) but do dissipate less leakage current (power dissipated by the cell in absence of any switching activity) than standard cells with lower threshold voltages. The other two types of standard cells are regular VT and low VT cells. The latter are switching faster but dissipate way more leakage current.

Depending on the requirements of the standard cell needed for a certain part of the design, the synthesis tool can decide which standard cell it wants to use. It will of course

always try to meet the speed requirement constrained by the designer but on the other hand it will also optimize in terms of leakage power dissipated.

5.6. Results

This section deals about the results I've received during design time.

5.6.1. Power

For the time being I conducted three different measurements that are mostly representative for Imperio's application context.

1. Interrupt Test: The core is clock gated most of the time in this scenario. It is woken up periodically and performs an UART transaction when it wakes up.
2. Hello World: Trivial UART output. The core is not put to a particular low power state. Most of the operations are memory operations (instruction and data fetches).
3. Matrix Multiplication: Computational intensive operation. The core needs to fetch a lot of data from the memories and performing computational costly operations, like multiplications and additions on it. This case should be representative for high density computation.

Power consumption can be split up into the following three components [4]:

- Internal power P_{int}: power dissipated for charging and discharging the cell's internal capacitances.
- Switching power P_{ext}: this amounts to the power that gets dissipated for charging external load capacitances (input capacitance of driven cell plus wire capacitances) that are connected to the cell's output(s).
- Leakage power P_{leak}: power dissipated by the cell in absence of switching activity.

The total power dissipated by the circuit is the sum of these:

$$P_{tot} = P_{int} + P_{ext} + P_{leak}$$

All power simulations where performed at 500 MHz, 1.2 V and 25 °C. Detailed results are depicted in table 5.1.

Table 5.1.: Power Results (UMC65 - LVT 1.2V) @ 2ns, 25 °C

Test	P_{int}	%	P_{ext}	%	P_{leak}	%	P_{tot}
Interrupt Test	7.10	57.84	5.00	40.64	0.17	1.52	12.28
Hello World	19.80	59.64	13.25	39.88	0.19	0.56	33.23
Mat. Mul.	32.00	60.00	21.17	39.68	0.19	0.35	53.35

5.7. Area

On the final design there is approximately 25% core area left (on a ninth module). Detailed area results are listed in figure 5.5. Note that the RAMs are no longer listed as a separate design entry as they are completely ungrouped for better synthesis results. Pad instances do not show up as seperate design entries since they are not over 1 kGE alone, but accumulated they are a significant part of the overall area.

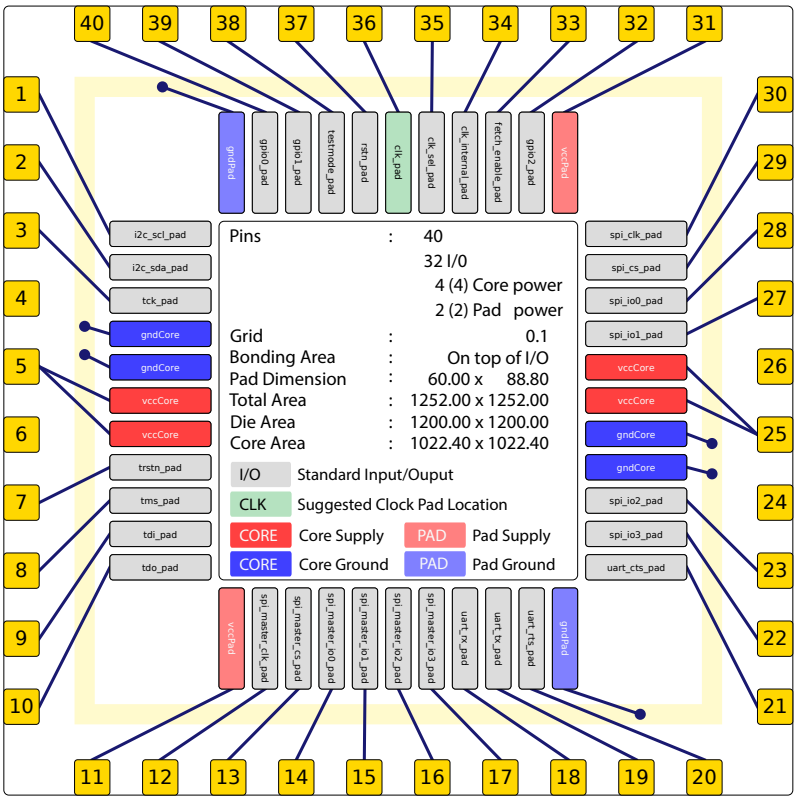


Figure 5.4.: Bonding diagram

Entity	**Total Cells (kGE)**	**% Total**
imperio	700	100.0
pulpino_i	500	73.1
axi_interconnect_i	6	1.0
axi_node_i	6	1.0
others	0	0
clk_rst_gen_i	12	1.8
others	0	0.0
core_region_i	400	63.6
RISCV_CORE	40	5.8
adv_dbg_if_i	5	0.8
axi_slice_core2axi	2	0.4
data_mem_axi_if	5	0.8
instr_mem_axi_if	5	0.8
others	0	0.0
peripherals_i	50	6.8
apb_event_unit_i	3	0.5
apb_gpio_i	3	0.5
apb_i2c_i	1	0.2
apb_pulpino_i	2	0.3
apb_spi_master_i	8	1.2
apb_timer_i	3	0.5
axi2apb_i	1	0.2
axi_spi_slave_i	8	1.3
i_apb_uart	14	2.1
others	0	0.0

Figure 5.5.: Area estimates (UMC65 - LVT 1.2V) @ 1.3ns - 1 GE = 1.44 µm^2

Chapter 6

Conclusion and Future Work

6.1. Conclusion

My contributions to the project have been very divers. This begins with the implementation of freeRTOS on the software side as well as on the hardware side with the implementation of the timer and interrupt peripherals. Those peripherals are simpler than the ones used by PULP and therefore better fitting our needs.

Moreover I contributed to the build and verification framework of PULP/PULPino by implementing new features like for example the core trace annotation as well as improving already implemented features like rewriting the the testbench's JTAG and SPI interfaces in a more object oriented fashion.

Finally I designed a well performing ASIC in terms of high speed and low power requirements.

6.2. Future Work

Imperio will be taped out in the end of January 2015. Since it has always been designed to be employed on a PCB this will certainly be some work that needs to be done. In particular it would be nice to have an development board that makes use of all of Imperio's features and peripherals. Furthermore it will be necessary to develop software in order to program Imperio accordingly.

Another aspect that should not be lost sight of is the support for the open source community. It will be crucial for the widespread gain of PULPino to have a community that is using and supporting it for their own projects and products. Especially at the beginning support will be one of the key driving factors for successful project.

Lastly I am hoping that the design is going to be employed in an educational aspect and that everybody has the possibility to learn as much as I did during this semester theses.

Appendix A

Task Description

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Semester Project at the Department of
Information Technology and Electrical Engineering

Fall Semester 2015

Florian Zaruba

PULPino implementation in 65nm CMOS

September 15, 2015

Advisers: Frank K. Gürkaynak (DZ), ETZ J60.1, Tel. +41 44 632 27 26, kgf@ee.ethz.ch
Michael Gautschi (IIS), ETZ J69.2, Tel. +41 44 632 99 58, gautschi@iis.ee.ethz.ch
Handout: September 14, 2015
Due: December 18, 2015

The final report will be turned in electronic format. All copies remain property of the Integrated Systems Laboratory.

1 Introduction

At the Integrated Systems Laboratory (IIS) we have been working on a Parallel Ultra-Low Power Processor (PULP) System for the past two years[1]. Throughout the project we have worked on different processor cores. For the first few designs, the original or1k code from the opensource project was used[2]. We then heavily modified this core and improved its IPC (instructions per clock) close to its optimum value of 1. In a next step we started a new implementation from scratch, and called this implementation Or10n[3]. This core has been the main processor core used in PULP projects.

The RISC-V[4] effort started also during this time and is being driven by the Computer Science Division of the EECS Department at the University of California, Berkeley. At the moment, this instruction set architecture (ISA) is very popular in academic circles, so much so that several members of the Open RISC projected have stated that they would stop working on the OpenRISC project and concentrate on RISC-V.

For this reason, we have also investigated the possibility to have a 32-bit version of the RISC-V architecture and during a recent Master Thesis we have developed RI5CY, our own implementation of the RISC-V32 architecture that is reaching a level of maturity close to that of the latest Or10n cores. This core has attracted much attention, also from the lowrisc project[5] of the Cambridge University with which we have a close collaboration.

From the start the intention was to make the entire PULP project open, and provide everyone access to high-quality processor complete with all support (tool flow, debugging, FPGA and ASIC mapping scripts). Until now, we have been unable to rollout a release mainly due to lack of experience in large open projects, and the state of the documentation. As a first step we have decided to release PULPino, a single core processor that uses the same cores we have developed, together with a selected set of peripherals. The release is expected within 2015 and should also contain an FPGA mapping to the popular ZED platform which will allow many people to start using it.

This project has attracted considerable attention, and we have been asked whether or not the processor would support operating systems. In several instances a desire to run FreeRTOS[6] as well as sel4[7] was stated.

2 Project Description

The goal of this project is to implement the first ASIC version of PULPino in UMC65 CMOS process, and to make the necessary changes to the system to run FreeRTOS and/or Sel4. The group is already working on a PULPino release (for FPGA) and the goal is to work together to adapt and extend the PULPino so that it can also support the FreeRTOS and/or sel4. This will at the very least require a timer unit that needs to be added to the system, but may also require some additional changes such as support for supervisor mode commands etc. The tape-out is scheduled to coincide with the student tape-out early in January 2016.

We will join the Orconf conference in October in Geneva[8], to present the PULPino project to a wider audience, and we expect to get some more feedback from the community in general on this topic. Following this conference, and time permitting, we might wish to integrate the ideas from this meeting into the current project as well.

3 Goals

First and foremost, the goal of this project, is to learn and experience the design process for a digital ASIC. At the end of the project a chip in UMC65 will be taped out. We expect that the manufactured chip:

- Uses the latest PULP core either Or10n or RI5CY.
- Be able to boot from an external EEPROM like other PULP systems
- Run an operating system (FreeRTOS, sel4 or other) and a demo application on it.
- Have a basic set of interfaces (SPI, GPIO)

We would like that this work complements our current PULPino project. Furthermore, it would be good if the result of this project could be done in a way that will benefit future collaborations with our partners, specifically ACP which has an interest in running OsmocomBB on FreeRTOS and Simless (part of the LowRISC project) that wants to develop a security module based on a core running sel4.

4 Project Realization

4.1 Project Plan

Within the first month of the project you will be asked to prepare a project plan. This plan should identify the tasks to be performed during the project and sets deadlines for those tasks. The prepared plan will be a topic of discussion of the first week's meeting between you and your advisers. Note that the project plan should be updated constantly depending on the project's status.

4.2 Meetings

Weekly meetings will be held between the student and the assistants. The exact time and location of these meetings will be determined within the first week of the project in order to fit the students and the assistants schedule. These meetings will be used to evaluate the status and progress of the project. Beside these regular meetings, additional meetings can be organized to address urgent issues as well.

4.3 HDL Guidelines

Since most of the PULP project is written in System Verilog, it is strongly suggested to use System Verilog for this project as well. However, any other HDL can also be used if there are strong arguments to use them.

Adapting a consistent naming scheme is one of the most important steps in order to make your code easy to understand. If signals, processes, and entities are always named the same way, any inconsistency can be detected easier. Moreover, if a design group shares the same naming convention, all members would immediately *feel at home* with each others code. At the IIS we make use of the naming convention proposed by the Microelectronics Design Zentrum [9]. The PULP code uses a similar but slightly different style. Thus, try to maintain the PULP naming convention in order to create readable and maintainable HDL code. Note that there might still be some legacy code which may not be compatible to the naming convention. It is not the goal of this work to re-write and adapt all code to a common naming convention, but the newly developed code should be compatible, and the top-level interfaces to legacy code should be adapted to be compatible to the rest of the system.

4.4 Report

Documentation is an important and often overlooked aspect of engineering. One final report has to be completed within this project.

The common language of engineering is de facto English. Therefore, the final report of the work is preferred to be written in English. Any form of word processing software is allowed for writing the reports, nevertheless the use of LaTeX with Tgif[1] or any other vector drawing software (for block diagrams) is strongly encouraged by the IIS staff.

Final Report The final report has to be presented at the end of the project and a digital copy need to be handed in. Note that this task description is part of your report and has to be attached to your final report.

4.5 Presentation

There will be a presentation (15 min presentation and 5 min Q&A) at the end of this project (usually last Thursday of the semester) in order to present your results to a wider audience. The exact date will be determined towards the end of the work.

[1] Tgif is a simple vector drawing software, quite useful for drawing block diagrams. For further information about Tgif we refer to `http://bourbon.usc.edu:8001/tgif/index.html` and `http://www.dz.ee.ethz.ch/en/information/how-to/drawing-schematics.html`.

References

[1] PULP home page: http://pulp.ethz.ch

[2] Opencores website: http://opencores.org

[3] Or10n project website: http://iis-projects.ee.ethz.ch/index.php/Ultra-low_power_processor_design

[4] The RISC-V website: http://riscv.org

[5] The LowRISC website: http://www.lowrisc.org

[6] The FreeRTOS website: http://www.freertos.org

[7] The Sel4 website: https://sel4.systems

[8] The ORconf2015 website: openrisc.io/orconf

[9] The EDA www page (ETH Zurich internal) http://eda.ee.ethz.ch and VHDL naming conventions: http://eda.ee.ethz.ch/index.php/Naming_Conventions

[10] H. Kaeslin. "Top-Down Digital VLSI Design, 1st Edition From Architectures to Gate-Level Circuits and FPGAs". *Morgan Kaufmann*, 2014.

Zurich, September 15, 2015 Prof. Dr. Luca Benini

Appendix B

Declaration of Originality

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

This is a sample title

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
First	Student
Second	Student

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 01.01.2000	First Student Signature
	Second Student Signature

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Appendix C

Datasets

C.1. More Evaluation Results

C.2. Algorithms / Tables

C.3. Boot code

```
1  // TO BE COMPILED ONLY ONCE
2
3
4  #include <spi.h>
5  #include <gpio.h>
6  #include <uart.h>
7  #include <utils.h>
8  #include <pulpino.h>
9
10 const char g_numbers[] = {
11                            '0', '1', '2', '3', '4', '5', '6', '7',
12                            '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'
13                          };
14
15 int check_spi_flash();
16 void load_block(unsigned int addr, unsigned int len, int* dest);
17 void uart_send_block_done(unsigned int i);
18 void jump_and_start(volatile int *ptr);
19
20 int main()
```

```
21 {
22   /* sets direction for SPI master pins with only one CS */
23   spi_setup_master(1);
24   uart_set_cfg(0, 1);
25
26   for (int i = 0; i < 3000; i++) {
27     //wait some time to have proper power up of external flash
28     #ifdef __riscv__
29         asm volatile ("nop");
30     #else
31         asm volatile ("l.nop");
32     #endif
33   }
34
35   /* divide sys clock by 4 */
36   *(volatile int*) (SPI_REG_CLKDIV) = 0x4;
37
38   if (check_spi_flash()) {
39     uart_send("ERROR: Spansion SPI flash not found\n", 36);
40     while (1);
41   }
42
43
44   uart_send("Loading from SPI\n", 17);
45
46   // sends write enable command
47   spi_setup_cmd_addr(0x06, 8, 0, 0);
48   spi_set_datalen(0);
49   spi_start_transaction(SPI_CMD_WR, SPI_CSN0);
50   while ((spi_get_status() & 0xFFFF) != 1);
51
52   // enables QPI
53   // cmd 0x71 write any register
54   spi_setup_cmd_addr(0x71, 8, 0x80000348, 32);
55   spi_set_datalen(0);
56   spi_start_transaction(SPI_CMD_WR, SPI_CSN0);
57   while ((spi_get_status() & 0xFFFF) != 1);
58
59   //-----------------------------------------------------------------
60   // Read header
61   //-----------------------------------------------------------------
62
63   int header_ptr[8];
```

```
64    int addr = 0;
65
66    spi_setup_dummy(8, 0);
67
68    // cmd 0xEB fast read, needs 8 dummy cycles
69    spi_setup_cmd_addr(0xEB, 8, ((addr << 8) & 0xFFFFFF00), 32);
70    spi_set_datalen(8 * 32);
71    spi_start_transaction(SPI_CMD_QRD, SPI_CSN0);
72    spi_read_fifo(header_ptr, 8 * 32);
73
74    int instr_start = header_ptr[0];
75    int *instr = (int *) header_ptr[1];
76    int instr_size =  header_ptr[2];
77    int instr_blocks = header_ptr[3];
78
79    int data_start = header_ptr[4];
80    int *data = (int *) header_ptr[5];
81    int data_size = header_ptr[6];
82    int data_blocks = header_ptr[7];
83
84    //-----------------------------------------------------------------
85    // Read Instruction RAM
86    //-----------------------------------------------------------------
87
88    uart_send("Copying Instructions\n", 21);
89
90    addr = instr_start;
91    spi_setup_dummy(8, 0);
92    for (int i = 0; i < instr_blocks; i++) { //reads 16 4KB blocks
93      // cmd 0xEB fast read, needs 8 dummy cycles
94      spi_setup_cmd_addr(0xEB, 8, ((addr << 8) & 0xFFFFFF00), 32);
95      spi_set_datalen(32768);
96      spi_start_transaction(SPI_CMD_QRD, SPI_CSN0);
97      spi_read_fifo(instr, 32768);
98
99      instr += 0x400;  // new address = old address + 1024 words
100     addr  += 0x1000; // new address = old address + 4KB
101
102     uart_send_block_done(i);
103   }
104
105   while ((spi_get_status() & 0xFFFF) != 1);
106
```

```
107     //-----------------------------------------------------------------
108     // Read Data RAM
109     //-----------------------------------------------------------------
110
111     uart_send("Copying Data\n", 13);
112
113     uart_wait_tx_done();
114     addr = data_start;
115     spi_setup_dummy(8, 0);
116     for (int i = 0; i < data_blocks; i++) { //reads 16 4KB blocks
117       // cmd 0xEB fast read, needs 8 dummy cycles
118       spi_setup_cmd_addr(0xEB, 8, ((addr << 8) & 0xFFFFFF00), 32);
119       spi_set_datalen(32768);
120       spi_start_transaction(SPI_CMD_QRD, SPI_CSN0);
121       spi_read_fifo(data, 32768);
122
123       data += 0x400;  // new address = old address + 1024 words
124       addr += 0x1000; // new address = old address + 4KB
125
126       uart_send_block_done(i);
127     }
128
129     uart_send("Done, jumping to Instruction RAM.\n", 34);
130
131     uart_wait_tx_done();
132
133     //-----------------------------------------------------------------
134     // Done jump to main program
135     //-----------------------------------------------------------------
136
137     //jump to program start address (instruction base address)
138     jump_and_start((volatile int *)(INSTR_RAM_START_ADDR));
139 }
140
141 int check_spi_flash() {
142     int err = 0;
143     int rd_id[2];
144
145     // reads flash ID
146     spi_setup_cmd_addr(0x9F, 8, 0, 0);
147     spi_set_datalen(64);
148     spi_setup_dummy(0, 0);
149     spi_start_transaction(SPI_CMD_RD, SPI_CSN0);
```

```
150   spi_read_fifo(rd_id, 64);
151
152
153   // id should be 0x0102194D
154   if (((rd_id[0] >> 24) & 0xFF) != 0x01)
155     err++;
156
157   // check flash model is 128MB or 256MB 1.8V
158   if ( (((rd_id[0] >> 8) & 0xFFFF) != 0x0219) &&
159        (((rd_id[0] >> 8) & 0xFFFF) != 0x2018) )
160     err++;
161
162   return err;
163 }
164
165 void load_block(unsigned int addr, unsigned int len, int* dest) {
166   // cmd 0xEB fast read, needs 8 dummy cycles
167   spi_setup_cmd_addr(0xEB, 8, ((addr << 8) & 0xFFFFFF00), 32);
168   spi_set_datalen(len);
169   spi_start_transaction(SPI_CMD_QRD, SPI_CSN0);
170   spi_read_fifo(dest, len);
171 }
172
173 void jump_and_start(volatile int *ptr)
174 {
175 #ifdef __riscv__
176   asm("jalr x0, %0\n"
177       "nop\n"
178       "nop\n"
179       "nop\n"
180       : : "r" (ptr) );
181 #else
182   asm("l.jr\t%0\n"
183       "l.nop\n"
184       "l.nop\n"
185       "l.nop\n"
186       : : "r" (ptr) );
187 #endif
188 }
189
190 void uart_send_block_done(unsigned int i) {
191   unsigned int low  = i & 0xF;
192   unsigned int high = i >>  4; // /16
```

```
193
194     uart_send("Block␣", 6);
195
196     uart_send(&g_numbers[high], 1);
197     uart_send(&g_numbers[low], 1);
198
199     uart_send("␣done\n", 6);
200
201     uart_wait_tx_done();
202 }
```

Appendix D

ASIC Datasheet (Imperio)

Contents

D.1. Features **53**
D.2. Description **54**
D.3. Packaging **54**
D.4. Bonding Diagram **54**
D.5. Pin Map **56**
D.6. Pin Description **56**
D.7. Floorplan **57**
D.8. Pad Configuration **59**
D.9. Interface Description **59**
D.10.Register Map **59**
D.11.Operation Modes **59**
D.11.1. Functional Modes 59
D.11.2. Test Modes 59
D.12.Electrical Specifications **61**
D.12.1. Recommended Operating Regions 61

D.1. Features

- RISC-V 32-bit architecture.
  - 31 x 32-bit General Purpose Registers
  - Support for RV32C and partial support for M standard extension

  - Support for hardware loops, post incremental load and stores and SIMD packed instructions
- 64 kByte RAM (32 KByte Data and Instruction)
- 2 kByte boot ROM
- Integrated FLL that offers variable speed from 0-1.2 GHz
- JTAG Debug Interface
- 19 GPIOs
- Peripheral Features
  - Two 32-bit counter
  - UART
  - I2C
  - SPI Master
  - SPI Slave
- Full scan-able design with 7 scan chains.

D.2. Description

Imperio is the ASIC implementation of PULPino using a RISC-V core. A high-level block diagram can be seen in figure 4.1. The final mission (once the chip passed testing) of the ASIC is to be employed on a PCB. It therefore was mission critical to have an efficient way of generating the (high) clock, this is achieved by an on-chip FLL. Two 32 kByte on-chip RAMs supply the core with data and instructions. A boot ROM loads the necessary data (through SPI) from an off-chip memory at startup.

D.3. Packaging

A QFN40 package is used with 6 power pins.

D.4. Bonding Diagram

The extended power (ep) bonding is used for this chip. The clock pin is placed on the recommended position in order to facilitate the already manufactured tester boards in use at IIS.

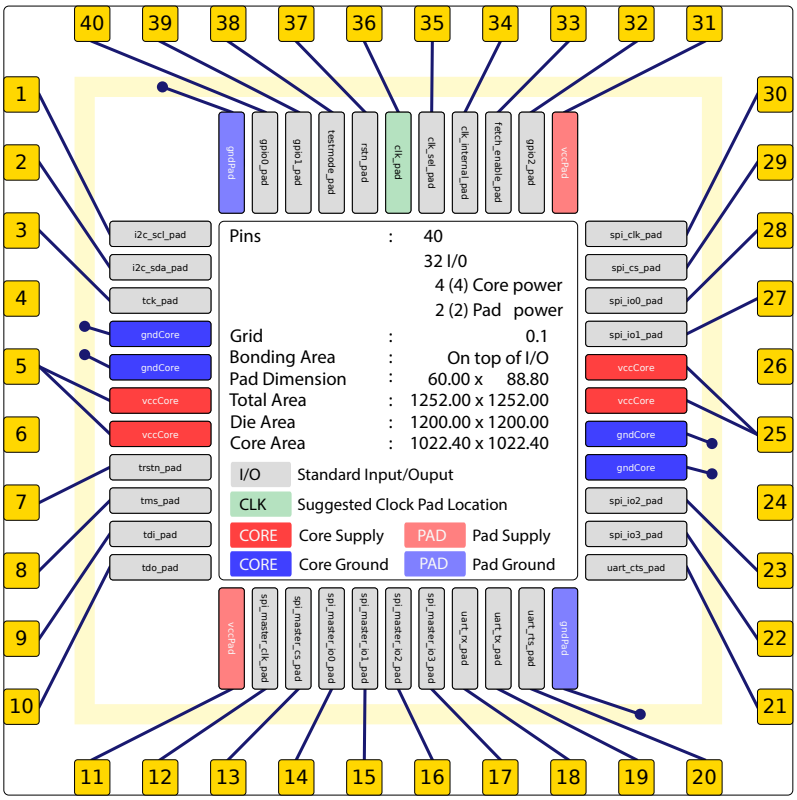


Figure D.1.: Bonding diagram.

D.5. Pin Map

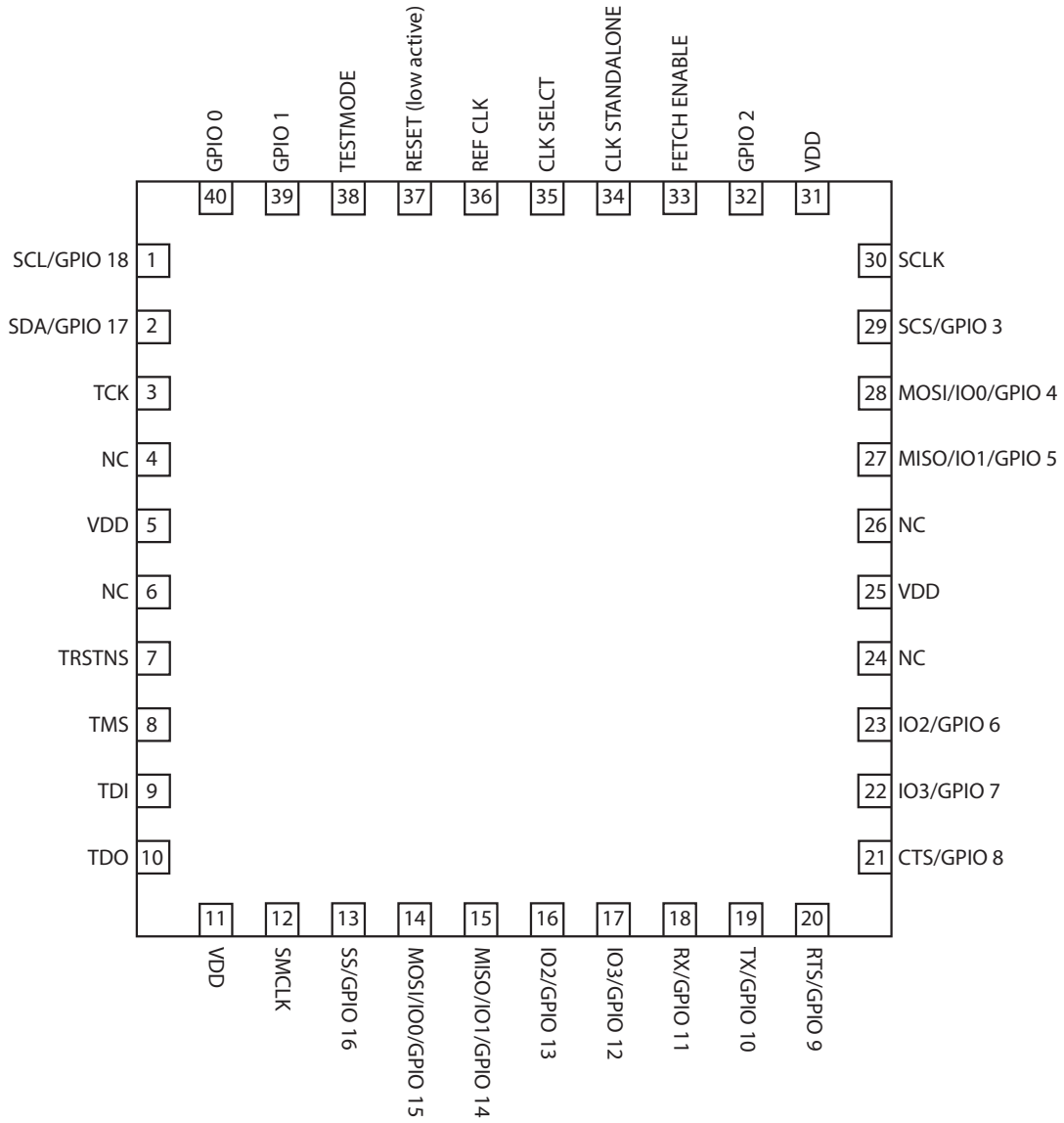


Figure D.2.: Imperio pinout (QFN40)

D.6. Pin Description

Table D.1 lists all pins along with their usage, second and testmode function. Testmode is achieved by pulling the `pad_testmode_i` (38) pin high. Secondary function can be con-

figured by writing the corresponding register in the PULPino APB peripheral, see 4.4.3.

D.7. Floorplan

The floorplan with macro block positioning is depicted in figure D.3.

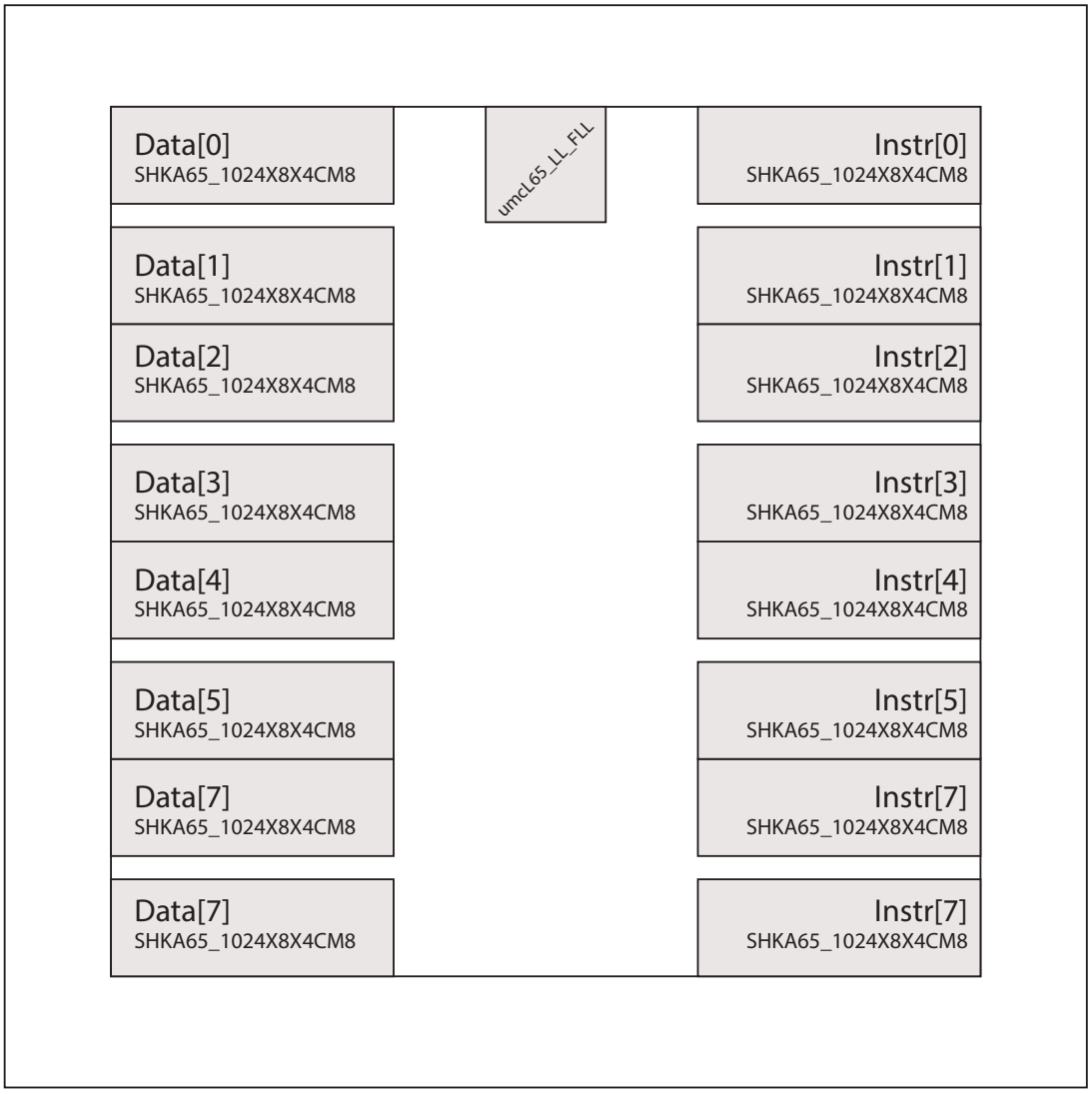


Figure D.3.: Floorplan with macro block positioning

Table D.1.: Pin Description. (Power, IO, Clock, Reset, Test)

Pin No.	**Pin Name**	**Primary function**	**Sec. Func.**	**Testmode**
5, 25		VDD Core		
11, 31		VDD Pad		
1	pad_scl_io	I2C SCL	GPIO 18	-
2	pad_sda_io	I2C SDA	GPIO 17	-
3	pad_tck_i	TCK (JTAG)	-	Scan Clk
7	pad_trstn_i	TRSTN (JTAG)	-	Test RST
8	pad_tms_i	TMS (JTAG)	-	SI 7
9	pad_tdi_i	TDI (JTAG)	-	-
10	pad_tdo_o	TDO (JTAG) SCL	-	SO 7
12	pad_msclk_o	SPI Master Clock	-	-
13	pad_mcs_io	SPI Master SS	GPIO 16	SO 5
14	pad_mio0_io	SPI Master MOSI/IO0	GPIO 15	SO 4
15	pad_mio1_io	SPI Master MISO/IO1	GPIO 14	SI 4
16	pad_mio2_io	SPI Master IO2	GPIO 13	SI 5
17	pad_mio3_io	SPI Master IO3	GPIO 12	-
18	pad_rx_i	UART RX	GPIO 11	-
19	pad_tx_o	UART TX	GPIO 10	SO 1
20	pad_rts_o	UART RTS	GPIO 9	SO 6
21	pad_cts_i	UART CTS	GPIO 8	SI 6
22	pad_sio3_io	SPI Slave IO3	GPIO 7	SI 3
23	pad_sio2_io	SPI Slave IO2	GPIO 6	SO 3
27	pad_sio1_io	SPI Slave MISO/IO1	GPIO 5	SO 2
28	pad_sio0_io	SPI Slave MOSI/IO0	GPIO 4	SI 2
29	pad_scs_io	SPI Slave CS	GPIO 3	Scan Clk
30	pad_ssclk_i	SPI Slave Clock	-	-
32	pad_gpio_io[2]	GPIO 2	-	-
33	pad_fetch_enable_i	fetch enable	-	-
34	pad_clk_standalone_i	Clock Standalone	-	-
35	pad_clk_sel_i	Clock Select	-	-
36	pad_clk_i	Clock	-	Scan Clk
37	pad_rstn_i	reset (low active)	-	-
38	pad_testmode_i	Testmode enable	-	-
39	pad_gpio_io[1]	GPIO 1	-	SI 1
40	pad_gpio_io[0]	GPIO 0	-	Scan En

D.8. Pad Configuration

D.9. Interface Description

D.10. Register Map

D.11. Operation Modes

Imperio distinct functional and test mode. Testmode is achieved by asserting pin `pad_testmode_i` (38).

D.11.1. Functional Modes

In functional mode most of the pads serve either as specialized I/O or as GPIO, see table D.1 for details on the pin configuration.

D.11.2. Test Modes

When the chip is put into testmode the pads are configured according to table D.1. Especially the output enable signals of the pad are configured accordingly, so that all Scan Ins and the Scan Enable signal are inputs and the Scan Outs are configured as outputs. All clock gates are disabled.

A special remark on the `pad_fetch_enable_i` pin. The core does not start fetching instructions until the pin gets asserted. This is meant to accurately time the start times of the execution sequence on the tester. For functional mode this pin should be pulled up.

Table D.2.: Pad configuration and corresponding reset values.

Pin	**Primary function**						**GPIO**						**Test**
	OE	PD	PU	SMT	SR	DS	OE	PD	PU	SMT	SR	DS	OE
1	*	0	0	0	0	0	0	0	0	0	0	0	0
2	*	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	-	-	-	-	-	-	0
7	0	0	0	0	0	0	-	-	-	-	-	-	0
8	0	0	0	0	0	0	-	-	-	-	-	-	0
9	0	0	0	0	0	0	-	-	-	-	-	-	0
10	1	0	0	0	0	0	-	-	-	-	-	-	1
12	0	0	0	0	0	0	-	-	-	-	-	-	0
13	1	0	0	0	0	0	0	0	0	0	0	0	1
14	*	0	0	0	0	0	0	0	0	0	0	0	1
15	*	0	0	0	0	0	0	0	0	0	0	0	0
16	*	0	0	0	0	0	0	0	0	0	0	0	0
17	*	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0
19	1	0	0	0	0	0	0	0	0	0	0	0	1
20	0	0	0	0	0	0	0	0	0	0	0	0	1
21	1	0	0	0	0	0	0	0	0	0	0	0	0
22	*	0	0	0	0	0	0	0	0	0	0	0	0
23	*	0	0	0	0	0	0	0	0	0	0	0	1
27	*	0	0	0	0	0	0	0	0	0	0	0	1
28	*	0	0	0	0	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	-	-	-	-	-	-	0
32	*	0	0	0	0	0	-	-	-	-	-	-	0
33	0	0	0	0	0	0	-	-	-	-	-	-	0
34	0	0	0	0	0	0	-	-	-	-	-	-	0
35	0	0	0	0	0	0	-	-	-	-	-	-	0
36	0	0	0	0	0	0	-	-	-	-	-	-	0
37	0	0	0	0	0	0	-	-	-	-	-	-	0
38	0	0	0	0	0	0	-	-	-	-	-	-	0
39	0	0	0	0	0	0	-	-	-	-	-	-	0
40	0	0	0	0	0	0	-	-	-	-	-	-	0

D.12. Electrical Specifications

Table D.3.: DC characteristics [5]:

Symbol	Description	Min.	Max.	Unit
V_{Il}	Input low voltage	-	0.7	V
V_{Ih}	Input high voltage	1.7	-	V
V_{Ol}	Output low voltage	-	0.4	V
V_{Oh}	Output high voltage	1.85	-	V
Pull-up	Pull-up-resistor	53	53	k Ω
Pull-down	Pull-down-resistor	53	53	k Ω

D.12.1. Recommended Operating Regions

Table D.4.: Recommended Operating Conditions [5]

Symbol	Description	Min.	Typ.	Max.	Unit
VDD	Core power supply	1.08	1.2	1.32	V
VDDIO	IO power supply	2.25	2.5	2.75	V
T_J	Operating junction temperature	-40	25	125	°C

Appendix E

PULPino environment

E.0.1. Requirements

You should have at least the following programs and tools installed:

- **git** $\geq$ 2.4.8
- **ModelSim** in a version $\geq$ 10.2c
- **CMake** $\geq$ 2.8.0, if you want to use Ninja (a build framework compareable to Make) you need a CMake version that supports this ($\geq$ 3.1.0)
- **riscv-toolchain** if you do not want to use the advanced interrupt features you can go with the official version maintained by UCB[1]. Otherwise I recommend using the version maintained at ETHZ.

As the C shell is the de-facto standard in engineering I would highly recommend you to use it. This setup has not been tested with any other shell like for example bash.

E.0.2. Directory Structure

```
/
├── README ..................... README file with a quick start guide.
├── ips_list.txt ............... Listing of all ip cores that are used by the project
├── update-ips.py .............. Updates all IPs according to ip_list.txt
├── ci ......................... Folder containing scripts used by continuous integration
│   ├── setup.csh .............. Setup scripts that initialize the CI environment
│   └── *.csh .................. Runs the corresponding tests
└── doc ........................ Contains the actual source files of your report.
```

[1] as

- datasheet ..... PULPino
- api ..... Driver api description
- report ..... Contains this report
- fpga ..... This folder contains everything special needed for FPGA emulation
- ips ..... Contains the IP cores used for PULPino.
  - adv_dbg_if ..... Source of the Advanced Debug Unit (ADU)
  - apb ..... APB peripherals
    - apb_event_unit ..... APB Event and Interrupt Unit
    - apb_fll_if ..... FLL configuration interface
    - apb_gpio ..... APB General Purpose I/O
    - apb_i2c ..... APB I2C
    - apb_pulpino ..... APB SoC Control for PULPino
    - apb_spi_master ..... APB SPI Master
    - apb_timer ..... APB Timer
    - apb_uart ..... TL16C550 compatible UART implementation.
  - axi ..... AXI interconnect and AXI peripherals
    - axi2apb ..... AXI to APB adapter
    - axi_mem_if_DP ..... AXI to memory adapter
    - axi_node ..... AXI Bus
    - axi_slice_dc ..... Slices the AXI Bus and inserts registers
    - axi_spi_slave ..... AXI SPI Slave
  - or10n ..... Source of the RISCV core
  - riscv ..... Source of the OpenRISC core
  - scm ..... Standard Cell Memories
  - scripts ..... Scripts used for compiling the corresponding IPs
- rtl ..... Contains glue logic and project specific RTL.
  - components ..... Technology dependent standard cells and their behavioral counterparts
  - scripts ..... Scripts used for compiling the corresponding RTL
  - top.sv ..... Top level entity
- sw ..... Contains software for PULPino.
  - apps ..... Here is the actual source code for every application.
  - libs ..... Libraries and HALs for PULPino
  - ref ..... Contains linker script and c runtime initialization code.
  - utils ..... Contains scripts that are needed for simulation.
- cmake_configure.or1k.llvm.sh ... Sample script to configure environment for LLVM and the OpenRISC core.
- cmake_configure.riscv.gcc.sh ... Sample script to configure environment for GCC and the RISC-V core.
- tb ..... Contains RTL testbench and helper packages needed.
- vsim ..... Contains scripts needed for simulation.
  - scripts ..... Compile scripts - invoked by CMake.

└ `tcl_files` .The actual scripts location to run the CMake target with desired configuration.

E.1. Getting started

PULPino is organized to use different git repositories for every IP. Start by checking out the main repository.

```
sh~> git clone git@iis-git.ee.ethz.ch:pulp-project/pulpino.git
```

By invoking the `update_ips.py` Python script you start pulling all remaining IPs specified in the `ips_list.txt` file.

```
sh~> ./update-ips.py
```

After the script has successfully pulled all IPs you can start running simulations. The whole PULPino project uses CMake as a meta build framework and make or ninja respectively as the main build framework. CMake supports out of tree builds. Create a folder somewhere in the project e.g. in the `sw/` subdirectory and copy the `cmake\_configure.*.*sh` (depending on the setup you'd like to run) into the newly created build directory.

```
sh~sw/> mkdir build
sh~sw/> cp cmake_configure.riscv.gcc.sh build/
```

Change into that build directory and executed the shell script.

```
sh~sw/> cd build
sh~sw/build> ./cmake_configure.riscv.gcc.sh build
```

If you are at ETH you probably need to prefix the shell script with `riscv`. Invoking that script starts the CMake configuration. It sets up all make/ninja targets and creates the necessary directory structure to run simulations. Before you can start you have to compile the RTL sources with ModelSim. We provide the following target to do so:

```
sh~sw/build> make vcompile
```

Based on how you initialized your build framework it builds the correct core automatically (depending on whether you used `cmake_configure.riscv.gcc.sh` or `cmake_configure.or1k.llvm.sh`).

The final step in running a simulation comprises of invoking the program you want to run. Have a look at the `sw/apps` folder. You can see that every program is contained in its own uniquely named directory. CMake automatically creates a simulation target

for each of those programs. For example if you want to run the helloworld program just type:

```
sh~sw/build> make helloworld.vsim
```

You are as well given the possibility to run ModelSim without the GUI. To do so invoke your program with **.vsimc**. For example:

```
sh~sw/build> make helloworld.vsimc
```

To use Ninja instead of make have a look at the configuration scripts. After you have reconfigured your environment with the new script just call ninja instead of make:

```
sh~sw/build> ninja helloworld.vsim
```

The call to *.vsim generates all files necessary to run the simulation inside your current build folder. It therefore copies the application structure from the apps folder. All outputfiles are written to the corresponding app folder within the build folder. What maybe especially useful for debugging is the output of the coretracer. Every instruction that is executed by the core is put into the TODO:NAme file.

E.1.1. Utilities

As soon as you get more comfortable with the build environment you are probably going to want start developing application. In order to support you and make your life a bit easier we have implemented some utility targets.

To disassemble a program call the *.read target:

```
sh~sw/build> make helloworld.read
```

Sometimes it is difficult to have an overview of the cores control flow. We support the possibility to annotate the core's tracer output with the section name from the disassembly. This prints some sort of stack trace to a separate file called TODO:filename inside the built folder. That feature can be particularly useful if you want to debug highly nested function calls. To annotate the core's trace call:

```
sh~sw/build> make helloworld.annotate
```

E.1.2. Imperio related CMake targets

For now I've only mentioned CMake targets that run for both, Imperio and PULPino, I will now give an overview about targets that are solely available to Imperio. They are not of interest if you do not have post-synthesis or post-layout netlists.

To compile post synthesis (ps) and post layout (pl) netlists along with all functional models call:

```
sh~sw/build> make vcompile.ps
sh~sw/build> make vcompile.pl
```

In postlayout simulation the netlist gets back annotated with timing information from Cadence Encounter. Consequently simulation takes longer to finish.

Now you are presented with different options depending on how you would like to load the program (either from external flash through the boot ROM or via the SPI Slave), this can either be one of the two options:

```
sh~sw/build> make helloworld.boot.ps
sh~sw/build> make helloworld.spi.ps
```

The same applies to post layout simulation:

```
sh~sw/build> make helloworld.boot.pl
sh~sw/build> make helloworld.spi.pl
```

Finally the utility target for running power simulation is

```
sh~sw/build> make helloworld.power
```

This loads the program via SPI and starts the VCD Dump of ModelSim as soon as GPIO 1 has been asserted by the program.

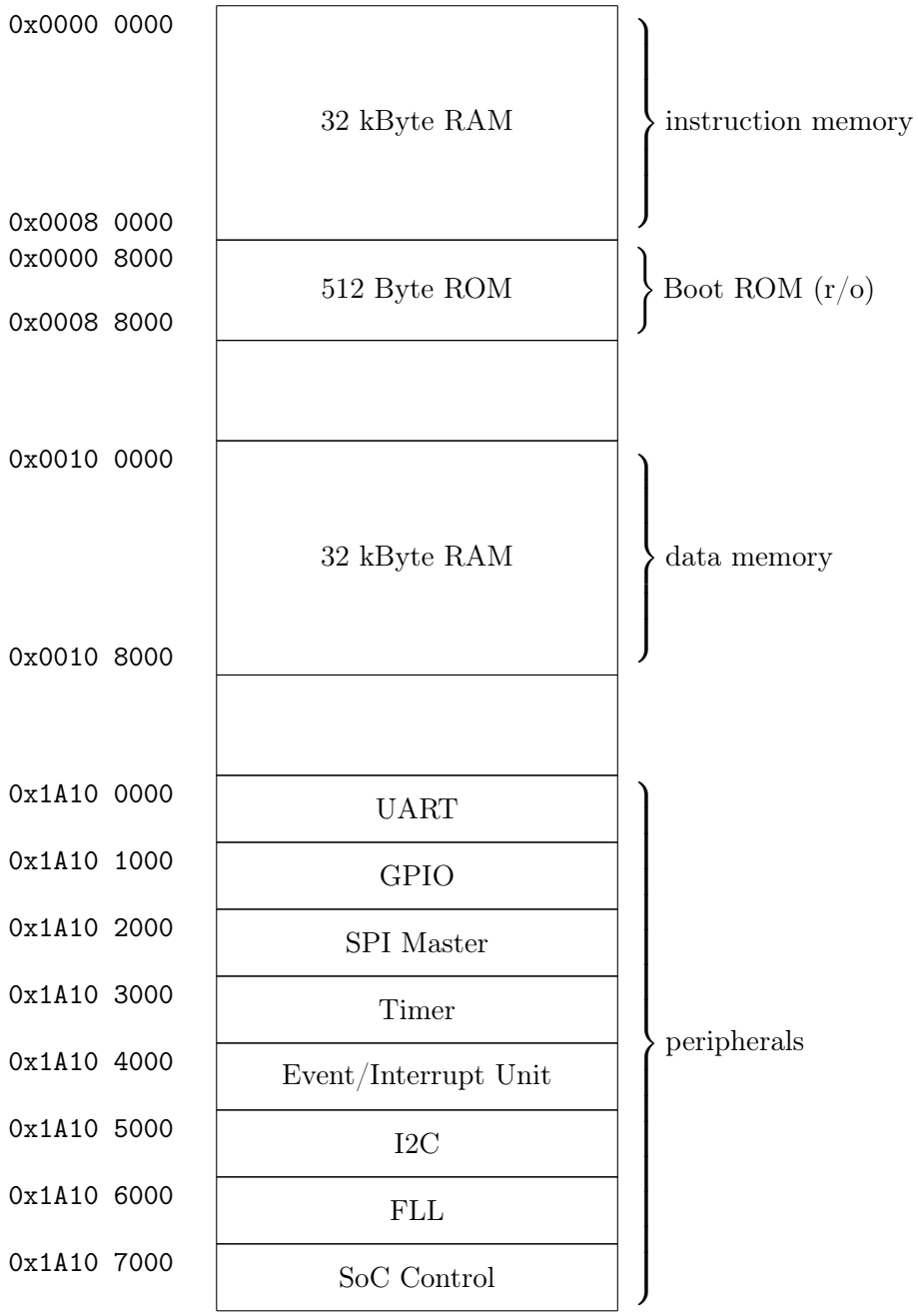
Appendix F

Memory and Register Map

The following section is an ehaustive listing of all registers available in the current implementation. All registers are read/write except for those where it is explicetly stated otherwise.

This sections begins with an overview of the system's memory map and concludes with the register listing. It is ment to be some sort of reference card for the application developer. For any descirption about highlevel functionality please refer to the section describing the architecture.

F.1. Memory Map



F.2. Register Map

F.2.1. UART

F.2.2. GPIO

F.2.3. SPI Master

F.2.4. Timer

F.2.5. Event/Interrupt Unit

F.2.6. I2C

SPR Address: `0x1A10 5000 - 0x1A10 5018`
Reset Value: `0x0000_0000, 0x0000_0000, 0x0000_0000, 0x0000_0000`

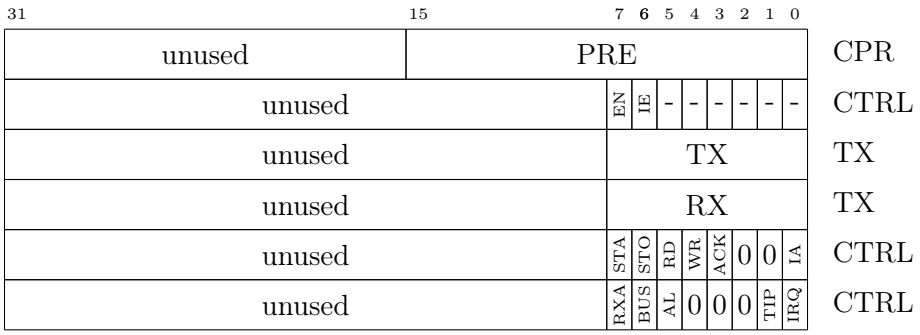


Figure F.1.: I2C Registers

- **CPR (Clock Prescale Register)**: Sets the clock prescaler by the value in PRE to achieve the desired I2C clock by dividing the current system clock by the given factor.
- **CTRL (Control Register):**
  - Bit 7: EN (Enable). Enable the I2C peripheral.
  - Bit 6: IE (Interrupt enable). Enable interrupts.
  - Bit 5 - 0: Are currently not in use, thy can be written and read but do not have any effect.
- **TX (Transmit Register):** Transmit register.
- **RX (Receive Register):** Receive register.

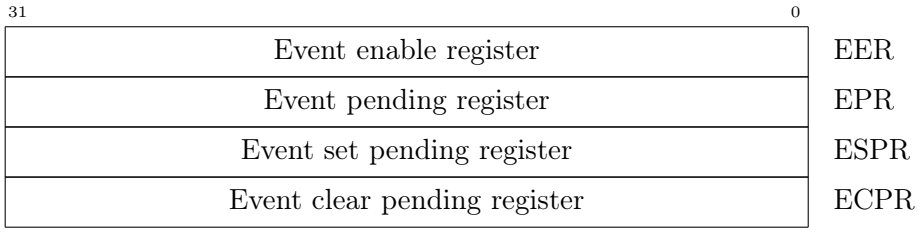
- **CMD (Command Register):** The command register is cleared when command is done or arbitration is lost.
  - Bit 7: (STA) Send start bit.
  - Bit 6: (STO) Send stop bit.
  - Bit 5: (RD) Read from bus.
  - Bit 4: (WR) Write to bus.
  - Bit 3: (ACK) Acknowledge received data.
  - Bit 2 - 1: reserved, set to 0.
  - Bit 0: IA (Interrupt Acknowldge): Set to one to acknowledge interrupt. Cleared when transmission is done or arbitration is lost.
- **STATUS (Status Register):**
  - Bit 7: (RXA) Acknowledge from sent data.
  - Bit 6: (BUS) Bus is busy.
  - Bit 5: (AL) Arbitration lost.
  - Bit 4-2: reserved, set to 0.
  - Bit 1: (TIP) Transfer in progress.
  - Bit 0: (IRQ) Interrupt received. This flag is always set when transmission has finished or bus arpitration was lostm, regardless of whether interrupts are enabled or not. This flag can possibly polled and is cleared by writing 1 to the IA command register.

F.2.7. FLL

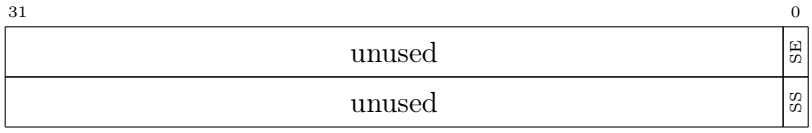
F.2.8. SoC Control

31 … 0	
Interrupt enable register	IER
Interrupt pending register	IPR
Interrupt set pending register	ISPR
Interrupt clear pending register	ICPR

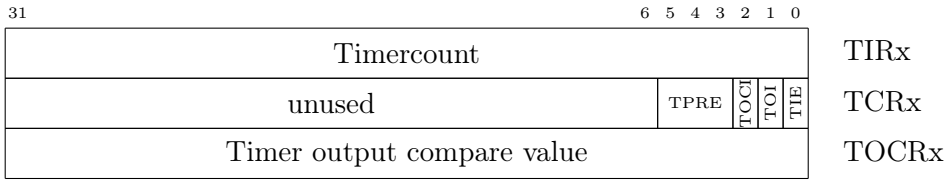
-


31
0
Event enable register
EER
Event pending register
EPR
Event set pending register
ESPR
Event clear pending register
ECPR

-


31
0
unused
SE
SCR
unused
SS
SSR

-


31
6 5 4 3 2 1 0
Timercount
TIRx
unused
TPRE
TOCI
TOI
TIE
TCRx
Timer output compare value
TOCRx

-

Bibliography

[1] R. Andri and M. Baer, "Implementation and optimization of the openrisc processor," IIS, ETH Zurich, Tech. Rep., February 2014.

[2] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa, version 2.0," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54, May 2014. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[3] ——, "The risc-v compressed instruction set manual, version 1.9," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-209, Nov 2015. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-209.html

[4] H. Kaeslin, *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication*. Cambridge University Press, Apr. 2008.

[5] UMC, "Um065gioll25mvir_b umc 65nm low-k 1.8v/2.5v/3.3v low leakage rvt boac in-line io library databook," September 2013, i/O Library Documentation.

[6] Wikipedia, "Isaac newton — wikipedia, the free encyclopedia," 2012, [Online; accessed 1-October-2012]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Isaac_Newton&oldid=514997436