FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Design of System On Chip with RISC-V processor for USI graphical pen controller
Student:	Bc. Martin Stahl
Supervisor:	Ing. Tomáš Novák
Study program:	Informatics
Branch / specialization:	Design and Programming of Embedded Systems
Department:	Department of Digital Design
Validity:	until the end of summer semester 2023/2024

Instructions

1. Familiarize yourself with USI Pen Controller SoC requirements, with the existing platform based on CoolRISC and with available IP blocks.
2. Analyze differences between RISC-V and CoolRISC CPU platform and prepare system level design for System On Chip controller based on RISC V.
3. Implement the design at the RTL level and demonstrate its correct operation.
4. Validate the designed system on the FPGA circuit.
5. Estimate the power consumption and compare it with the CoolRISC processor implementation.

Electronically approved by prof. Ing. Hana Kubátová, CSc. on 3 January 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Design of System On Chip with RISC-V processor for USI graphical pen controller

Bc. Martin Stahl

Department of Digital Design
Supervisor: Ing. Tomáš Novák

May 3, 2023

Acknowledgements

I would like to thank my supervisor Ing. Tomáš Novák for his guidance and project leadership during my work on this Master's thesis. Furthermore, I would like to thank my colleagues Ing. Tomáš Tuček, Ing. Michal Skiba, Ing. Radek Hájek, and Ing. Petr Jašek, for their great help and expertise. Lastly, I would like to thank my partner Domen Stropnik for his great mental support during my entire Master's Degree studies.

Declaration

Citation of this thesis

Stahl, Martin. *Design of System On Chip with RISC-V processor for USI graphical pen controller.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

Tato diplomová práce se zabývá RTL návrhem a implementací systému systému na čipu na procesorové platformě RISC-V pro USI ovladač grafického pera. Současný SoC ovladače pera založeného na CoolRISC je analyzován a na základě této analýzy je vytvořen systémový návrh pro nové SoC ovladače pera založeného na RISC-V. RTL návrh nového SoC je implementován do 180 nm technologie a jeho systémová spotřeba energie je měřena v simulaci a poté porovnána se stávajícím systémem na bázi CoolRISC. Práce se také zabývá technickým srovnáním procesorových platforem CoolRISC a RISC-V.

Klíčová slova Systém na čipu, RISC-V, CoolRISC, systémový návrh, RTL implementace, FPGA validace, analýza spotřeby

Abstract

This diploma thesis covers the RTL design and implementation of System On Chip based on the RISC-V processor platform for USI graphical pen controller. The current CoolRISC-based pen controller SoC is analysed and based on this analysis new system design for the RISC-V-based pen controller SoC is created. The RTL design of the new SoC is implemented into 180 nm technology, and its system power consumption is measured in simulation and then compared to the existing CoolRISC-based system. The thesis also covers the technical comparison between CoolRISC and RISC-V processor platforms.

Keywords System On Chip, RISC-V, CoolRISC, system design, RTL implementation, FPGA validation, power analysis

Contents

List of Figures

List of Tables

Introduction

Modern pens for graphic tablets are not just pieces of plastic with a hard tip which makes contact with the tablet to make the contact for drawing in resistive displays. Today most graphic pens are much more complex. They can measure pen pressure, the angle at which the pen is held against the tablet and the precision is far better. They also feature buttons and a battery which requires charging.

Furthermore, these graphic pens shall work according to the Universal Stylus Initiative (USI) which defines the industry standards for graphic pens communication protocols. This is to enable these pens to work with a large variety of touch screens in devices such as phones, tablets, and other platforms.

These functionalities require a digital controller module which also needs to be ultra-low-power to last long on a very small battery. ASICentrum s.r.o has been developing pen controller SoCs for these graphical pens for several years. They have been using the CoolRISC CPU platform as the brain of the controller, however, it is only an 8-bit CPU and it is increasingly becoming a bottleneck in the system's performance.

They want to move to a new CPU platform for better performance but they do not know how much the new CPU is going to affect the overall power consumption of the system. The ultra-low power consumption of the system is a requirement.

They have tasked me with implementing their CoolRISC-based pen controller SoC with a modern RISC-V CPU to simulate and estimate the power consumption of the system with the new CPU and compare it to the existing CoolRISC-based system. Part of this will be the creation of an FPGA environment where the customer can test their software.

The result of this thesis will be the pen controller SoC implemented with the RISC-V CPU platform. It will be simplified in some ways to make the development faster while still keeping the core functionality necessary for valid power comparisons. The basic functionality of the system will be verified and then customer-required functionality will be validated on an FPGA. 180 nm technology will be used for physical implementation which is then going to be used for power consumption simulations.

Part of the analysis will be also the comparison of CoolRISC and RISC-V CPU platforms to highlight the differences, advantages and disadvantages of each platform which go beyond just the power consumption.

Chapter 1

Goal

The goal of this thesis is to analyse the existing CoolRISC-based pen controller SoC and implement it with the RISC-V CPU platform in RTL and estimate the power consumption using simulation measurements of its physical implementation.

The RISC-V CPU, which is going to be used for the new implementation, is the CV32E40P developed by the OpenHW group and is fully open source. The newly implemented system will be simplified in some ways to make the implementation faster while keeping the core functionality necessary for valid power comparisons. The basic functionality of the system will be verified in a test bench simulation environment and then customer-required functionality will be validated on an FPGA. After the system is verified and validated, synthesis and placement into 180 nm technology will be performed. This physical implementation will be used for the power consumption simulations and estimations and the results will be compared with the results from the existing CoolRISC-based system.

The goal of the USI Pen Controller SoC requirements chapter is to analyse the existing CoolRISC-based system to identify the key components and features of the system. These findings will be used to create requirements for the new RISC-V-based implementation called ”PENRISCV”. Those requirements will be used to make a system design for the new implementation. Part of this chapter is also the analysis of both CPU platforms and their comparison.

The goal of the implementation is to use the system design from the previous chapter and based on it implement PENRISCV in RTL.

The goal of the verification and validation is to verify PENRISCV's basic functionality and to check the correct behaviour of the system. Validation on FPGA is intended to check customer-required functionality for software testing.

The goal of physical implementation and power simulations is to implement PENRISCV into 180 nm process technology and then estimate the theoretical power consumption simulated and estimated in selected scenarios. The results of these estimations will be then compared to the results from the CoolRISC-based system.

Chapter 2

USI Pen Controller SoC requirements

In this chapter, I analyse the existing system, which utilized the CoolRISC CR816L CPU core, its features and characteristics. I focus on the digital part of the system (dig_top) because it is the core of the system and the main focus of power comparison. Analogue parts will stay the same in both systems. Then I compare the CoolRISC CPU platform to the RISC-V platform.

Based on the features of the existing system and the differences between both CPU platforms I create requirements for the new system that utilizes a new RISC-V CPU core.

2.1 Existing CoolRISC pen controller SoC

The pen controller is a multi-chip system built into a graphical pen used for drawing on touch screens and graphic tablets. It is the main control unit of the graphical pen.

The main functionality of the pen controller is to transmit and receive signals on its two pentips[1] from the electrostatic field generated by a touch screen or a graphic tablet.

Those signals represent the data communication stream between the tablet and the pen controller. In the real environment, there is a lot of electrostatic noise during communication, therefore, computation algorithms are required for reliable communication. This is managed by the pen controller hardware and software. The pen controller also handles other functionality of the graphical pen such as battery charging.

The pen controller consists of two chips. The first chip is the low-voltage SoC which contains the CPU, memory, power management, peripherals and

[1]A pentip is a wire connected to a pen controller IO pad which is designed for high voltages

low-voltage analogue parts. It is the main control unit of the system which runs the software.

The second chip is the high-voltage pentip driver which contains voltage control logic, DCDC and other high-voltage analogue parts for transmitting and receiving high-voltage signals on the pentips.

These two parts of the system communicate together through a specific interface and a custom SPI. The manufactured implementation consists of both chips connected together in a single package.

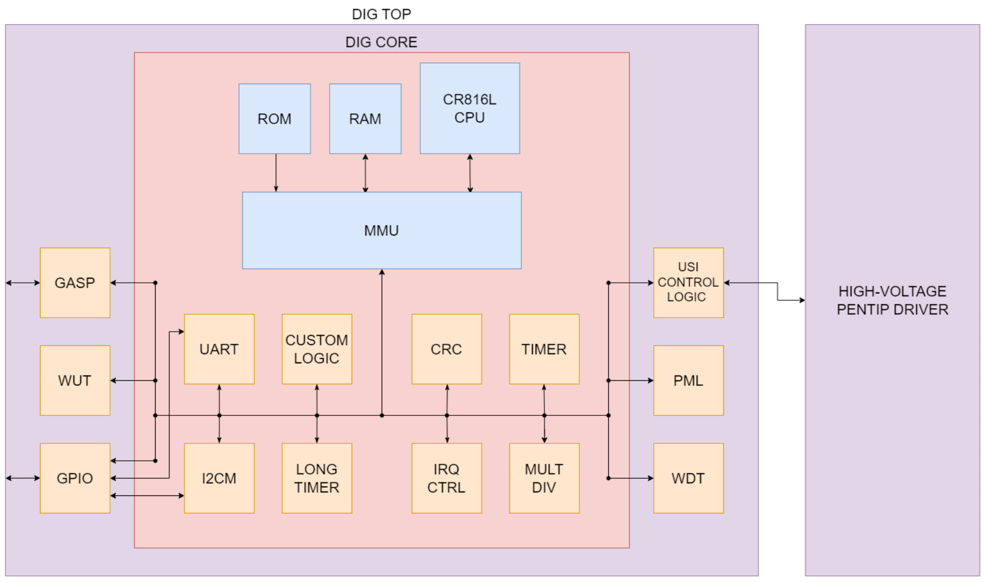


Figure 2.1: CoolRISC SoC block diagram

2.1.1 Block description

Here I briefly described the function of each block in the CoolRISC implementation. Each block is visible in the system block diagram in Figure 2.1.

1. **Central Processing Unit (CPU):** The CR816L CPU core is responsible for running compiled software. It accesses the RAM, ROM and Flash memory through the MMU. Peripherals are accessed through CR816L's data bus. Each peripheral has its own defined address space.

2. **Random Access Memory (RAM):** Memory for runtime data while the software is being executed on the CPU.

3. **Read-Only Memory (ROM):** Contains boot software and other minor procedures needed for the system before the CPU starts fetching from the Flash memory.

4. **Memory Management Unit (MMU):** Serves as an arbitrer of the address space. Multiplexed devices are accessed based on the address from the CPU. Address space for RAM, ROM and peripherals is defined during implementation. It also features an NVM controller for Flash memory.

5. **Power Management Unit (PML):** Is mainly responsible for managing system clock switching, reset synchronization and distribution, reset synchronization from WDT, and system power mode transitions. It also features DCDC control, Switch control, LED control and battery charging control.

6. **WatchDog Timer (WDT):** To prevent deadlock situations a watchdog timer is available to monitor embedded firmware activity. Watchdog is controlled by the CPU and generates a system reset upon a time-out.

7. **Wake-Up Timer (WUT):** 8-bit programmable down counter from 512 us to 2 ms. It is used to switch the running system from SLEEP or POWER-DOWN mode to ACTIVE mode.

8. **General-Purpose Input/Output (GPIO):** Configurable 3-bit input and output pads that can be controlled through software. Contains debouncers and edge/level detectors. Two pads are also used for the I2CM interface. Also manages the selection of UART interface on two GASP pads.

9. **Interrupt Controller:** Control unit to enable, mask and give priority to interrupts which are generated by peripherals.

10. **UART:** Standard UART interface with TXD and RXD interface. The frame structure is: start bit / 8 bits data / 1 stop bit. Parity may be even / odd / no (default). Supported baud rates are 9 600 Bd, 14 400 Bd, 19 200 Bd, 28 800 Bd, 38 400 Bd, 57 600 Bd, 115 200 Bd.

11. **I2CM:** Standard I2CM interface. Implements the standard mode with a baud rate of up to 100 kHz and the Fast mode with a baud rate of up to 400 kHz. It can operate as a single master-receiver or master-transmitter.

12. **Timer:** 16-bit down counter with auto-reload mode or single shoot (stopped when 0x00 value is reached)

13. **CRC:** Hardware acceleration for cyclic redundancy check. Allows to define polynomials for CRC calculation.

14. **Custom Logic:** Hardware acceleration for specific bitwise operations.

15. **Multiplication/Division accelerator (MULTDIV):** Multiplication performs 16-bit x 8-bit operation. The duration is a maximum of 9 clocks. The division performs 16-bit / 8-bit operations. The duration is a maximum of 18 clocks.

16. **USI[2] Control Logic:** Custom control logic IP which implements the USI pen protocol. It sends and receives data from the HV pentip driver.

17. **GASP:** Custom debugging interface.

2.1.2 Power modes

To help the CoolRISC system to save power, several power modes are implemented.

1. **ACTIVE:**
   - All parts of the dig_top are active.
   - CPU is running and executing code.
2. **SLEEP:**
   - CPU is in sleep mode and not executing code.
   - System clock is switched to a lower frequency oscillator.
   - System can be switched back to ACTIVE mode upon interrupt or event or system reset.
3. **POWER-DOWN:**
   - VDD1 power domain is switched off.
   - System clock is switched to a lower frequency oscillator.
   - System can be switched back to ACTIVE mode upon GPIO or Wake-up Timer event or by a system reset.

2.1.3 Reset structure

The SoC has two external negative reset sources, power-on reset and battery charger reset. Internally there are other reset sources such as WDT, GASP and switching off VDD1 domain reset. The PML is responsible for synchronizing these reset sources and distributing them in the system.

[2]Universal Stylus Initiative (USI) defines communication standards for graphic pens

2.1.4 Power domains

The CoolRISC implementation features two power domains to be able to switch the second one off and save power when all functions of the system are not needed at the moment. These domains are VDD0 and VDD1.

- **VDD0** domain stays switched on and powers PML (Power Management Logic), GPIO, Watchdog, Wake-up timer and the USI Control Logic.
- **VDD1** domain is switched off when the system enters POWER-DOWN mode. It stays on during other power modes. It powers the remaining blocks (for example CPU, peripherals, memory, etc.) of the system.

The voltage supplied to both power domains should be between 0.85-1.4 V.

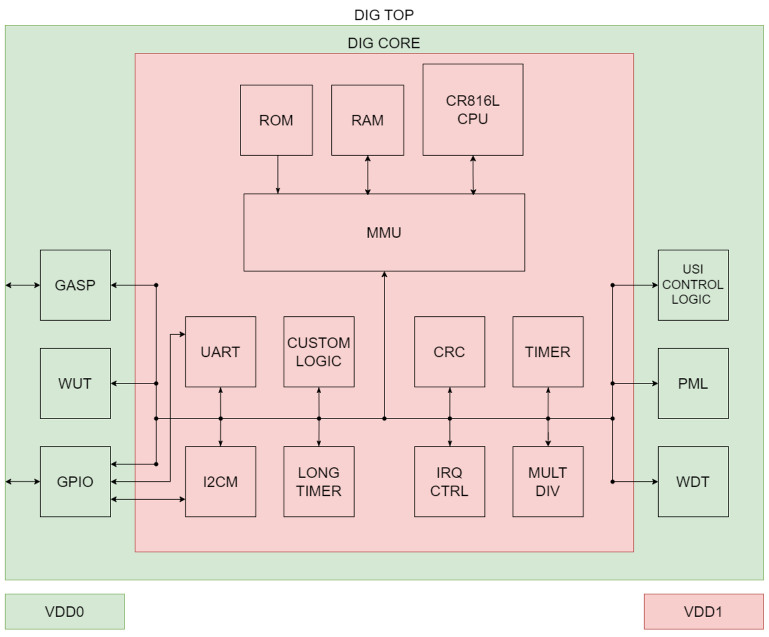


Figure 2.2: CoolRISC SoC power domains

2.1.5 Clock structure

There are three main clock sources in the CoolRISC implementation.

- **RC oscillator:** Main clock source which is always running. It can provide 2 MHz, 4 MHz and 6 MHz frequencies for the system clock which can be selected by software through the PML.

- **PLL:** 8 MHz clock source which is internally divided by two (creating a 4 MHz clock) when PLL is selected as the system clock source. This clock source is generated by an external PLL. The PLL is not active at all times, it may be enabled and disabled as needed.
- **XTAL oscillator:** Secondary slower 32 kHz clock used in SLEEP and POWER-DOWN mode of the system as a system clock as well as a clock source for Watchdog timer, Wakeup timer, Long timer and GPIO.

These clock sources are then distributed through the system and controlled by the PML.

PLL 8 MHz clock source is present in the system to be able to supply 8 MHz to the USI Control Logic.

The reason for the slower XTAL oscillator clock source is to allow larger time-out intervals for timers without using large clock dividers or counters. A slower clock is also used during sleep mode to lower switching activity. Both use cases for the slower clock are to save power.

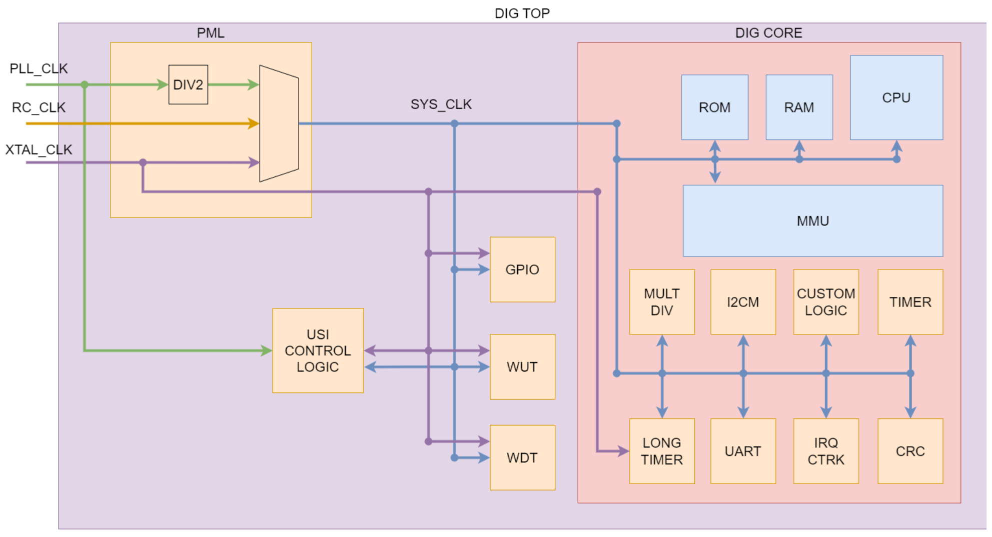


Figure 2.3: CoolRISC SoC clock structure

2.1.6 Peripherals

The current CoolRISC implementation features various peripherals which will be also re-used in the new RISC-V implementation. These peripherals are accessible by the CPU on the system bus in a specified address range. Those peripherals are PML, GPIO, Wakeup-timer, Watchdog, UART, I2CM, Timer, Custom Logic, CRC, Interrupt controller, Long-timer, MultDiv and USI Control Logic.

2.1.7 Software

Software running on the CPU is the ”brain” of the system. It is present in the Flash which is located outside of the system. The Flash is accessed through the NVM controller implemented in the MMU. Boot code and smaller procedures are hard-wired into the ROM.

2.2 CoolRISC CR816L overview

The CR816L core is a member of the CoolRISC [1] family of 8-bit microcontroller CPUs developed by the Swiss Center for Electronics and Microtechnology (CSEM) [2]. In this section, I describe the key features of the CoolRISC platform based on its documentation.

2.2.1 Harvard RISC-like architecture

Instructions are stored in the instruction memory whereas general-purpose data and peripherals are stored in separate data memory. The advantage of this architecture is that it gives the CPU the capability to read instruction operands in the data memory simultaneously with one instruction fetch.

2.2.2 Register-memory architecture

Instructions can operate with operands stored either in registers or in the data memory.

2.2.3 Memory sizes

Maximum data memory size is 64 Kbytes. The maximum program memory size is 64 Kinstructions, where one instruction is 22-bit wide.

2.2.4 Three-stage pipeline

One instruction enters the pipeline every clock cycle and it takes at most three clock cycles to execute. The pipeline does not feature branch prediction therefore it remains relatively simple.

2.2.5 8bx8b multiplier

The CR816 includes an 8-bit multiplier unit which executes one 8-bit multiplication in one clock cycle.

2.2.6 Stand-by mode

The HALT instruction can switch the core to halt mode in which the internal clock is stopped. The processor can be woken up using either events, interrupts or a reset. This minimizes power consumption.

2.2.7 Data and peripheral bus

The CPU features its own custom 8-bit data bus for data memory and peripherals. The bus does not feature any additional decoder. Each peripheral is connected to the data bus directly and decodes the address on the data bus in its own register map.

2.2.8 Instruction set

The ISA offers both RISC and CISC instructions to achieve a very dense program code. Each instruction is 22-bit wide.

2.2.9 Double-latch design

The CPU uses two non-overlapping clock sources with shifted phases and the CPU is made entirely made out of D-Latch registers. This reduces the area, power consumption and STA timing requirements of the CPU.

2.3 RISC-V overview

RISC-V [3] is an open-source, royalty-free ISA designed for computer processors. It was first introduced in 2010 at the University of California, Berkeley, and has since gained significant attention in the technology industry. It is a simplified and streamlined ISA that emphasizes simplicity and modularity, making it ideal for use in embedded systems and low-power devices.

2.3.1 Simplicity and modularity

One of the key features of RISC-V is its simplicity. It has a reduced number of instructions compared to other ISAs, making it easier to understand and implement. Additionally, RISC-V is modular, allowing designers to choose the specific instructions and features they need for a particular application. This modularity also makes it easier to customize and optimize the ISA for specific use cases.

2.3.2 Extendability

RISC-V is also designed to be extensible, meaning that new instructions and features can be added without breaking compatibility with existing software. This makes it easier to incorporate new technologies and innovations into the ISA without requiring significant changes to existing software and hardware.

2.3.3 Open source

Another advantage of RISC-V is that it is open-source and royalty-free. This means that anyone can access and use the ISA without paying licensing fees, making it a cost-effective choice for many applications.

2.3.4 RISC-V Foundation

To make the ISA stable and well defined the RISC-V Foundation was created in 2015 to own, maintain and publish current RISC-V ISA specifications.

"More than 3,100 RISC-V members across 70 countries contribute and collaborate to define RISC-V open specifications as well as convene and govern related technical, industry, domain, and special interest groups." [3]

2.3.5 ISA architecture

The RISC-V ISA is a load-store little-endian architecture supporting 32-bit or 64-bit address space. It features 32 general-purpose registers which are 32-bit wide. It does not require data to be properly aligned for the load and store instructions. In other words, any value may be stored at any address.

However, if data is not word-aligned then the load-store unit takes extra cycles to complete an instruction.

2.3.6 CV32E40P core

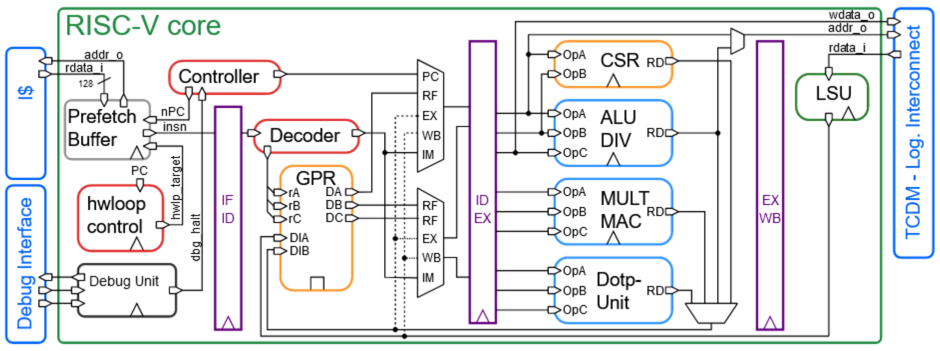


Figure 2.4: CV32E40P block diagram [4]

The CV32E40P [4] is a high-performance, low-power, 4-stage in-order 32-bit RISC-V processor core designed for embedded and IoT applications. Like CoolRISC, it uses Harvard architecture and has separate memory for instructions and data. The ISA of CV32E40P has been extended to support multiple additional instructions including hardware loops, post-increment load and store instructions and additional ALU instructions that are not part of the standard RISC-V ISA. The core also includes a debug interface and support for trace and profiling, making it easy to debug and optimize system performance.

The core is maintained by the OpenHW group [5] and it is fully verified by them which is one of the reasons why this core was selected for the implementation.

The RISC-V instruction sets supported by this core are RV32IMAC [6].

- **RV32I:** Base integer instruction which includes instructions for arithmetic, logical, and data transfer operations.
- **RV32M:** Extension to RV32I that adds instructions for integer multiplication and division.
- **RV32A:** Extension to RV32I that adds instructions for atomic memory operations.
- **RV32C:** Extension to RV32I that provides compressed 16-bit instructions to reduce code size and improve performance.

The RV32A is not needed and will not be used for the new SoC implementation to save the area and power consumption of the CPU.

The core uses Open Bus Interface (OBI) [7] protocol for instruction and data memory access. OBI is a point-to-point bus interface request-grant-based protocol developed by the OpenHW group. It is compatible with other industry bus protocols such as AMBA AHB and AMBA AXI developed by ARM.

2.4 CR816L and CV32E40P difference analysis

There are many differences between the CR816L core and the CV32E40P core. The first major difference is the data/instruction/address width. The CR816L features 8-bit data operations and 22-bit instructions compared to CV32E40P which has larger width of 32 for both. This gives the CV32E40P an advantage when it comes to larger address space and a performance advantage when it comes to operations with larger data types.

2.4.1 Code and data storage in the same memory

In the CoolRISC-based system, Flash memory is used for storing both code and data which means storing together 8-bit data and 22-bit instructions. This either causes poor memory utilization or requires additional logic to manage data alignment in the memory.

In comparison, the CV32E40P is 32-bit for both data and instructions. This means there are neither of these issues. Furthermore, the CV32E40P supports 16-bit compressed instructions which can be tightly packed in the memory along with 32-bit instructions because the CPU features hardware support for this.

2.4.2 General Purpose Registers

The CR816L features 16 8-bit data registers which are used during code execution. In comparison, the CV32E40P (and the RISC-V ISA in general) features 32 32-bit data registers. This gives the CV32E40P an advantage in being able to store more temporary data in itself instead of storing and fetching data from memory.

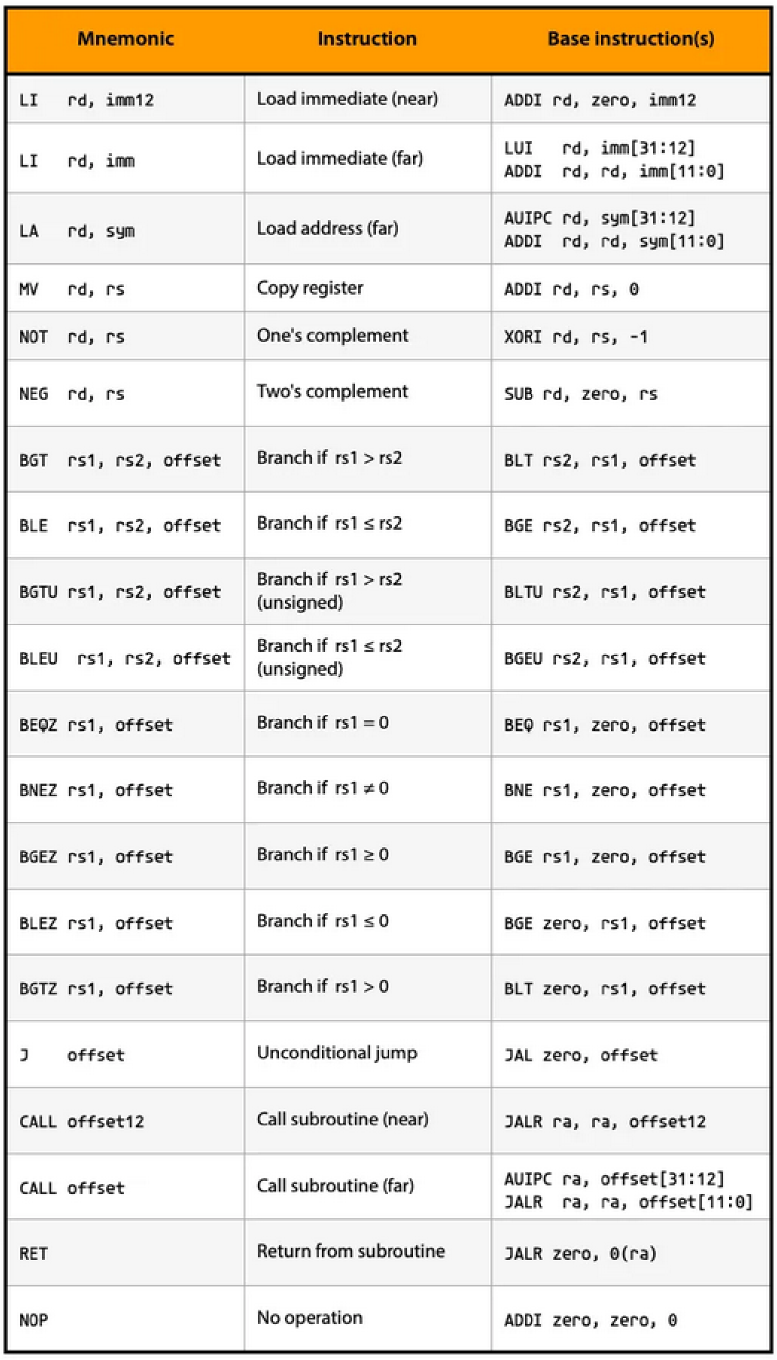
2.4.3 Instruction set

Comparing both instruction sets it comes clear that there are several differences between them. Arithmetic and logical operations are similar and present in both, however, the CV32E40P lacks any kind of ”carry” flags in arithmetic operations.

CR816L also features a hardware stack which CV32E40P does not implement at all.

More differences are present in the branching instructions and instructions for specific comparison and bit manipulation instructions. CV32E40P does not have a direct counterpart to some of them however those instructions can be emulated by pseudo instructions which are performed by different CV32E40P instructions with specific values in operands. Such pseudo instructions can be seen in Figure 2.5

Additional instructions, which the CV32E40P offers compared to CR816L, are mainly integer division instructions and custom instructions built into the core on top of the standard RISC-V instruction sets. However, the custom instructions will not be used in the new system to save area and power as their functionality is not needed.



Mnemonic	Instruction	Base instruction(s)
`LI rd, imm12`	Load immediate (near)	`ADDI rd, zero, imm12`

`LI rd, imm`	Load immediate (far)	`LUI rd, imm[31:12]`
		`ADDI rd, rd, imm[11:0]`

`LA rd, sym`	Load address (far)	`AUIPC rd, sym[31:12]`
		`ADDI rd, rd, sym[11:0]`

`MV rd, rs`	Copy register	`ADDI rd, rs, 0`
`NOT rd, rs`	One's complement	`XORI rd, rs, -1`
`NEG rd, rs`	Two's complement	`SUB rd, zero, rs`
`BGT rs1, rs2, offset`	Branch if rs1 > rs2	`BLT rs2, rs1, offset`
`BLE rs1, rs2, offset`	Branch if rs1 ≤ rs2	`BGE rs2, rs1, offset`
`BGTU rs1, rs2, offset`	Branch if rs1 > rs2 (unsigned)	`BLTU rs2, rs1, offset`
`BLEU rs1, rs2, offset`	Branch if rs1 ≤ rs2 (unsigned)	`BGEU rs2, rs1, offset`
`BEQZ rs1, offset`	Branch if rs1 = 0	`BEQ rs1, zero, offset`
`BNEZ rs1, offset`	Branch if rs1 ≠ 0	`BNE rs1, zero, offset`
`BGEZ rs1, offset`	Branch if rs1 ≥ 0	`BGE rs1, zero, offset`
`BLEZ rs1, offset`	Branch if rs1 ≤ 0	`BGE zero, rs1, offset`
`BGTZ rs1, offset`	Branch if rs1 > 0	`BLT zero, rs1, offset`
`J offset`	Unconditional jump	`JAL zero, offset`
`CALL offset12`	Call subroutine (near)	`JALR ra, ra, offset12`

`CALL offset`	Call subroutine (far)	`AUIPC ra, offset[31:12]`
		`JALR ra, ra, offset[11:0]`

`RET`	Return from subroutine	`JALR zero, 0(ra)`
`NOP`	No operation	`ADDI zero, zero, 0`

Figure 2.5: RISC-V Pseudo Instructions [8]

2.4.4 Multiplication/Division accelerator

The CR816L supports multiplication with 8-bit operands and 16-bit result. This multiplication takes one clock cycle to execute. The core does not feature division operation.

In comparison, the CV32E40P supports multiplication with 32-bit operands and gives lower 32-bit result while also supporting multiplication operation for the upper 32-bit result. Together this can provide 64-bit result. The lower half of the result takes one clock cycle to execute while the upper half takes five clock cycles to execute.

The core does support division operation which accepts 32-bit operands and is able to calculate the result in 3-35 clock cycles.

2.4.5 Interrupts

CR816L supports 24 interrupts divided into 3 levels of priority. In comparison, the CV32E40P supports up to 1008 interrupts each with up to 255 levels of priority.

2.4.6 Double latch vs. edge-based design

CV32E40P is a fully synchronous rising edge design. Meaning it is nearly fully made out of D-Flip-Flop registers. In comparison, CR816L is the direct opposite. The core is a latch-based design (or double-latch design) meaning that the core is made out of just D-latch registers.

Such a system works on a principle of two separate non-overlapping clock sources and D-latch registers being clocked by one or the other clock source.

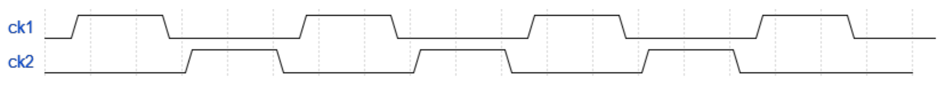


Figure 2.6: Non-overlapping clock example waveform

Data between registers is transferred in two phases. First CK1 clocked latches open for new data when CK1 is at ‘1’, then they close when CK1 is at ‘0’, and then CK2 clocked latches open for new data from CK1 clocked registers which are closed and stable at that moment when CK2 is at ‘1’.

In the pipeline, the first phase clock covers instruction prefetch and operand prefetch for ALU. The second phase clock covers the instruction decode and ALU execution.

In comparison in the edge design new data is loaded into registers on the rising (or falling) edge of the clock, and there is only one clock source for all registers (assuming the design is fully synchronous). Registers in such design are generally edge-triggered D-Flip-Flop type.

This gives the CR816L advantage in area and power consumption compared to RISC-V because latches are approximately half the size of normal D-Flip-Flop.

A D-Latch register generally consists of one inverter gate (2 transistors), two AND gates (2x6 transistors) and two NOR gates (2x4 transistors). In total, this accounts for 22 transistors. The structure of a D-Latch register is shown in Figure 2.7.

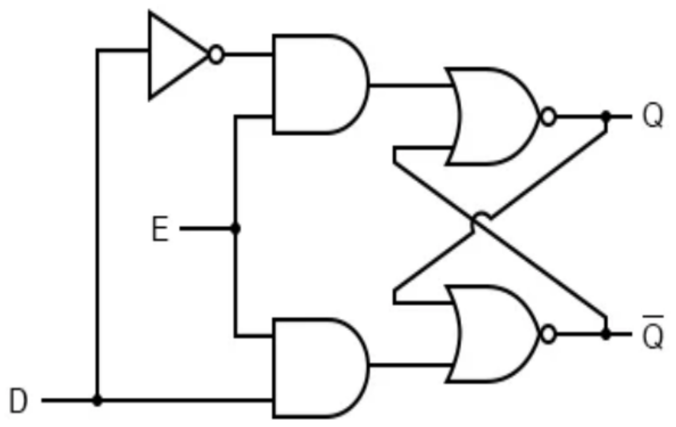


Figure 2.7: General D-Latch structure in CMOS [9]

However, in practice, there is a more efficient implementation used for the D-Latch register which is built out of fewer transistors. Such D-Latch is built using Tri-State buffers in structure in Figure 2.8. This is the D-Latch structure which was used in the 180 nm technology for PENRISCV.

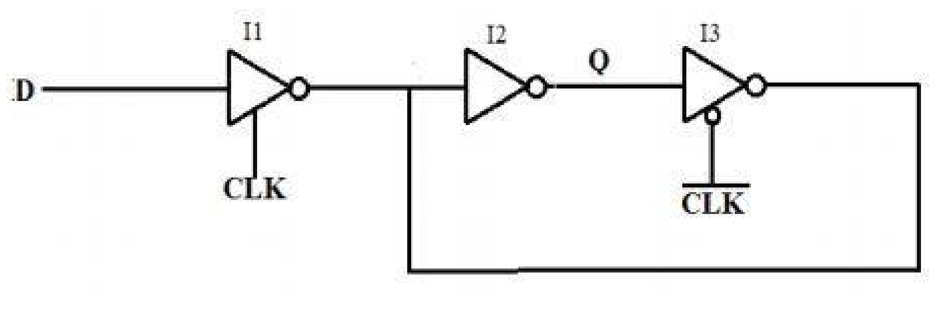


Figure 2.8: More efficient D-Latch structure in CMOS [10]

This structure consists only of two inverter gates (2x2 transistors) and two tri-state buffers with inverted outputs (2x4 transistors). In total this accounts for 12 transistors.

In comparison, an edge-triggered D-Flip-Flop is made out of two D-Latch structures with an additional inverter gate (meaning extra 2 transistors to the total count of 46 transistors). An example of an edge-triggered D-Flip-Flop can be seen in Figure 2.9.

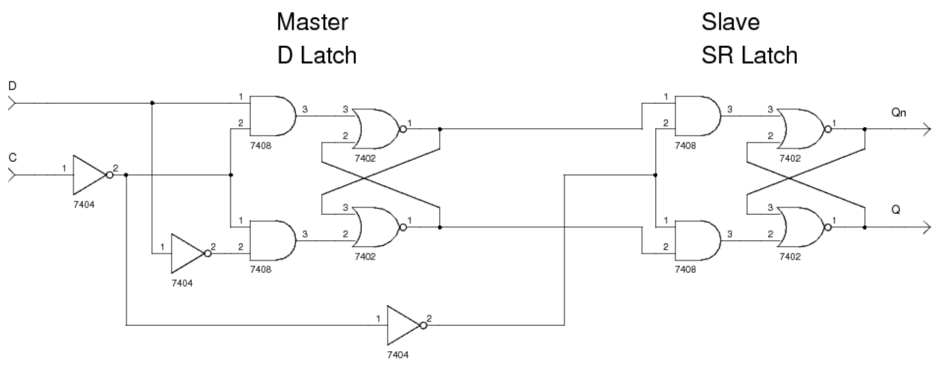


Figure 2.9: General D-Flip-Flop structure in CMOS [11]

The 180 nm process technology, which was used for the CoolRISC-based system and is going to be used for PENRISCV, uses a patented structure of an edge-triggered D-Flip-Flop.

This structure, in Figure 2.10, consists of 6 NAND gates (6x4 transistors) and one inverter gate (2 transistors). This accounts for 26 transistors in total. This is the DFF structure which was used in the 180 nm technology for PENRISCV.

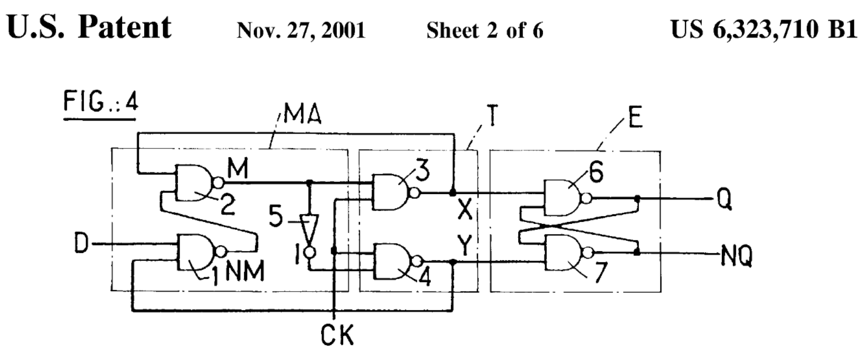


Figure 2.10: More efficient D-Flip-Flop structure in CMOS [12]

If we compare both register types then the D-Latch structure consists of less than half of the number of transistors of the D-Flip-Flop structure.

The other advantage (design-wise) of the double-latch design approach is that thanks to D-Latch registers, the CPU has better timing characteristics. This means that (in theory) the clock tree does not need to be so strictly balanced because it is expected that paths from CK1 to CK2 and backwards are not open at the same time. This reduces the number of required cells for the clock tree.

Better timing characteristics are thanks to **latch time borrowing** because latches are level sensitive. This can be defined as: "Time borrowing is the property of a latch by virtue of which a path ending at a latch can borrow time from the next path in the pipeline such that the overall time of the two paths remains the same. The time borrowed by the latch from the next stage in the pipeline is, then, subtracted from the next path's time." [13]

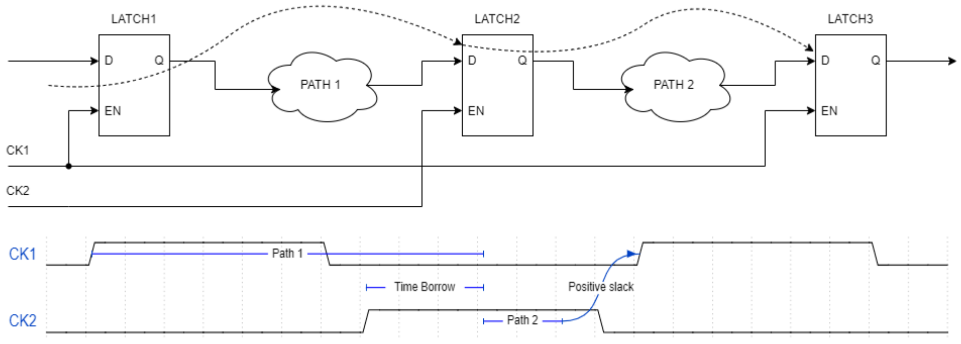


Figure 2.11: Latch time borrowing example

2.4.7 System clock speed vs MIPS

One disadvantage of the double-latch design is that it makes the system run only at half of the MIPS relative to the system clock frequency. In standard rising-edge design, a CPU executes an instruction in the pipeline every clock cycle. In comparison, the CR816L requires positive pulses on both CK1 and CK3 to execute an instruction in the pipeline. This effectively means that the number of MIPS the CPU can offer equals roughly half the number of MHz of the system clock.

2.4.8 Software support

As CR816L was developed in 2001, meaning there is very little continuous software/compiler support for it today. There is only one compiler which is not actively developed. The only modern tool for the CoolRISC CPU platform is Ride7 IDE [14] developed by Raisonance.

In comparison, RISC-V ISA is supported by a variety of modern compilers (such as GNU GCC, CLANG and more), modern IDEs (GNU MCU Eclipse, PlatformIO and more) as well as simulators and other useful tools. These and other new tools are in active development by the RISC-V community. [15]

2.4.9 Area and power consumption

In terms of area, CR816L has an advantage compared to CV32E40P. From internal measurements, CR816L has a size of around 4 000 standard gate cells whereas CV32E40P ends up at around 40 000. This difference will result in higher theoretical total power consumption on the CV32E40P in exchange for twice the MIPS/MHz, more instructions in the ISA and 32-bit data/instruction width.

To know if CV32E40P is a feasible alternative to CR816L in this system in terms of power and area, the system needs to be re-implemented with the new CPU and the power consumption estimated in post-layout gate level simulation.

If the CV32E40P turns out to not be a feasible alternative, the SoC implementation can still be used for testing other more efficient RISC-V cores. This is the advantage of the RISC-V ecosystem because it is the same ISA with many open-source (or even licensable) cores out there. With the same or similar interface, the CV32E40P is easily swappable for another RISC-V CPU. One that is currently being explored is the IBEX [16] RISC-V core which has a size of only around 15 000 standard gate cells.

2.5 Requirements for the new RISC-V pen controller SoC

Based on the analysis of the existing CoolRISC-based system and the differences between CR816L and CV32E40P cores I compiled the following requirements for the new RISC-V-based implementation of the SoC:

1. Perform transmission and reception of data on the pentips. Run software like CoreMark on the CPU.
2. Utilize the CV32E40P CPU core and minimize its area.
3. Use the ROM instead of Flash for software fetching. Memory Interface does not feature an NVM controller and this Flash memory will not be used for future projects anymore.
4. Feature the same power modes as the CoolRISC SoC.
5. Feature the same power domain structure as the CoolRISC SoC.

6. Feature the clock structure without the RC oscillator clock source. The main clock source shall be the 8 MHz PLL with a 4 MHz system clock. This is to simplify the development of the system to do the power consumption simulations because only one clock source is active during the majority of the system's operation.

7. Use the AHB-lite as a data bus for peripherals.

8. Reuse the following peripherals: PML, GPIO, Wakeup-timer, Watchdog, UART, I2CM, Timer, Custom Logic, CRC, Interrupt controller, Long-timer, and USI Control Logic. If any peripheral is too difficult to reuse for any reason, it shall be replaced with a peripheral with similar functionality.

9. Use the existing high-voltage pentip driver chip which was used for the CoolRISC SoC. For FPGA validation, use a manufactured sample.

10. DCDC control, Switch control, LED control and battery charging control do not need to be implemented in the system for power consumption results because analogue parts will be the same in both systems and will not affect the power consumption.

Chapter 3

System design of RISC-V pen controller SoC

In this chapter, I create a system design with the necessary changes for the new RISC-V-based pen controller SoC utilising the CV32E40P core. I name the new SoC ”PENRISCV”.

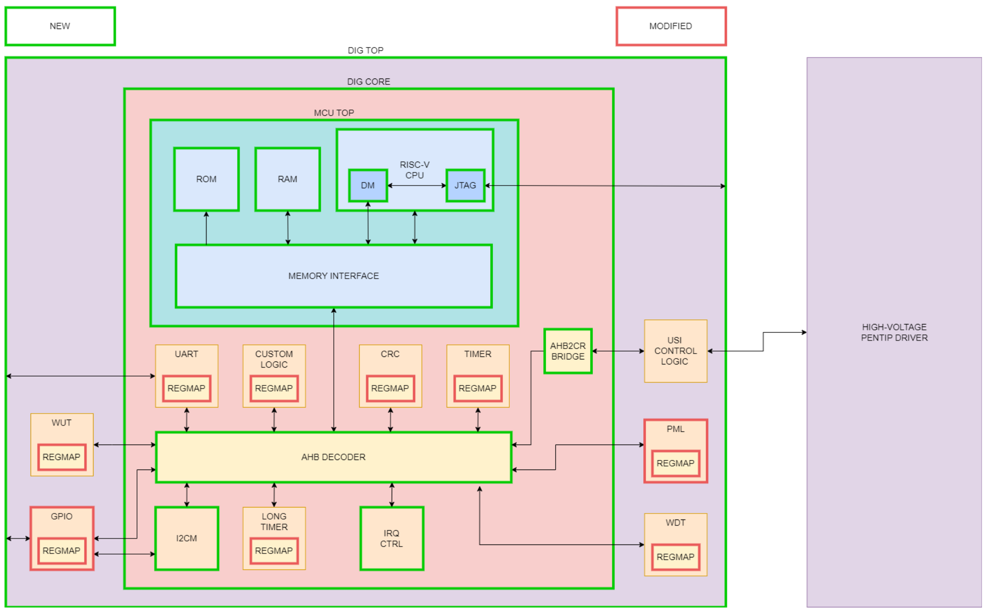


Figure 3.1: Changes in PENRISCV SoC compared to previous SoC. Green outline = new module, Red outline = modified existing module

3.1 System design

- The new SoC will implement enough functionality to be able to perform transmission and reception of data on the pentips as well as run software like CoreMark on the CPU.

- The new SoC will share the same power modes and power domains.

- The clock structure will be reduced to PLL and XTAL oscillator clock. The PLL clock will replace the RC oscillator clock as the main clock source for the system. The RC oscillator is not necessary for power estimations of the digital part of the SoC and omitting it is going to make the implementation of the new SoC faster.

- The external reset structure will be reduced to just a power-on reset source as the battery charger functionality will not be implemented in the new SoC for power estimations.

- The CV32E40P core will be configured in its smallest configuration possible. Such configuration disables all custom CV32E40P instruction extensions which are not part of the standard RISC-V ISA, it sets the number of debug performance counters inside the CPU to zero and uses the latch-based version of the CV32E40P register file. This is to save as much area and power as possible as the core is expected to be much bigger than the CR816L core. The core will be placed into a new design block along with the memory interface, RAM, ROM and debug module to better structure the system hierarchy.

- Software will be fetched from the ROM instead of the Flash. The ROM will have a size of 32768 bytes. RAM will have a size of 8192 bytes.

- The memory interface is a new design block which will replace the Memory Management Unit from the previous SoC. Because CR816L and CV32E40P use different data/instruction interface the original MMU cannot be reused.

- The system bus for peripherals in the new SoC will be AMBA AHB-lite [17]. AHB-lite was chosen because it is one of the compatible bus protocols the CV32E40P supports and internally in the company AHB-lite already has various generating scripts to speed up integration.

- Power Management Logic will be reused and simplified/adjusted to fit in the new SoC design. Elements such as non-overlapping clocks for CoolRISC will be removed as the new SoC will be mainly a rising edge-based design. DCDC control, Switch control, LED control and battery charging control logic inside the PML will not be implemented.

- A new bus interface for peripherals means that each peripheral needs to have its register map replaced by one that is AHB-lite compliant. However, the USI Control Logic cannot be modified, it does not feature a standard register map which would be easily replaceable. It is also an externally made IP. An interface converter from AHB-lite to the CoolRISC data interface will be needed for this block.

- Current I2C Master will be replaced by a new I2C Master which is AHB-lite compliant. The previous I2C Master is not compatible with AHB-lite and new system clock structure therefore it was decided to replace it with a new I2C Master IP.

- AHB-lite requires a dedicated AHB Decoder which will manage access to peripherals based on a given address. The decoder will be created to fit the defined address space and peripherals used in the SoC.

- The previous interrupt controller will be replaced by a new one because CV32E40P and CR816L have different interfaces for interrupts. All interrupts in the system will be considered asynchronous. Interrupt sources are UART, I2CM, GPIO, WDT, WUT, and USI Control logic.

- The GASP interface and custom CoolRISC debug module was used for debugging in the previous SoC. The GASP interface will be replaced by JTAG and the debug module will be replaced by RISC-V specific debug module.

- UART interface will be connected directly to the top and not through GPIO. This is to make the debugging easier.

3.1.1 Top interface

Top interface signals are described in tables 3.1, 3.2, 3.3 and 3.4.

Table 3.1: PENRISCV top interface - generics

Generic name	Description
G_ACTIVE_EDGE_LVL	Active edge level (rising/falling) of the system
G_ACTIVE_EDGE_LVL_N	Negative edge level (rising/falling) of the system
G_IO_ADDR_WIDTH	Address width of AHB
G_IO_DATA_WIDTH	Data width of AHB
G_REG_ADDR_OFFSET	Register offset of AHB peripherals
G_TARGET_TECH	Target technology. '0' for ASIC, '1' for FPGA

Table 3.2: PENRISCV top interface - Main signals

Signal name	Description
clk_i	Main 8 MHz clock from PLL
clk_32khz_i	32 kHz clock from XTAL oscillator
rst_ni	Power-on negative asynchronous reset
test_en_i	Test mode enable
idcode_i	IDCODE for JTAG TAP
tck_i	JTAG test clock pad
tms_i	JTAG test mode select pad
trst_ni	JTAG test reset pad
td_i	JTAG test data input pad
td_o	JTAG test data output pad
tdo_oe_o	Data out output enable
DoC_and_core_acc_dis_i	disable all access except for SBA
uart_rxd	RXD pad of UART
uart_txd	TXD pad of UART
core_busy_o	Indicator signal that CPU is not in sleep mode
dmi_req_active_o	DMI Clock Request
vdd1_sleep_ack_n	Acknowledge signal for VDD1 switching
pml_xtalosc_en	XTAL oscillator enable

Table 3.3: PENRISCV top interface - USI Control Logic signals

Signal name	Description
pp_data	Pen pressure data
pp_rdy	Pen pressure ready
usi_pp_cpmd	Pen pressure compare
usi_pp_oe	Pen pressure output enable
usi_pp_ck_32k	Pen pressure 32 kHz clock
usi_pll_en	PLL enable
usi_pp_run	Pen pressure run
usi_pp_double	Pen pressure double run
sw_data	Switch data
pll_rdy	PLL ready
pll_err	PLL error
hv_spi_cs	SPI chip select for HV pentip driver
hv_spi_ck	SPI clock for HV pentip driver
hv_spi_do	SPI data to from dig_top to HV pentip driver
hv_spi_do_oe	SPI data output enable for HV pentip driver
hv_spi_di	SPI data input from HV pentip driver to dig_top
hv_es_rx0	Data from pentip 0
hv_es_rx1	Data from pentip 1
hv_p0en	Pentip 0 enable
hv_p1en	Pentip 1 enable
hv_ulen	Uplink enable
hv_es_tx0	Data to pentip 0
hv_es_tx1	Data to pentip 1
hv_reset_n	HV pentip driver reset

Table 3.4: PENRISCV top interface - GPIO signals

Signal name	Description
pa_in	GPIO pads input
pa_in_en	GPIO pads input enable
pa_out	GPIO pads output
pa_pu	GPIO pads pull-up enable
pa_oe	GPIO pads output enable
pa_i2c_pu_en	GPIO I2C pull-up enable
pa_i2c_od	GPIO I2C open-drain enable
pa_i2c_q	GPIO I2C output signals

3.1.2 Memorry address mapping

Addresses for each memory can be found in Table 3.5.

Table 3.5: Memory address mapping

Memory	Base address (hexadecimal)
ROM	0x00008000
RAM	0x00010000
Debug Module	0x00015000
Peripherals (AHB)	0x00020000

3.1.3 Peripheral address mapping

Addresses for each peripheral can be found in Table 3.6.

Table 3.6: Peripheral address mapping

Peripheral	Base AHB address (hexadecimal)
IRQ Ctrl	0x0000
UART	0x0400
I2CM	0x0800
Timer	0x0C00
Custom Logic	0x1000
CRC	0x1400
WDT	0x1800
WUT	0x1C00
Long Timer	0x2000
GPIO	0x2800
USI Control Logic	0x2C00
PML	0x3000

3.2 FPGA vs ASIC implementation

In this section, I describe the differences and limitations between FPGA and ASIC and how they affect the design implementation.

3.2.1 FPGA's clocking resources

FPGAs are limited in some ways for implementing digital designs. One of the limiting factors is the limited number of clock tree resources. In an ASIC, a clock tree is created and balanced in the place-and-route step of the design. However, FPGA only has a set number of pre-balanced clock trees and corresponding buffers for its registers. This limits the number of separate clocks in the design and other constructs such as clock gating or clock multiplexing. [18]

3.2.2 FPGA LookUp Tables

Another limiting factor of FPGAs is the fact that when a design is placed onto the FPGA, it is not made out of standard gates as it would be on an ASIC. The design will be implemented onto the FPGA using its Lookup Tables (LUTs), Flip-Flop registers, Block RAM modules, etc.

"The LUT is the basic building block of an FPGA and is capable of implementing any logic function of N Boolean variables." [19]

For normal synchronous designs without any special constructs, this is not an issue. However, when a design features specific structures made out of basic gates to ensure required behaviour, issues may arise in the design.

For example, a design may feature some specific logic constructs which are made out of specific basic gates to ensure glitch-free output (for example clock gates). Another example may be a debouncer structure.

In an ASIC you can simply hand-build these structures using basic gates and you can be (almost) sure that the structure will be present in the design after synthesis exactly as you designed it.

In an FPGA there are no basic gates. There are only LUTs configured to give desired output based on the boolean function that is defined by the RTL code. Since this is a LUT and not a basic gate, it cannot be guaranteed that the output will have the same glitch-free characteristics as the basic gate on ASIC.

Furthermore, the FPGA synthesis tool may optimize the design and the boolean functions and the final implementation may have different characteristics while giving the same output.

3.2.3 FPGA register asynchronous set and reset

Another limitation of an FPGA is that a design cannot use both asynchronous sets and reset at the same time. Only one may be implemented at the time.

“The flip-flops in Xilinx FPGAs can support both asynchronous and synchronous reset and set controls. However, the underlying flip-flop can natively implement only one set / reset / preset / clear at a time.” [20]

This is not an issue when implementing a fully synchronous design on an FPGA, however, it may be an issue when specialized constructs are being implemented, which require this both set and reset. This is for example a register which has its value loaded into it asynchronously. Then this implementation needs to utilise both asynchronous set and reset.

Chapter 4

Implementation

In this chapter, I describe the RTL implementation of the new pen controller SoC with RISC-V CPU. I also describe the implementation of the test bench for its verification.

Before beginning the RTL implementation, I studied how the CoolRISC-based system was implemented in detail. This helped me when creating the new implementation. The old implementation served as a reference when creating the new implementation.

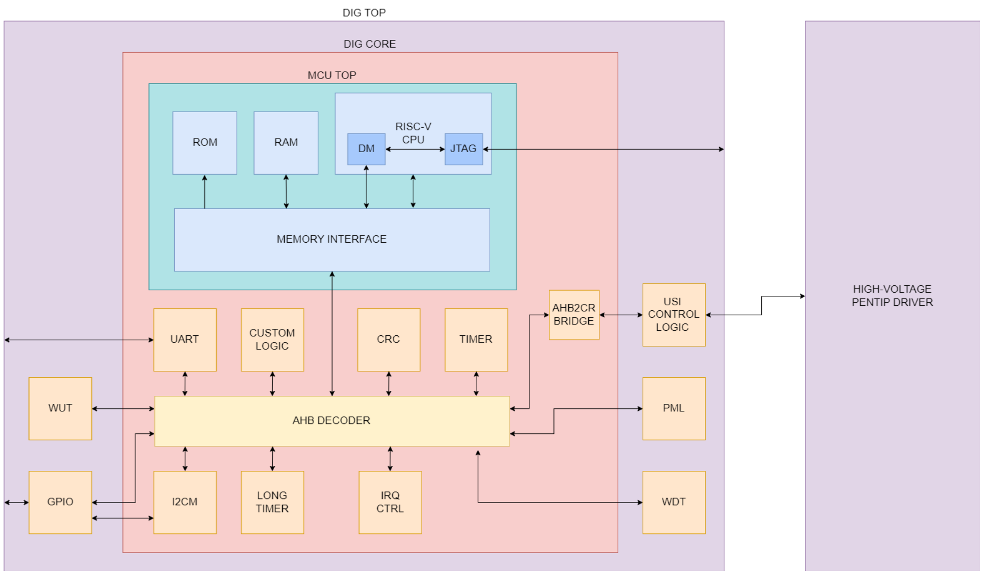


Figure 4.1: New RISC-V pen controller SoC - PENRISCV

4.1 mcu_top

I decided to use a combination of SystemVerilog and VHDL for the implementation. This was not a problem as modern simulators support mixed language simulation.

To start the implementation, I first created the mcu_top module from the SoC block diagram4.1 which contained the CPU, Debug Module, JTAG, Memory Interface, RAM and ROM. I reused a module which contained the CPU and Debug Module. This module was provided to me by the company.

Next, I modified an XML file for the script that generated the Memory Interface. This script was provided to me by the company. In the XML file, I defined the interface properties for the module, as well as the address ranges for RAM, ROM, Debug Module, and peripheral access. Based on this definition, the script generated a Memory Interface module in SystemVerilog that included interfaces for the CPU and the interfaces defined in the XML file.

RAM and ROM were implemented in the form of a SystemVerilog RTL model. These models were also generated by the company's script using an XML definition. These modules were replaced by actual technology cell models later in the Physical implementation chapter.

4.2 dig_core

After creating the mcu_top module I moved to the dig_core module which contained the mcu_top, AHB-lite decoder, UART, I2C Master, Custom Logic, CRC, Timer, Long Timer and IRQ Ctrl.

For the AHB-lite decoder and the IRQ Ctrl, I used another script provided to me by the company to generate them. AHB-lite decoder was defined by an XML similarly to Memory Interface. Each peripheral had a specified base address and range in the XML. Based on this definition the decoder was generated as a SystemVerilog module.

The IRQ Ctrl also had an XML definition with required interrupt sources and the type of each interrupt. All interrupts in this design were considered asynchronous.

4.2.1 Peripherals

The remaining peripherals are going to be reused (I2CM is going to be reused from a different project). However, first, peripherals needed to have their register maps and data interface adjusted to be AHB-lite compliant.

To do this I used the company's scripts for register map generation. The script can receive a PERL register map definition and generate a register map block in VHDL based on given register templates.

For each peripheral, I created its own PERL register map definition. When creating each definition I tried to keep the same register structure as in the CoolRISC SoC implementation. Because of the move from an 8-bit to a 32-bit system, I decided to merge some registers into one where it made sense and would not affect functionality.

After generating new register maps for all peripherals, I integrated them into each peripheral.

I did this process for peripherals located outside of the dig_core block as well. The only exception is the USI Control Logic which cannot be modified with a new register map.

Some reused peripherals feature CK3 clock input even after the register map adjustments. That is because those peripherals may be using the CK3 clock internally. I connected the standard system clock to this clock source after verifying that it will not affect the functionality of the peripheral.

4.2.2 AHB2CR bridge

To connect the USI Control Logic, which still uses the CoolRISC data bus interface, to the AHB-lite bus, I needed to create an interface converter. After studying both AHB-lite and CoolRISC data bus protocol I came up with a solution.

Table 4.1: CoolRISC data bus interface

Signal name	Description
ck1	Phase 1 system clock
ck3	Phase 2 system clock
dm_e	Active transaction on the bus
read_nwrite	'1' for the read transaction, '0' for the write transaction
dm_addr	Address from the CPU to peripheral
wdata	Data from the CPU to peripheral
rdata	Data from peripheral to CPU

The CoolRISC data bus protocol executes a transaction in two steps. The first step is the start of a transaction by asserting the *dm_e*, *read_nwrite*, *data_out* and *dm_addr* signals. Because this is a double latch-based design the *dm_e*, *read_nwrite*, *data_out* and *dm_addr* signals are latched when CK1 is high.

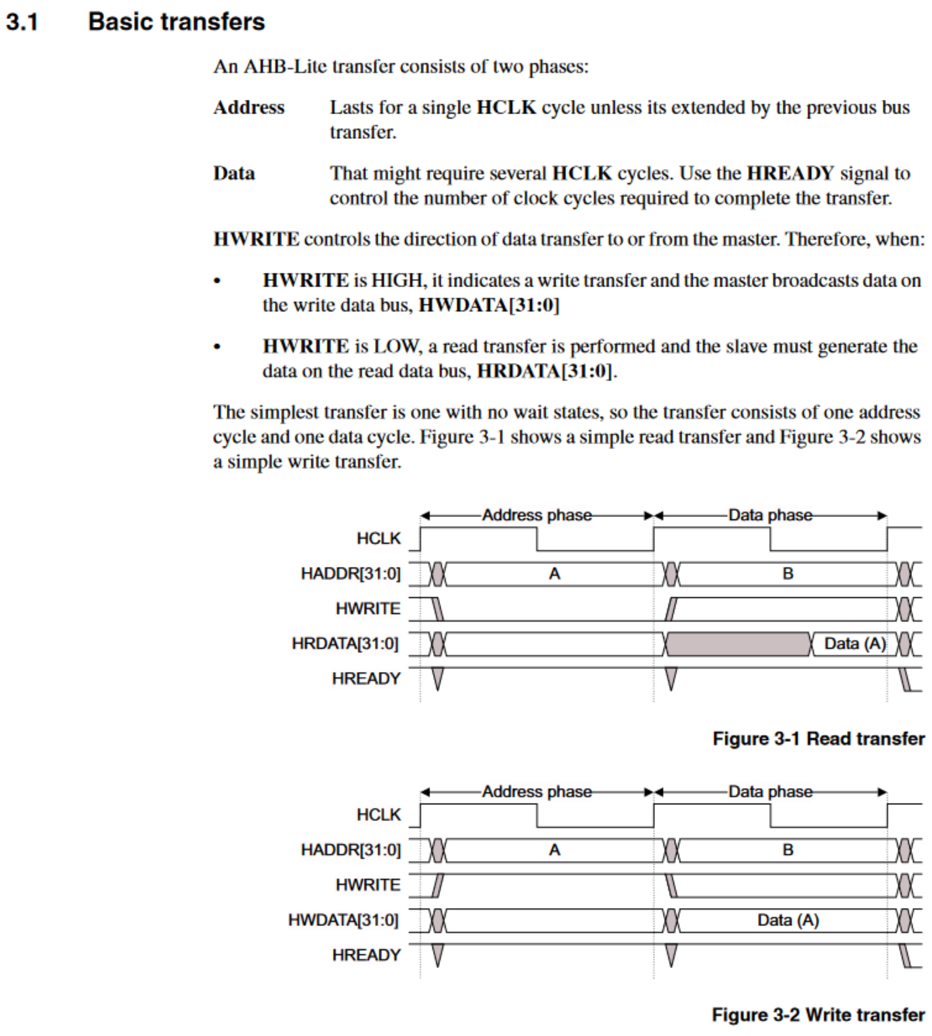
The second step is data access. A write access can start on the falling edge of the CK1 signal and for read access, the read data is latched when CK3 is high.

In PENRISCV there was only a single clock source for the system clock. This means that the first step described above would happen on the first rising edge of the clock. The write access would happen on the second rising edge of the clock. Read access data would be available on the third rising edge of the clock.

Table 4.2: AHB-lite bus interface

Signal name	Description
hclk	System (bus) clock
hsel	Decoded peripheral bus select
hwrite	Bus transfer direction indication. '0' for read transaction, '1' for write transaction
haddr	Address from the AHB master to peripheral
htrans	Transfer type
hsize	Transfer size per clock
hready_in	Bus ready (from bus multiplexer, indicates CPU bus is in a wait state)
hwdata	Write data bus
hready_out	Ready output (used to insert bus wait states based on peripheral response rate)
hrdata	Read data bus
hresp	Bus transfer response (0 = OK, 1 = ERROR)

AHB-lite works similarly but allows for higher data throughput because it can overlap transactions. What this means is when the data phase of the transaction is being executed the address of the next transaction is already on the bus for the slave device to sample. Like this, each transaction only takes two clock cycles.



3.1 Basic transfers

An AHB-Lite transfer consists of two phases:

Address	Lasts for a single **HCLK** cycle unless its extended by the previous bus transfer.
Data	That might require several **HCLK** cycles. Use the **HREADY** signal to control the number of clock cycles required to complete the transfer.

HWRITE controls the direction of data transfer to or from the master. Therefore, when:

- **HWRITE** is HIGH, it indicates a write transfer and the master broadcasts data on the write data bus, **HWDATA[31:0]**
- **HWRITE** is LOW, a read transfer is performed and the slave must generate the data on the read data bus, **HRDATA[31:0]**.

The simplest transfer is one with no wait states, so the transfer consists of one address cycle and one data cycle. Figure 3-1 shows a simple read transfer and Figure 3-2 shows a simple write transfer.

Figure 3-1 Read transfer

Figure 3-2 Write transfer

Figure 4.2: AHB-lite basic transfers [17]

AHB-lite also features a *hready* signal which is used by slave devices to indicate to the master whether they are ready to complete the data access. The *hready* signal is to be asserted after sampling the address from the master.

I used this feature when implementing the AHB2CR bridge by creating a wait cycle during the transaction to comply with the CoolRISC data interface protocol. The AHB-lite decoder implementation which is generated by the company's script has its interface outputs registered which creates one clock delay in their assertion towards the slave. This means that the AHB2CR bridge has to assert *hready* low immediately when the peripheral is selected by *hsel* high signal.

In summary, the final transaction flow of the AHB2CR bridge is as follows:

1. Phase 0 (Idle): While no transaction is ongoing, no signals are asserted for the CoolRISC slave.

2. Phase 1 (Address): When a transaction is being started on the AHB side (*hsel* going high), the AHB2CR bridge immediately responds to the AHB master by setting *hready* low to request a wait state. At the same time, the bridge sets *dm_e* signal to the CoolRISC slave high to indicate an incoming transaction to the slave. The *read_nwrite* signal in the CoolRISC slave interface is an inverted *hwrite* signal from AHB.

3. Phase 2 (Data): The slave writes the data into the register selected by address or gives out data to be read by the master. At this time the bridge sets *hready* high and *dm_e* low. Afterwards, the bridge returns to Phase 0, or Phase 1 if there is another transaction right after.

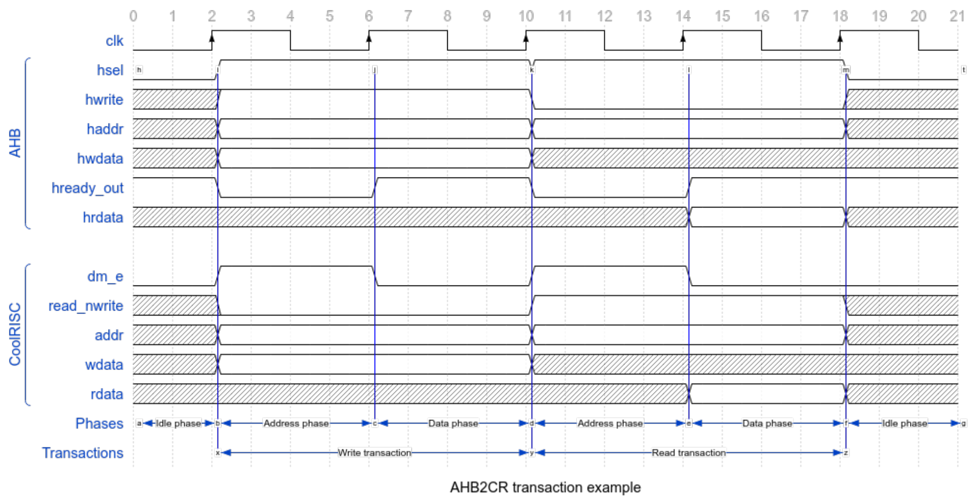


Figure 4.3: AHB2CR bridge transaction example

Another difference between AHB-lite and CoolRISC bus is that AHB-lite (in the case of PENRISCV) is 32-bit while CoolRISC bus is 8-bit. This has two effects on implementation:

1. AHB can read/write up to 4 bytes per transaction, however only LSB byte is considered for read/write to CoolRISC slave in this implementation. (Regardless if the AHB transaction is 1, 2, 3 or 4 bytes at a time)

2. AHB register addresses are incremented by 4 whereas CoolRISC register addresses are incremented by 1. This is simply solved by taking the AHB address which is given to the slave and divided by 4.

4.3 dig_top

The top module of the digital part of the SoC is the dig_top. In this module, I placed the dig_core along with the remaining peripherals. Those peripherals are the WDT, WUT, USI Control Logic, PML and GPIO.

4.3.1 GPIO

Aside from replacing the register map with a new one that is AHB-lite compliant, I needed to make further adjustments to the GPIO to fit the new requirements. In the CoolRISC-based system, the GPIO was responsible for selecting the I2CM interface on two of the GPIO pads and selecting the UART interface on two of the GASP pads. In PENRISCV however, the GASP is no longer present and it is replaced by JTAG which is routed directly outside to the top and not through the GPIO. This means the switching functionality inside GPIO for GASP and UART will be removed and GPIO will only switch between standard GPIO operation and the I2CM interface.

4.3.2 PML

PENRISCV does not feature DCDC, LED, Switch and battery charger logic which means I needed to adjust PML to function correctly. Inside PML I carefully removed functionality for the previously mentioned components.

RC oscillator clock is not used in PENRISCV, therefore I needed to remove RC oscillator-related functionality from the clock controller inside PML. I also removed the clock functionality for the (now removed) DCDC logic.

Chapter **5**

Verification

To verify that the PENRISCV implementation functions correctly I had to build a test bench environment. To do this I re-used some components from the CoolRISC SoC test bench environment which was provided to me by the company.

As a simulation environment, I used the "run_sim" script which was provided to me by the company. This script uses the Cadence Xcelium [21] simulator which is used for the compilation and simulation of the design as well as the test bench environment. The script also gathers all necessary design files, configuration switches and paths necessary for compilation. It is run from a command line and it can be configured to select between specified designs (RTL, gate simulation, post-layout simulation, etc.), select a specific test, open the GUI of Xcelium, and more.

5.1 Test bench structure

The test bench environment used Universal Verification Methodology (UVM). It is a standardized methodology for digital design verification. The main components of the test bench are UVM_TEST and TB_TOP.

UVM_TEST is the main module which contains everything UVM-related and the test sequence. Inside the UVM_TEST there is also the UVM_ENV which contains UART agent and I2CM agent modules which serve as an automatic driver, monitor and checker for the corresponding peripherals.

TB_TOP is the module which contains the Design Under Test (DUT) which is in our case dig_top of PENRISCV. It also contains the high-voltage pentip driver model and IO pad models.

There are several virtual interfaces which are used for communication between the TB_TOP and the UVM environment.

The test bench environment features "C2T" functionality. It is a set of test bench procedures which are used to communicate with the software running on the CPU during a test. It uses specific places in RAM for writing and reading values which are used for data transfer and synchronization with the test sequence.

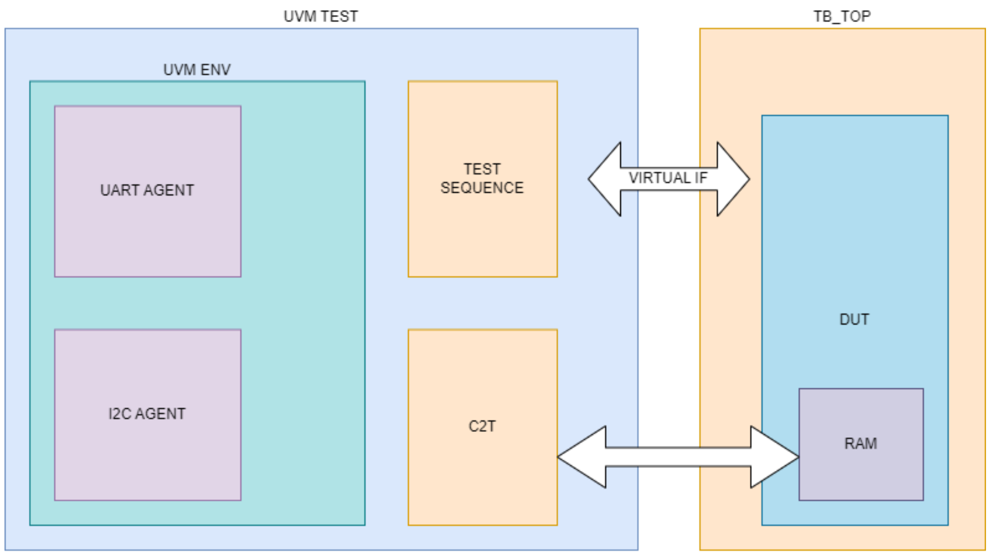


Figure 5.1: PENRISCV test bench structure

Table 5.1: Virtual interfaces

Interface	Description
tb_clk_if	Generating clocks for the DUT
cpu2tb_if	C2T functionality
hv_if	Specific signals inside HV pentip driver model
gpio_if	GPIO interface emulation
slave_i2c_if	Interface for I2C slave agent
uart_if	Interface for UART agent
mirror_if	Mirroring some signals inside DUT to the TB

5.2 Test bench implementation

I implemented the test bench using the UVM library and SystemVerilog. I created the tb_top module and placed the PENRISCV DUT (dig_top) in it along with the high-voltage pen tip driver model, IO pad models, instantiated interfaces and procedures.

UVM_TEST is implemented by penriscv_base_test.sv which extends the uvm_test class. UVM_ENV is implemented by the penriscv_env.sv which extends the uvm_test class.

I reused the penriscv_base_test.sv and penriscv_env.sv implementation from the previous test bench environment as well as the UART and I2CM agent modules.

I reused the virtual interfaces for communication with the TB_TOP and made minor adjustments to them to fit the PENRISCV test bench.

The C2T functionality was reused from the previous test bench. I needed to modify most of the procedures to correctly work with the RISC-V CPU. That included adjusting the RAM access because there is now a new RAM structure and procedures. After all, each memory address is now incremented by four instead of one as it was in the CoolRISC SoC.

Each test has its own test sequence which is implemented in the form of a class which extends the penriscv_base_test. These test sequences were reused from the previous test bench and I made minor modifications where needed to make them work with the new test bench. Along with this sequence, there is also software to be compiled for each test case. I used the previous software for each test case as a reference but I needed to rewrite/adjust it for the new RISC-V CPU.

5.3 Software compilation

Part of the test bench was also the software compilation for each test case. The compilation is implemented using a Makefile and the OpenHW RISC-V compiler for the CV32E40P CPU.

The compiler also needs a linker script file which defines the base address and size for individual sections (ROM, RAM, stack, bss, etc.) which I implemented into file cpu_sw.ld.

The source code which is compiled for each test consists of the startup assembler and C code for the CPU, interrupt vector table, C2T functionality implemented in C code, C header files with register access macros register maps of each peripheral, RISC-V header and the source code of the test itself.

The headers for peripheral register access macros are generated from the register map script described in the previous chapter along with the register maps themselves. I created a perifs.h header file which includes all of the other header files for all peripherals.

The C2T functionality described in the previous section has its software implementation in the form of C functions a c2t.h and c2t.c. I also needed to adjust these functions to work with the RISC-V CPU.

The final compiled software for each test is in the form of an ELF file which is then converted into a MIF file by elf2mif script. The ELF file is a binary file representing the compiled binary, whereas the MIF file is a text hexadecimal representation of the ROM contents (instructions and data). The CPU executes the software from the ROM.

The MIF file is then loaded into the ROM model during simulation compilation. SystemVerilog offers the readmemh function for that.

5.3.1 CoreMark

CoreMark [22] benchmark was used for power and performance measurement between PENRISCV SoC and the CoolRISC-based SoC. It is a widely used benchmark designed for measuring the performance of microcontrollers and processor cores used in embedded systems. It supports 8-bit to 64-bit CPU architectures therefore it can be used to compare RISC-V and CoolRISC (8-bit vs 32-bit).

This benchmark comes in the form of several C source files and headers. I created a test case which contains CoreMark software as the main application. To compile it I created a separate Makefile which includes the CoreMark source files. This way I was able to simulate the CoreMark run and check if everything works in simulation and then simply use the same software compilation flow for later.

I modified the in the core_portme.c and core_portme.h CoreMark files to fit the PENRISCV design. That mainly included start/stop time functions which control the timer and define correct data types which CoreMark uses for its calculations.

CoreMark comes with an ee_printf.c print function implementation for embedded systems. I modified it so it prints the string to the PC terminal using the UART peripheral. This function is used by CoreMark to print information and results during the benchmark run.

5.4 List of tests

The goal of the thesis was to simulate the power consumption of the system and not to manufacture it yet. Therefore it was not necessary to test every part of the system and every functionality and make it fully verified. It was decided to have at least one test for each major component of the system (aside from CPU debug features).

5.4.1 penriscv_crc_01

Compares the CRC result of every input data to every polynomial configuration to a CRC result calculated by the test bench.

5.4.2 penriscv_custom_01

The test bench sends data into the CPU as input data for Custom Logic and compares it to its results. It tries every combination of data.

5.4.3 penriscv_gpio_00

Checks the input and output functionality of each GPIO pad.

5.4.4 penriscv_gpio_01

Checks each edge and level detection functionality together with the debouncer functionality of each GPIO pad.

5.4.5 penriscv_i2c_01

Uses the I2C UVM agent to emulate an I2C Slave device. Test writes and reads data to the device and checks if data matches.

5.4.6 penriscv_long_timer_01

Tests various time durations and checks if Long Timer values correspond with them.

5.4.7 penriscv_pml_01

Switches between system power modes. First from ACTIVE to SLEEP mode and then from ACTIVE to POWER DOWN mode. Checks system clock frequency after each switch.

5.4.8 penriscv_timer_00

Sets value to Timer to count down from and after time-out checks if it occurred in the expected time.

5.4.9 penriscv_uart_00

Uses the UART UVM agent to send and receive various data. UART agent automatically checks data and protocol correctness in various settings.

5.4.10 penriscv_wdt_00

A basic test for watchdog time-out. Sets value to watchdog and checks if time-out occurred in the expected time.

5.4.11 penriscv_wdt_01

More advanced test for watchdog timer which checks the time-out occurrence, system reset and with and without auto-reload of watchdog value.

5.4.12 penriscv_usi_01

In this test, USI Control Logic is first configured for transmitting on pentip 0. After a short transmission of ones and zeros on the pentip 0, it is configured for receiving. A square wave sequence is forced on pentip 0 by the test bench for the USI Control Logic to detect. If detection is successful then the software will check this bit of the USI Control Logic status register.

5.4.13 penriscv_wut_00

Test configures the wake-up timer at various durations in ACTIVE mode and SLEEP mode and checks the duration in each configuration. It also checks if the wake-up from SLEEP mode functionality of the system works correctly.

5.4.14 penriscv_coremark_01

Test case containing the CoreMark software compilation and simulation of a single run. By default, CoreMark is configured with the data size set to 1600 to correspond with the CoolRISC-based system configuration and the number of iterations set to 2 to be able to finish the simulation in a reasonable time.

Chapter 6

FPGA validation

In this chapter, I describe the steps that I had taken in the implementation of PENRISCV on an FPGA. The main purpose of an FPGA environment for PENRISCV is to enable the customer to test their software in the new system in real-time.

6.1 FPGA platform

I used the Xilinx FPGA platform for validation. It is already being used in the company and I could set it up quickly.

The company provided me with an FPGA board with an already integrated Xilinx FPGA chip. It is a Xilinx Atrix-7 family model xc7a100tcsg324-2 FPGA.

I used Xilinx Vivado 2019.1 [23] for synthesis, placement and bitstream generation and Xilinx Vivado Lab 2022.2 [24] for bitstream upload to the FPGA.

6.2 FPGA design changes

To implement PENRISCV on the FPGA I needed to make several design changes which are specific to an FPGA. I implemented these changes on the RTL level using parametrization. On the top level of each RTL module, there is a *target_tech* parameter which selects if the design is meant for FPGA or ASISC. When FPGA is selected, the changes described in this section are applied.

6.2.1 GPIO Debouncer

The function of a debouncer is to filter out short glitches on the input asynchronous signal. The current implementation of the debouncer used in the

CoolRISC-based system requires the signal to be stable for at least one clock period of the 32 kHz XTAL clock.

As mentioned previously, the FPGA does not support having the asynchronous set and reset connected and used at the same time. Only one of them is available on each register. The current implementation of each debouncer in the GPIO uses a structure that utilises both asynchronous set and reset. This means that for it to function properly on the FPGA, it needs to be adjusted.

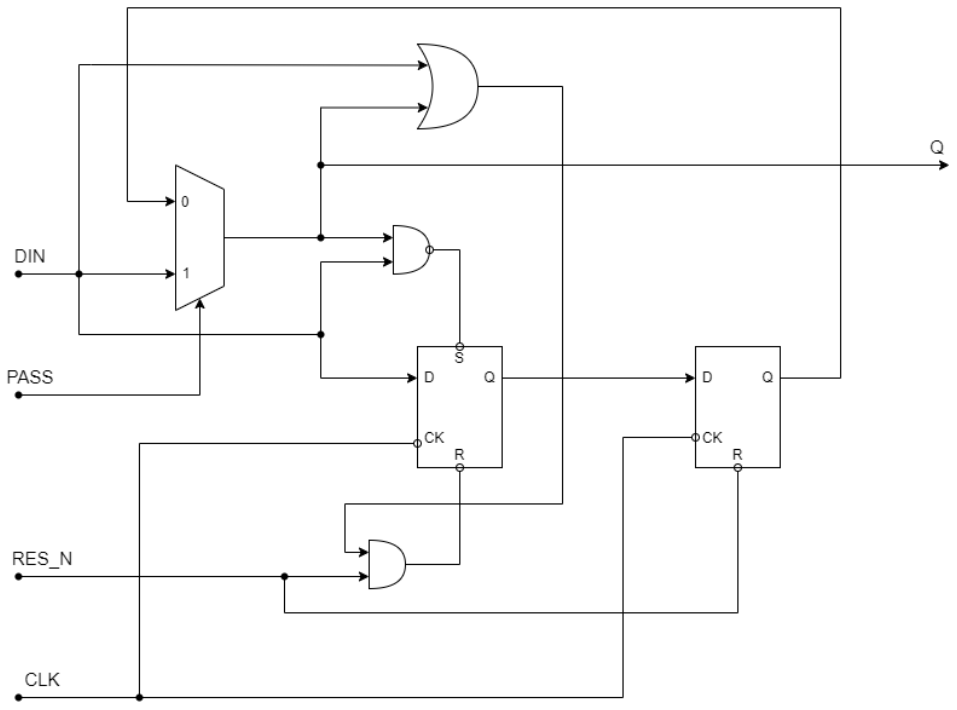


Figure 6.1: GPIO debouncer used in the CoolRISC-based system

I decided to implement the GPIO debouncer which uses more registers but no longer requires any register to utilise both asynchronous set and reset. The structure of this debouncer can be seen in Figure 6.2.

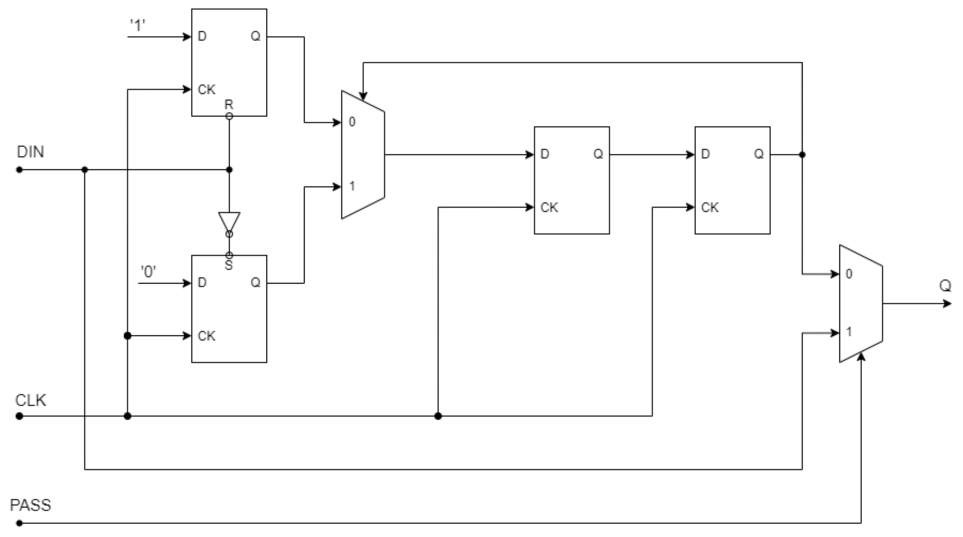


Figure 6.2: GPIO debouncer adjusted for FPGA

6.2.2 Removing clock multiplexors from WDT and WUT

As I mentioned in the previous chapter about FPGA limitations, there are fewer clocking resources on an FPGA than on an ASIC. For this reason, it was decided to remove the clock multiplexors inside WDT and WUT and hard-wire one source selection.

While this reduced the functionality of WDT and WUT, it did not affect the ability to test the software on the FPGA implementation.

6.2.3 Replacing buffers on clock sources in PML

Inside the clock controller in the PML, there are three buffers for WDT, WUT, and GPIO on the XTAL clock source. This is for the clock tree synthesis. These three buffers needed to be replaced by FPGA clock buffer cells for FPGA implementation.

6.2.4 Replacing system clock multiplexor in PML

System clock switching between PLL and XTAL clocks is handled by a clock multiplexor inside the clock controller inside the PML. This multiplexor could not be removed therefore it was replaced by an FPGA clock multiplexor.

6.2.5 Emulating POWER DOWN mode by dig_core reset

On FPGA there was not going to be an actual implementation for turning off the VDD1 domain in POWER DOWN mode. To emulate the functional behaviour of the design, the VDD1 enable signal was used to hold the dig_core module in reset and thus emulating the power down state.

In the dig_top module, I placed an AND RTL module with system reset and VDD1 switch from the PML as its inputs and the output is then used as a reset signal for the dig_core.

6.3 FPGA implementation

The FPGA implementation used the same RTL design files as the ASIC implementation, but it used FPGA directives and parametrization to implement the design changes described above.

6.3.1 FPGA top

The dig_top module would not work on FPGA on its own. It needs to have emulation for the GPIO pads, 8 MHz and 32 kHz clock source, reset generation, etc.

For this purpose, I created the penriscv_fpga_top module. This module contains the instance of PENRISCV dig_top.

The GPIO pads are emulated on the RTL level, the FPGA synthesis was able to apply the RTL logic to its internal IO pads.

Listing 6.1: Pads emulation in FPGA top

```
pa_gen: FOR i IN 0 TO 1 GENERATE

   i_pad_a_q(i) <= pa(i) WHEN (i_pad_a_ie(i) = '1') ELSE '0';

   PROCESS (i_pad_i2c_od, i_pad_a_d, i_pad_a_oe) IS
   BEGIN
      IF i_pad_i2c_od(i) = '1' THEN
         pa(i) <= 'Z' WHEN (i_pad_a_d(i) = '1') ELSE '0';
      ELSE
         pa(i) <= i_pad_a_d(i) WHEN (i_pad_a_oe(i) = '1') ELSE 'Z';
      END IF;
   END PROCESS;

END GENERATE;

i_pad_a_q(2) <= pa(2) WHEN (i_pad_a_ie(2) = '1') ELSE '0';
pa(2) <= i_pad_a_d(2) WHEN (i_pad_a_oe(2) = '1') ELSE 'Z';

-- PortA pull-up and pull-down emulation
pa_upn <= NOT (i_pad_a_pu OR ('0' & i_pad_i2c_pu));
pa_dn  <= i_pad_a_pd;
```

The FPGA features a 100 MHz clock source. However, for PENRISCV I needed 8MHz and 32 kHz clock. To achieve this I used Vivado's Clocking Wizard [25] tool to generate a parametrized RTL module which generates an 8 MHz clock. The RTL module utilises the FPGA's PLL module and configures it to get an 8 MHz clock in the design.

The PLL module has a limited clock frequency range and cannot generate a 32 kHz clock. To achieve the 32 kHz clock I implemented a simple synchronous clock divider clk_div_fpga and placed it into the penriscv_fpga_top. This divider generates 32 kHz out of the 8 MHz clock provided by the PLL and it contains one FPGA clock buffer so that synthesis assigns this clock source to the clocking resources.

It takes some time at the power up of the system for the PLL to be locked and generate specified clock frequencies. The PLL has a "locked" signal which indicates whether the PLL is ready. This signal is used as a reset source for the PENRISCV system. Essentially the whole system is held in reset until PLL is ready and provides correct clocks.

I also decided to route the GPIO output signals from dig_top to the penriscv_fpga_top interface and connected them to the three LEDs on the FPGA board for debugging purposes.

6.3.2 Constraints

Constraints are needed for the FPGA synthesis of the design. I created two XDC constraint files. penriscv_top_fpga_tim.xdc for timing constraints where I defined clocks, generated clocks and IO delays, and penriscv_top_fpga_loc.xdc where I defined the properties for each IO port.

6.3.3 PCB for high-voltage pentip driver

As written in the chapter USI pen controller requirements, the pen controller consists of two chips. Low-voltage SoC with CPU and second high-voltage chip. The FPGA implements only the low-voltage SoC and the HV pentip driver chip had to be connected externally to the FPGA's IO pads.

The FPGA board (as it was designed) only supports 3.3 V voltage levels on the IO pads. But the HV pentip driver chip required 1.2 V and 2 V voltage levels and additional components. This meant I needed to design a PCB which features voltage level shifters, voltage regulators, HV pentip driver chip socket and other components required by the HV pentip driver.

To create the schematic for the PCB I used the KiCad 6.0 [26] tool. I defined which pins on the FPGA board will be used for the interface to connect it with the PCB and based on that created a symbol in KiCad to use in the schematic. I called this symbol fpga_if.

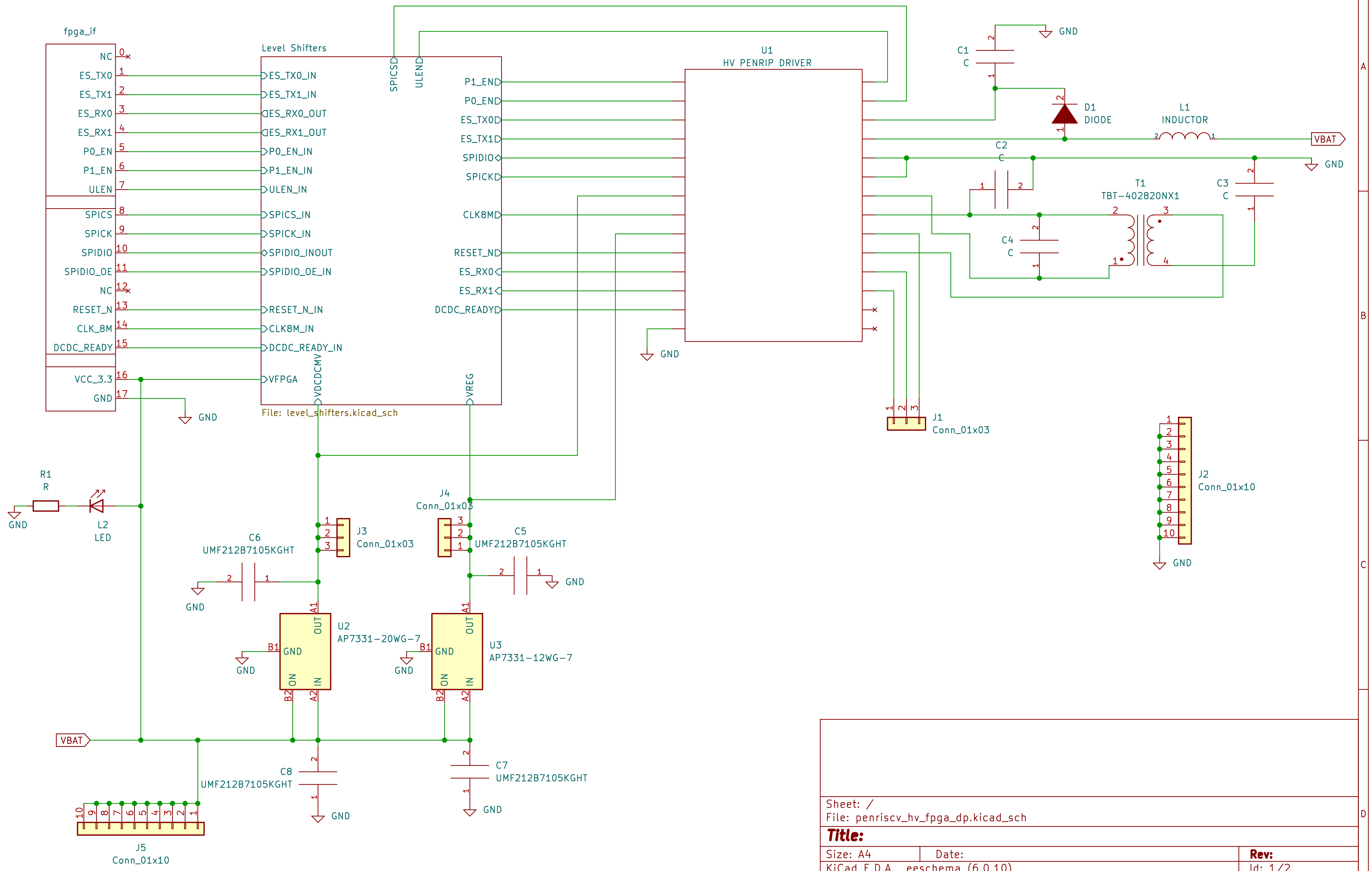
The main part of the PCB is the voltage regulators and level shifters. For voltage regulators, I used the AP7331-20WG-7 for 1.2 V and AP7331-20WG-7 for 2 V. For level shifters, I used the SN74LXC2T45.

The HV pentip driver needs additional components to work correctly. Such as the TBT-402820NX1 transformer. I created the circuitry schematic for the HV pentip driver based on its company specification.

When the schematic was completed, I sent it to a layout engineer to handle PCB layout design, manufacturing in Prague Board and installing individual components on the board.

The manufactured PCB can be seen in figure 6.3. The pins used for the interface between the FPGA board and the HV pentip driver PCB can be seen in figure 6.4. The final setup featuring the HV pentip driver in the DIL package socketed in the HV PCB which is connected to the FPGA board can be seen in figure 6.5.

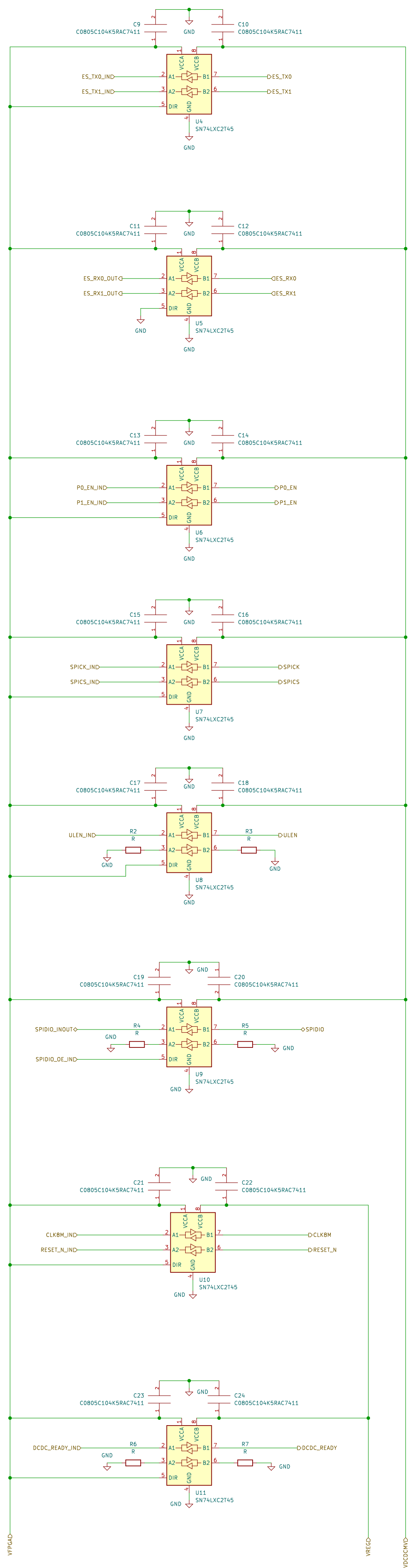
The next two pages feature the HV pentip driver PCB schematic from KiCad.



Sheet: /
File: penriscv_hv_fpga_dp.kicad_sch

Title:

Size: A4 | Date: | **Rev:**
KiCad E.D.A. eeschema (6.0.10) | Id: 1/2



Sheet: /Level Shifters/
File: level_shifters.kicad_sch

Title:

Size: A2 | Date: | Rev:

KiCad E.D.A. eeschema (6.0.10) | Id: 2/2

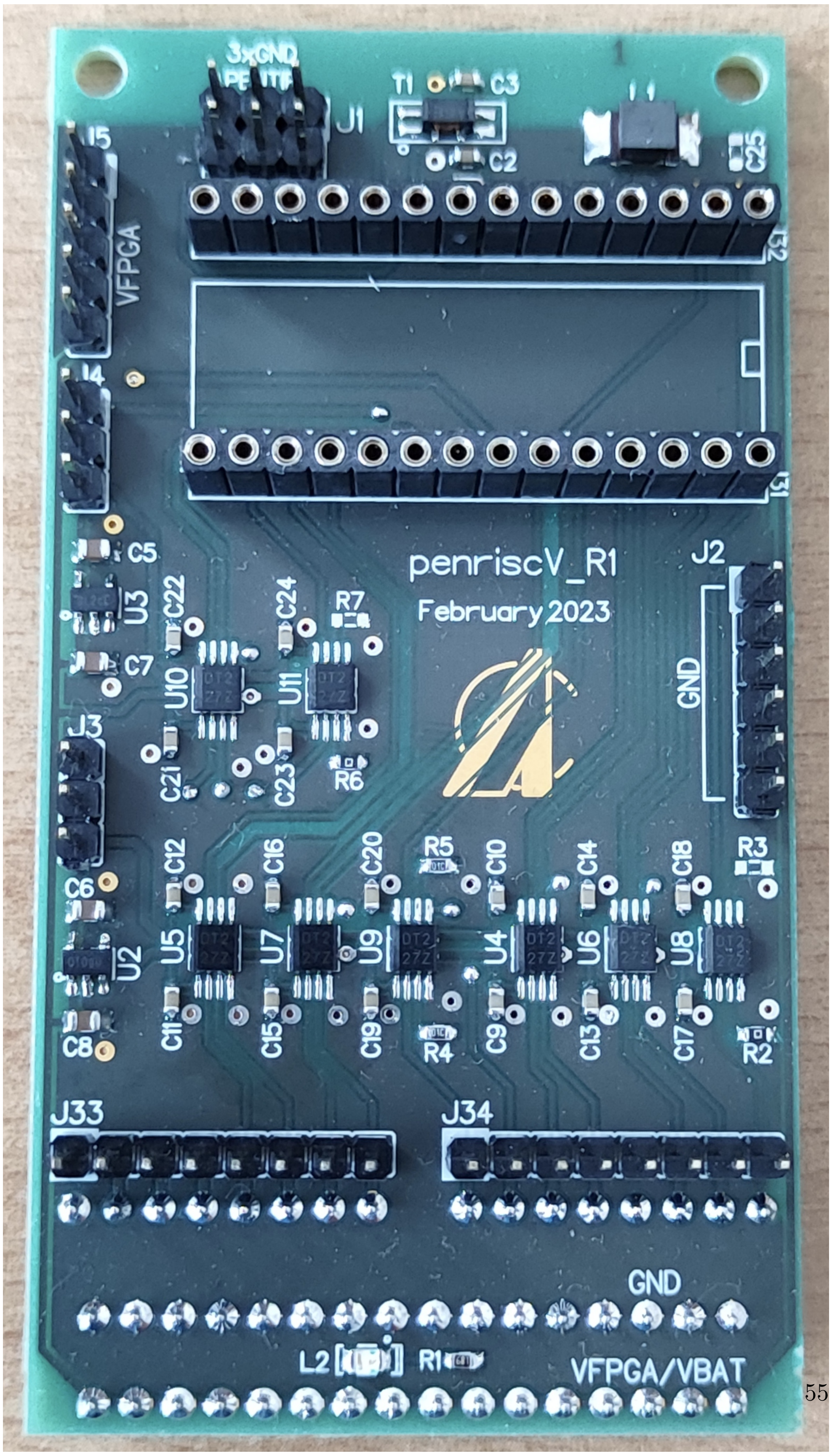


Figure 6.3: HV pentip driver PCB

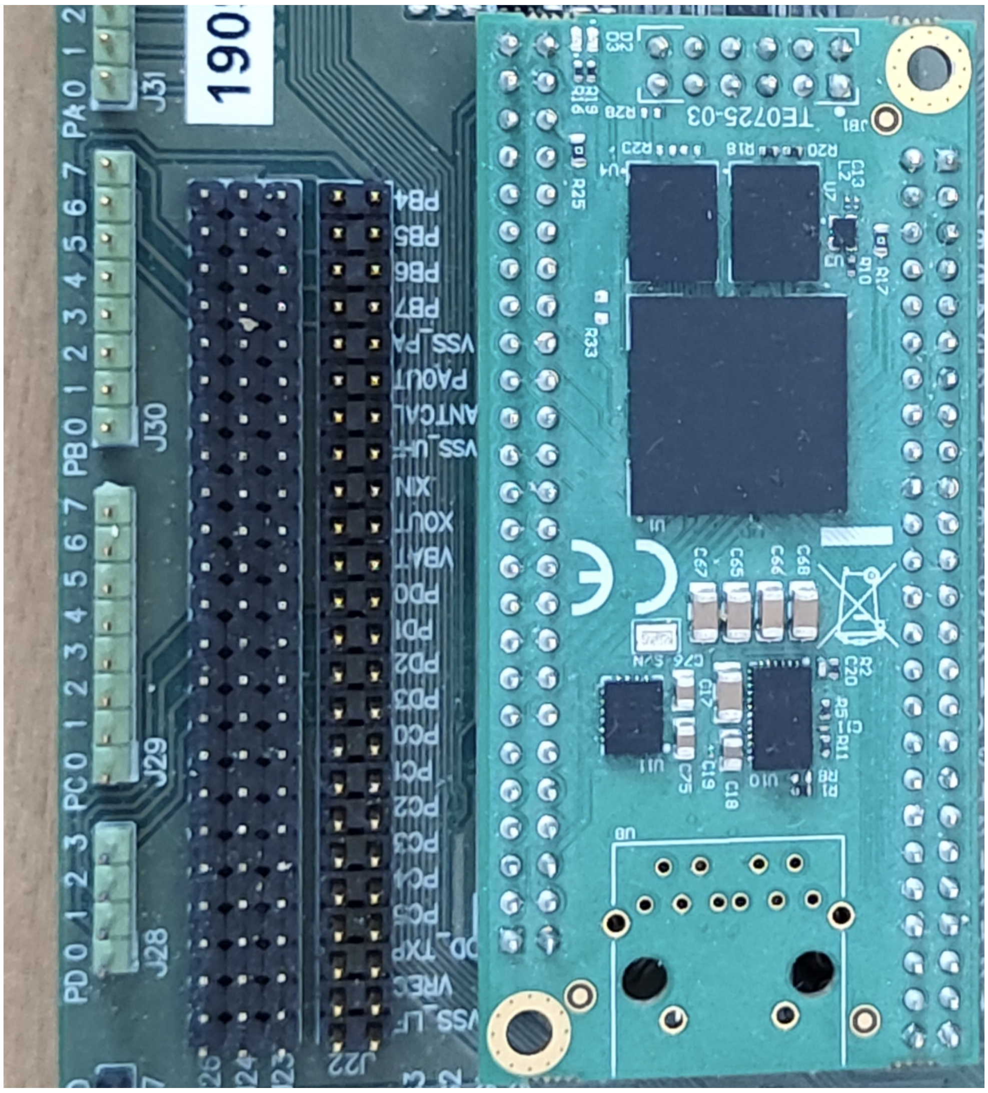


Figure 6.4: HV pentip driver PCB FPGA interface

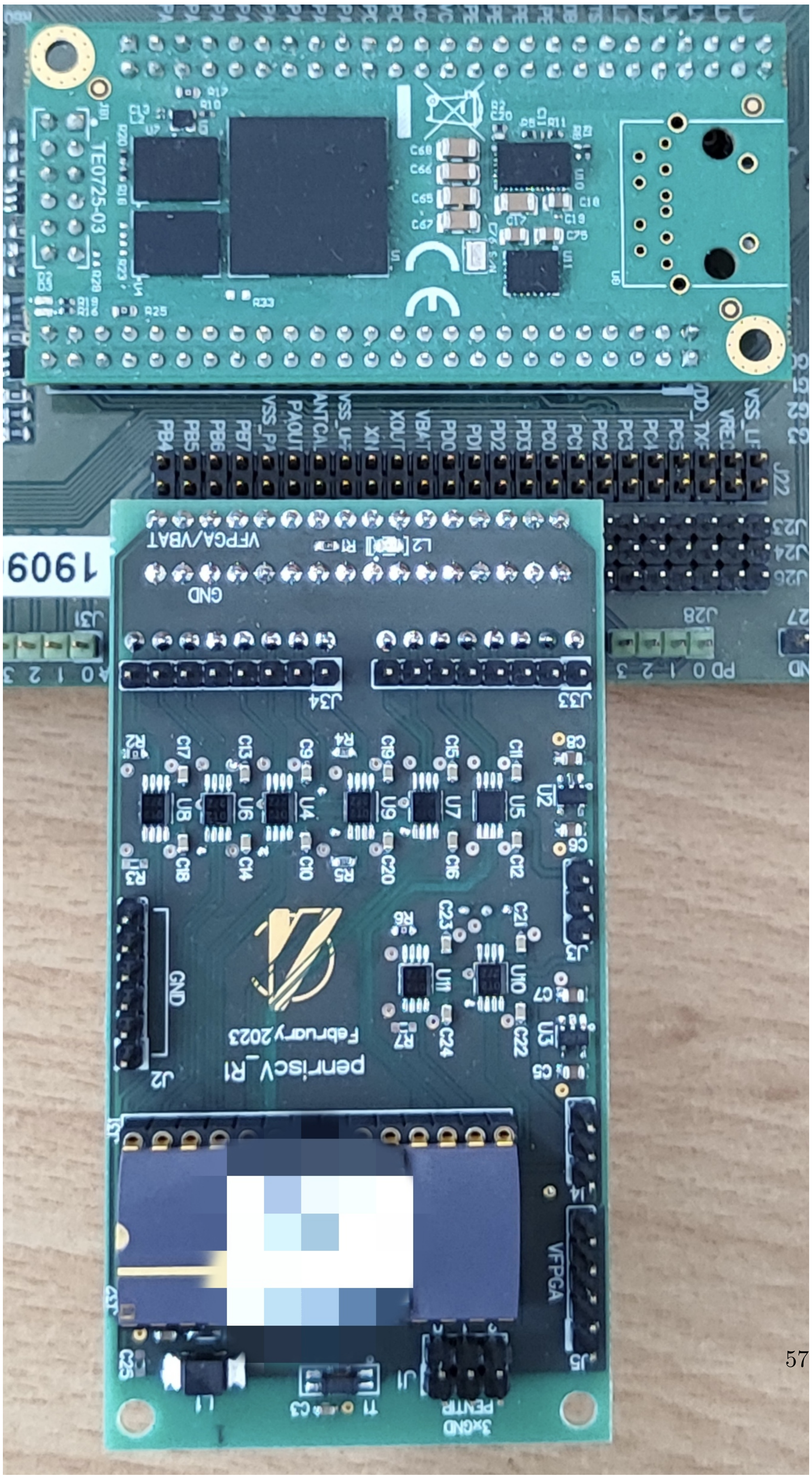


Figure 6.5: Connected HV pentip driver PCB to the FPGA board

6.4 Patching ROM contents in bitstream

Each synthesis run in Vivado takes around 15 to 20 minutes. This is a very long time to wait each time there is a change to the software inside the ROM. To avoid this I used the data2MEM tool from Xilinx which allows me to change the contents of the ROM inside the generated bitstream.

Because the ROM is a large continuous block of data the Vivado synthesis and placement tool maps it to its built-in Block RAM modules inside the FPGA. Thanks to this it is possible to identify these blocks of BRAM inside the synthesized design and use the data2MEM tool to change their contents.

This allows for instant software change inside the ROM and no need to run the entire synthesis-placement-bitstream flow unless there is a design change.

To automate this process I created the generate_rom_bmm.tcl script which is used by Vivado during the bitstream generation flow. This script is used to generate the Block Memory Map (BMM) file for the data2MEM tool.

The final bitstream patching flow starts with software compilation where the output is an ELF file. This ELF file is converted to a MEM file which is a text hexadecimal representation of the ROM contents. Then the data2MEM tool is used to patch the bitstream contents. It takes the bitstream, the MEM file and the BMM file as inputs and outputs the patched bitstream.

6.5 Validation scenarios

To enable the customer to test their software on the FPGA PENRISCV implementation, only selected functionality needed to be validated. That included the software running correctly on the CPU, UART/I2C peripherals operating correctly for debugging and receiving/transmitting data on the pentips using the USI Control Logic. It was decided to validate the following scenarios to demonstrate to the customer that the FPGA platform works and can be used for software testing.

6.5.1 LED control by GPIO

As I mentioned in previous sections, I connected GPIO output pads to three LEDs which are built into the FPGA board. I used these LEDs this way for debugging software on PENRISCV when running it on the FPGA. First I checked that LEDs are correctly controlled by software by writing different output values to the GPIO pads and watching LEDs change their state.

6.5.2 UART communication with the PC

Aside from LEDs, I wanted to use UART for more detailed debugging. To connect PENRISCV UART pins to the PC I used an FTDI TTL-232R-3V3

USB converter cable and the Putty tool to send and receive ASCII data from PENRISCV over UART.

I created a software test case where the software configures UART to the standard 9600 baud rate and without parity bit and then sends a short message over the UART to the PC. The software then waits for UART transmission from the PC to PENRISCV and sends it back to the PC.

This way I tested both send and receive functionality of UART. Afterwards, I used UART to output debug and information messages when working on other scenarios.

6.5.3 I2C Master communication with an I2C Slave device

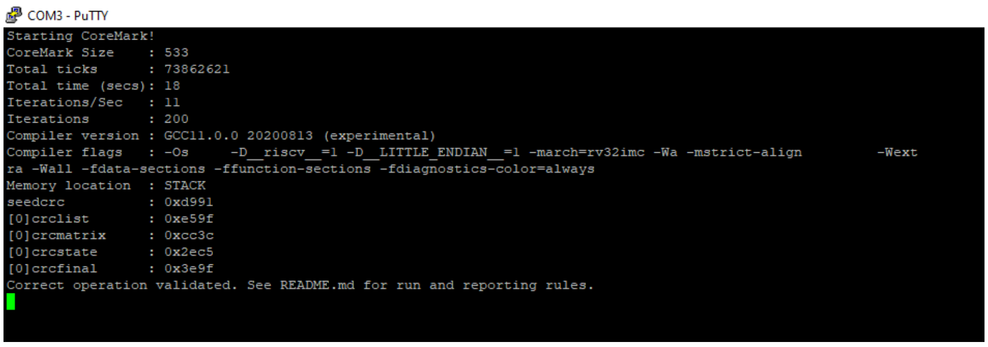
To test the I2C master peripheral I used the NCD2400MTR [27] I2C slave device. For this scenario, I created software for PENRISCV which configures the I2CM peripheral to the supported standard I2C speed. The device I2C address of NCD2400MTR is 0x60.

The software then writes to some registers inside the NCD2400MTR and then reads them and then compares both values to check if they are matching.

6.5.4 Coremark run longer than 10 seconds

CoreMark considers the final score valid only if the benchmark runs for more than 10 seconds. To make the run last longer than 10 seconds, I needed to adjust the data size and the number of iterations. The data size was set to 1600, the same value used in benchmarking the CoolRISC-based system, and the number of iterations was set to 200 which makes it last longer than 10 seconds.

A screenshot of the CoreMark result from PENRISCV run on the FPGA can be seen in figure 6.6.



```
Starting CoreMark!
CoreMark Size    : 533
Total ticks      : 73862621
Total time (secs): 18
Iterations/Sec   : 11
Iterations       : 200
Compiler version : GCC11.0.0 20200813 (experimental)
Compiler flags   : -Os     -D__riscv__=1 -D__LITTLE_ENDIAN__=1 -march=rv32imc -Wa -mstrict-align         -Wext
ra -Wall -fdata-sections -ffunction-sections -fdiagnostics-color=always
Memory location  : STACK
seedcrc          : 0xd991
[0]crclist       : 0xe59f
[0]crcmatrix     : 0xcc3c
[0]crcstate      : 0x2ec5
[0]crcfinal      : 0x3e9f
Correct operation validated. See README.md for run and reporting rules.
```

Figure 6.6: CoreMark result from PENRISCV run on the FPGA

6.5.5 Transmission on pentip 0

To test the transmission on the pentip 0 it was decided to use 100 kHz 20 V square wave mode that the USI Control Logic can configure the HV pen controller driver to generate. I created a software test case for simulation which configures the USI Control Logic and HV pen controller to do that. I used the validation script from a validation engineer as a reference when creating the C software for this test case.

Firstly I simulated this test case to see if the software and the system behave as expected. When everything worked in simulation I patched the bitstream with the compiled software and ran it in real time on FPGA. To check the correct behaviour on the pentip 0 I used an oscilloscope to check that it generates the 100 kHz 20 V square wave.

A photo of the oscilloscope measurement of the transmission on pentip 0 can be seen in figure 6.7.

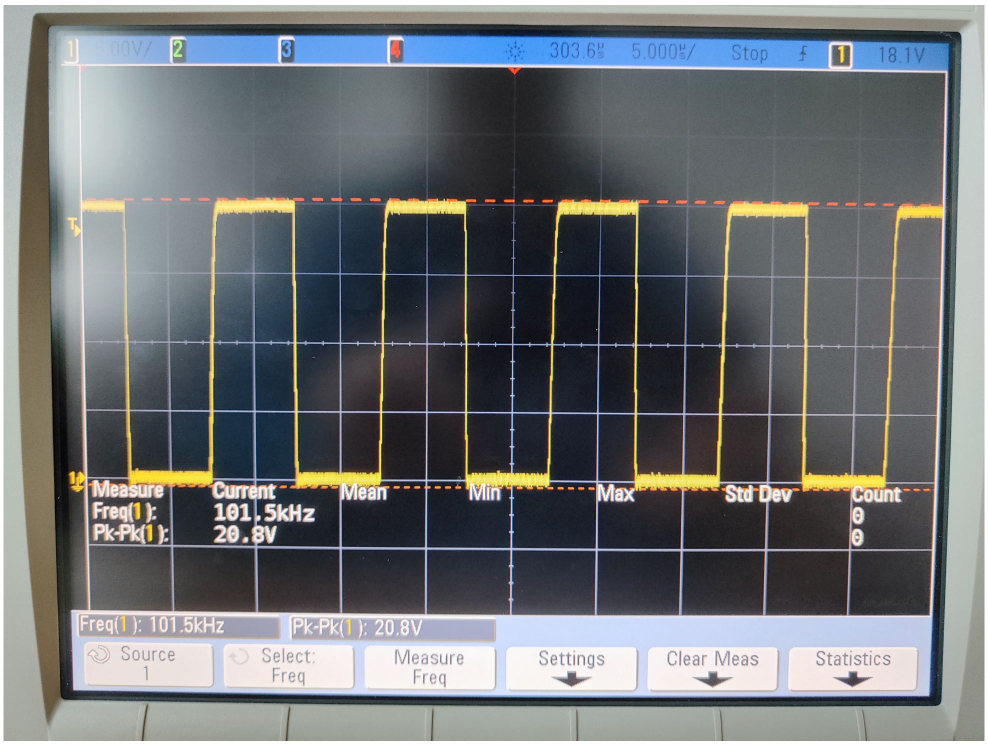


Figure 6.7: Transmission on pentip 0 measured with an oscilloscope

6.5.6 Reception on pentip 0

To test the reception on the pentip 0 it was decided to use a DELL Latitude 5320 laptop which generates an electromagnetic wave sequence on its touch display. This sequence is one of those which the USI Control Logic can detect by correlation.

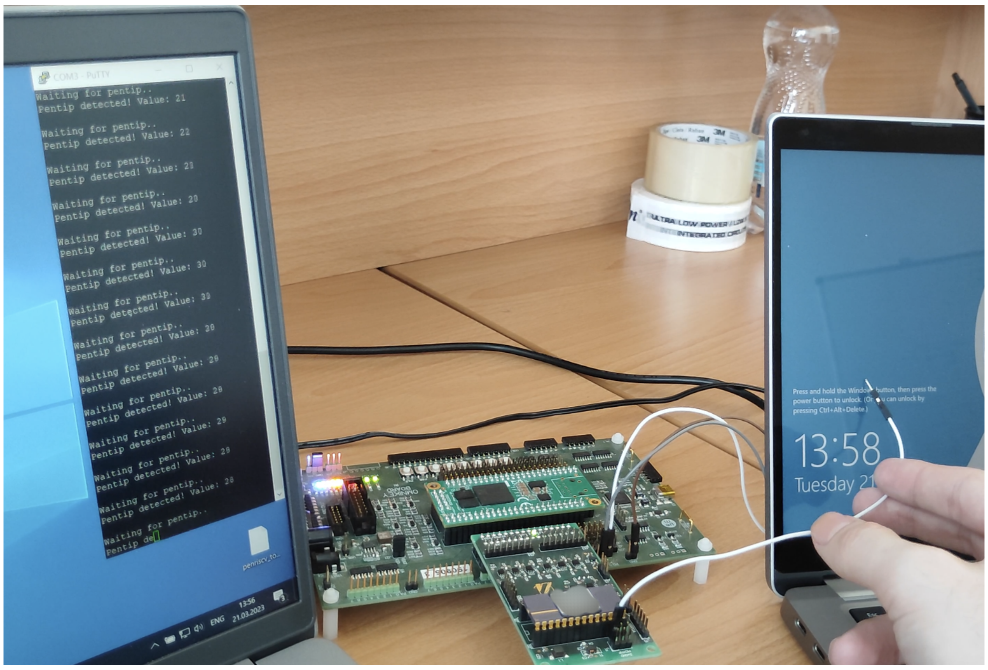
I used a validation script from a validation engineer as a reference when writing the software for this scenario. This software configures the USI Control Logic and HV pentip driver for reception and then waits in a loop until the expected sequence is detected.

When the sequence is detected a message is sent over UART to the PC terminal which also contains the correlation value from the USI Control Logic result register.

I emulated the pentip 0 by attaching a short wire to the pentip 0 pin on the HV pentip driver PCB and during the test, I was moving the end of the wire closer and further to the touch screen of the laptop.

It was already detecting the touch screen roughly 1.5 cm away from the screen with a correlation value in the range of 20-25. When I moved the pentip closer the correlation value got higher with a maximum of 32 when the end of the wire was touching the screen. 32 is the highest possible value of the correlation.

A photo of the reception scenario can be seen in figure 6.8.



```
Waiting for pentip..
Pentip detected! Value: 21

Waiting for pentip..
Pentip detected! Value: 22

Waiting for pentip..
Pentip detected! Value: 23

Waiting for pentip..
Pentip detected! Value: 28

Waiting for pentip..
Pentip detected! Value: 30

Waiting for pentip..
Pentip detected! Value: 30

Waiting for pentip..
Pentip detected! Value: 30

Waiting for pentip..
Pentip detected! Value: 30

Waiting for pentip..
Pentip detected! Value: 29

Waiting for pentip..
Pentip detected! Value: 28

Waiting for pentip..
Pentip detected! Value: 29

Waiting for pentip..
Pentip detected! Value: 29

Waiting for pentip..
Pentip detected! Value: 28

Waiting for pentip..
Pentip de
```

Figure 6.8: Reception on the pentip 0 from DELL laptop touch screen

Chapter 7

Physical implementation

When I finished the RTL implementation and checked the functionality of the system through test bench simulation and FPGA validation, the next step was to implement the design into actual logic gates using selected technology.

7.1 Technology

It was important to keep the same properties for physical implementation as close to the ones used in the CoolRISC-based system. That is why I used the same 180 nm process technology and corner[3] developed by EM Microelectronic.

The corner I used for the physical implementation was SS_0v88_85C. That translates to slow-slow type cells at 0.88 V and 85 C temperature. This corner was used for the max[4] corner.

7.2 Synthesis

The first step in the physical implementation is the synthesis of the design from RTL to actual logic gates. To do this I used the Design Compiler Ultra [28] synthesis tool from Synopsys. For this tool, I used TCL synthesis scripts which were provided to me by the company but I modified them to fit the PENRISCV design.

These scripts load the RTL design files and the required technology libraries into the Design Compiler and provide specified configurations and corners for the synthesis.

[3]A corner is a set of conditions that the chip is expected to operate in. Mainly voltage, temperature and type of cells. This decides the physical characteristics of the cells used in the corner.

[4]Max corner is a scenario where each cell in the technology features the slowest switching speeds resulting in the largest delays on paths.

I used the Design Compiler Ultra in ”topographical mode” when running the synthesis. Normally the synthesis uses a ”wire load model” to estimate wire length (capacitance and resistance). The values used for these estimations are provided by the library files for the technology. Topographical mode is more aware of the estimated physical size of each cell and uses a rough floor plan for the wire length calculations instead of just using the ”wire load model”.

The synthesis goes through two phases. The first phase translates the design from RTL to logic gates and does various optimisations on the design while following the specified design constraints. In the second phase, additional design constraints are applied to the physically implemented design and an incremental synthesis run is executed which further optimizes the implemented design based on newly defined design constraints.

7.2.1 Design Constraints

For synthesis to implement the design correctly into logic gates, it needs design constraints which define the physical and timing properties of the design. These constraints are given to the Design Compiler in the form of Synopsys Design Constraints (SDC) [29].

I defined the physical constraints in constr_phy.tcl. This specifies the driving cell and the capacitance of each IO port of the design. These are applied to the entire synthesis run.

I reused some timing constraints from the synthesis flow of the CoolRISC-based system. I needed to adjust the clock definitions and remove the rules which were no longer fitting for PENRISCV. These constraints are split into two sets of constraints.

The first set of constraints is defined in the constr_tim_4m.tcl. These constraints define main input clocks and generated clocks as well as delays on the IO ports of the design. There are also false path constraints which define which timing paths in the design should not be considered for timing analysis. This is because the timing analysis tool considers all paths and all scenarios and may detect a violation where it cannot happen by design.

The second set of constraints is defined in the constr_tim_post_syn_4m.tcl and it is applied after the first synthesis run before the incremental synthesis run. These constraints mostly contain false path rules for endpoints which are created only after the first synthesis run.

On top of the reused constraints, I also added some PENRISCV-specific false paths that I identified when analysing timing reports.

7.2.2 UPF

The synthesis also needs to know the power supply structure of the design. I used the Unified Power Format (UPF) [30] for this. UPF defines the power supply sources, voltages, power domains and isolation for signals.

I reused the UPF from the CoolRISC-based system as PENRISCV shares the same power domain structure. I made minor adjustments to the signal isolation control as some additional signals needed to be isolated when crossing power domains.

7.2.3 RAM and ROM technology cells

So far I used simple RAM and ROM RTL models in PENRISCV. However, this would not be a good practice to do for physical implementation. Synthesis would attempt to translate the models into actual registers and logic gates but this would be very inefficient. The more efficient way is to use pre-made ROM/RAM cells from the technology library. These cells are going to be smaller in the area and more efficient.

However, there was simply no 32-bit RAM and ROM cell available in this technology library. This forced me to create a RAM wrapper which combines 4 2048-byte 8-bit (data) wide RAM cells and create a RAM block which has 32-bit data width. I implemented this block in ram_4x2kx8_box.sv which was also used as a model in the RTL simulations.

Finding a workaround for ROM was more complicated. Not only was there no 32-bit ROM available, but the closest ROM available was the one used in the CoolRISC-based system and that ROM cell had 22-bit data width. The final workaround was to modify the library files for this ROM cell to feature 32-bit data width. This was done with the help of another digital design engineer in the company.

This modified ROM cell will not physically work. It was just modified to have a 32-bit data interface. The main purpose of this was to be able to do place-and-route and gate-level simulations (which are still possible with the modified RTL model of the cell) and then use the switching activity for power calculations which use this modified library cell.

7.2.4 Outputs

The outputs of synthesis are mainly the synthesized design represented by a netlist in a single Verilog file. Along with it is the written-out SDC file with all constraints which applied to the netlist.

There are also reports generated by Design Compiler Ultra. These reports contain information about the synthesized design. I checked those for any errors or missing constraints. Part of these reports was also a basic static timing analysis which provided information about possible timing setup and hold violations. These reports are limited because there is no clock tree in this synthesized design. A more detailed STA was done after the place-and-route step.

7.2.5 Area results

The total area of dig_top ended up 2.25x larger in comparison to the CoolRISC system. The difference was mostly made out of the CPU being 9 times the size of CoolRISC and the RAM being 4 times the size of the RAM in the CoolRISC system.

7.3 Place-and-route

I gave the synthesis outputs to a place-and-route engineer who created a square floorplan for the design and placed it. I was helping him with any timing violations and my insight into the design so he could have a better idea of how to balance the placed clock tree.

7.4 Static Timing Analysis

The place-and-route engineer was able to successfully balance the clock tree so that the design did not contain any setup and hold violations. I received the post-layout netlist as well as the Standard Parasitic Exchange Format (SPEF) [31] file which is used for representing parasitic data of wires in the design.

I used the PrimeTime [32] tool made by Synopsys to run my own STA on the post-layout netlist that I was given by the engineer. To load the netlist into the tool along with technology libraries, SPEF and SDC I used scripts given to me by the company. I checked the reports for timing violations and any other issues and generated a Standard Delay Format (SDF) [33] file which I needed for gate-level simulations.

7.5 Gate-level simulations

To verify the correct functionality of the post-layout netlist I used the same verification tests that I used for RTL simulations. It is important to do this step even if RTL simulations pass all tests because issues may occur once gate delays are introduced into the design. Especially with asynchronous elements in the design.

I added the gate-level configuration to the run_sim simulation script and I ran all tests with the max corner on the post-layout netlist using the generated SDF.

Not all tests were passing initially and some minor fixes had to be implemented. The design with applied fixes went through place-and-route flow and afterwards, all tests passed in the post-layout gate simulation.

Chapter 8

Power simulations and comparison

To estimate the average power consumption of the system I used the PrimePower [34] tool which is part of the PrimeTime suite made by Synopsys. I selected scenarios which were also simulated on the CoolRISC-based system.

8.1 Scenarios

The first step in the power estimations was to select scenarios which were applicable for both PENRISCV and the CoolRISC-based system to get the most accurate idea of the difference between power consumption between both systems. I selected CoreMark and the penriscv_usi_01 test case as power estimation scenarios.

I chose the 6 MHz system clock mode for the CoolRISC system power estimations to perform the comparisons with PENRISCV. This is because this mode provides the highest CPU performance possible in the CoolRISC system. The CPU performance in this mode is at 3 MIPS. PENRISCV's system clock is 4 MHz and the CPU runs at 4 MIPS.

8.1.1 penriscv_coremark_01

CoreMark is dependent on the CPU performance and the duration of its run changes based on how fast the calculations are performed. That means it is a good way to measure overall energy consumed by the calculation because even if the overall power consumption of the system is higher the total energy consumed during the run may be lower depending on the duration.

The CoreMark settings used for power simulations feature the data size set to 1600 to correspond with the CoolRISC-based system configuration and the number of iterations set to 2 to record the switching activity in a reasonable time. The recorded activity during the run ranges from Long Timer enable is set to ’1’ to when it is set to ’0’. This represents the section of the CoreMark run where the actual benchmark is executed.

8.1.2 penriscv_usi_01

I selected this test because it represented the main functionality of the whole PENRISCV system (transmitting and receiving on the pentips). It also behaved differently from the software perspective because here the software did not perform many calculations and instead mostly consisted of register reads, writes and waiting for an event. This means the duration of this scenario was similar in both systems.

I decided to simulate the power consumption of transmitting and receiving separately.

The simulation of the transmitting phase begins at the first rising edge of the pentip 0 and ends on the last falling edge of the pentip 0.

The simulation of the receiving phase begins at the first rising edge of the pentip 0 and a register write into the IRQ Ctrl peripheral indicating the end of the receiving phase by software.

8.2 Switching activity

To calculate the power consumption from simulations I needed to generate the switching activity of the post-layout netlist in these scenarios. I achieved this by running the test bench simulations in Xcelium and enabling the Value Change Dump (VCD) generation. This file contains all switching activity in the design during the simulation run.

8.3 Average power consumption calculations

I used the PrimePower tool for the average system power consumption which was calculated by the tool based on the provided switching activity (VCD), the post-layout netlist and the SPEF.

To run the tool I used scripts provided to me by the company. These scripts load necessary files into the tool and provide specified scenarios for the calculation. I created scenarios (start time, end time, VCD file) which were used by the tool to do the calculations. I obtained the start time and the end time manually from the waveform in a simulation of these scenarios.

8.4 Methodology

To assess the power consumption results correctly, it was important that I first analyse the power consumption across the hierarchy of both systems. PENRISCV is simplified in some ways which are described in previous chapters.

That is why I needed to apply necessary corrections to the power consumption calculations to make the results from both systems more comparable. This is to more accurately estimate the power consumption of the theoretical fully featured system.

8.4.1 Power consumption across hierarchy

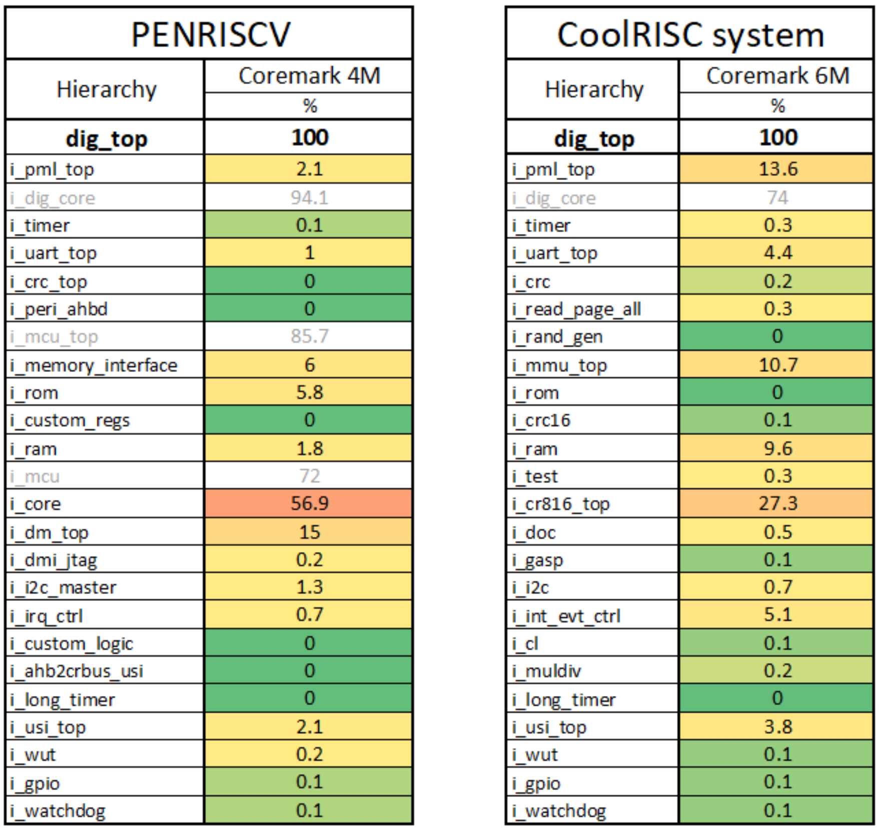
Analysing the CoolRISC-based system I identified the major contributors to the total power consumption during the CoreMark run. Those were the CR816L CPU (27.3 %), PML (13.6 %), MMU (10.7 %), RAM (9.6 %), Interrupt Controller (5.3 %), UART (4.4 %). A detailed write-down of power consumption results across the hierarchy can be seen in Figure 8.1.

Looking at the major contributors which should not be ignored, I observed that PML and MMU are among them and those are modules which had been simplified in PENRISCV. This means that full implementation of these modules would increase the total power consumption of the system.

The rest of the major contributors are either present in the system in their original form or they are replaced by a module with equivalent functionality.

The rest of the modules in the system have only a minor impact on the total power consumption. Those can be simply considered valid in the PENRISCV estimations if they are present in the system in their original or equivalent implementation. Those that are not present in PENRISCV will be considered in corrections.

When I compared the distribution of power consumption between both systems I observed that the distribution is very similar with the CV32E40P CPU being more dominant in the power consumption relative to the rest of the system. The differences that I observed were mainly the ROM and the Debug Module.



PENRISCV

Hierarchy	Coremark 4M
	%
dig_top	**100**
i_pml_top	2.1
i_dig_core	94.1
i_timer	0.1
i_uart_top	1
i_crc_top	0
i_peri_ahbd	0
i_mcu_top	85.7
i_memory_interface	6
i_rom	5.8
i_custom_regs	0
i_ram	1.8
i_mcu	72
i_core	56.9
i_dm_top	15
i_dmi_jtag	0.2
i_i2c_master	1.3
i_irq_ctrl	0.7
i_custom_logic	0
i_ahb2crbus_usi	0
i_long_timer	0
i_usi_top	2.1
i_wut	0.2
i_gpio	0.1
i_watchdog	0.1

CoolRISC system

Hierarchy	Coremark 6M
	%
dig_top	**100**
i_pml_top	13.6
i_dig_core	74
i_timer	0.3
i_uart_top	4.4
i_crc	0.2
i_read_page_all	0.3
i_rand_gen	0
i_mmu_top	10.7
i_rom	0
i_crc16	0.1
i_ram	9.6
i_test	0.3
i_cr816_top	27.3
i_doc	0.5
i_gasp	0.1
i_i2c	0.7
i_int_evt_ctrl	5.1
i_cl	0.1
i_muldiv	0.2
i_long_timer	0
i_usi_top	3.8
i_wut	0.1
i_gpio	0.1
i_watchdog	0.1

Figure 8.1: Power consumption distribution across the hierarchy in CoreMark

In the CoolRISC-based system, ROM has nearly no power consumption because there is simply no activity on it during the CoreMark run. That is because, in the CoolRISC system, the software is being fetched from the Flash memory outside of the system. Whereas PENRISCV fetches software from ROM and therefore ROM in PENRISCV has higher power consumption.

The second difference I observed was the Debug Module accounting for 15 % of the total system power consumption. This was very odd and after investigation, I discovered that it is a design issue which caused this high power consumption. Under normal circumstances, the power consumption of the Debug Module should be near zero. This will be also considered in corrections.

8.4.2 Corrections

Based on the power consumption analysis across the hierarchy between both systems, I came up with the following corrections for the total PENRISCV power consumption calculations.

- Subtract the PML from total power consumption and add power consumption of the PML from the CoolRISC system. PML is simplified in PENRISCV. To make the comparison closer to reality I decided to consider the power consumption of the full PML implementation from the CoolRISC-based system.
- Subtract the Memory Interface from total power consumption and add the power consumption of the MMU from the CoolRISC-based system. The Memory Interface in PENRISCV lacks the Flash memory controller which contributes to the power consumption of this module.
- Subtract the ROM from total power consumption. The CoolRISC-based system fetches software from the Flash which is not accounted for in the measurements. Whereas PENRISCV fetches it from ROM and it is accounted for. Therefore I subtracted the ROM from the total power consumption to make it equal.
- Subtract the Debug Module from the total power consumption. The Debug Module under normal circumstances should draw near no power. To make this comparison more accurate I added the DoC module from the CoolRISC-based system to the PENRISCV total power consumption.
- To account for the modules which are present in the CoolRISC-based system but not in PENRISCV, I decided to add them to the total power consumption if they showed some impact on the total system power in the CoolRISC-based system.

Table 8.1: Summary of the corrections to the total PENRISCV power

Applied corrections to the total PENRISCV power	
+	−
i_test (coolrisc)	i_pml_top (penriscv)
i_pml_top (coolrisc)	i_memory_interface (penriscv)
i_crc16 (coolrisc)	i_rom (penriscv)
i_mmu_top (coolrisc)	i_dm_top (penriscv)
i_doc (coolrisc)	
i_rand_gen (coolrisc)	

8.5 Results

I compared the final power calculations between both systems separately for CoreMark and for the transmission/reception scenario. Comparisons were made for the average power consumption (in Watts) and total energy consumed (in Joules) during each scenario. The energy was calculated by multiplying the average power consumption by the duration of the scenario (in seconds).

The results are presented as the power/energy of PENRISCV relative to the CoolRISC-based system. For example, 0.5x means half the CoolRISC-based system's power/energy and 2x means twice as much power/energy.

These comparisons are made with the previously defined corrections applied.

8.5.1 penriscv_coremark_01

In this scenario, the total power consumption of PENRISCV ends up being 1.75x that of the CoolRISC-based system. The total energy consumed ends up being only 0.54x. This is mainly because PENRISCV was able to complete the CoreMark run nearly three times faster than the CoolRISC-based system.

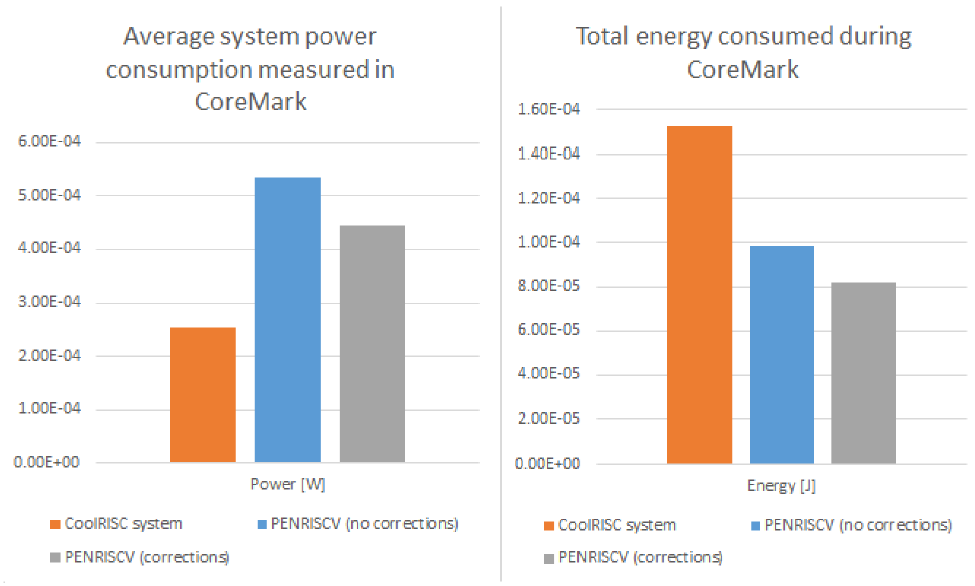


Figure 8.2: Average power and total energy consumption during CoreMark

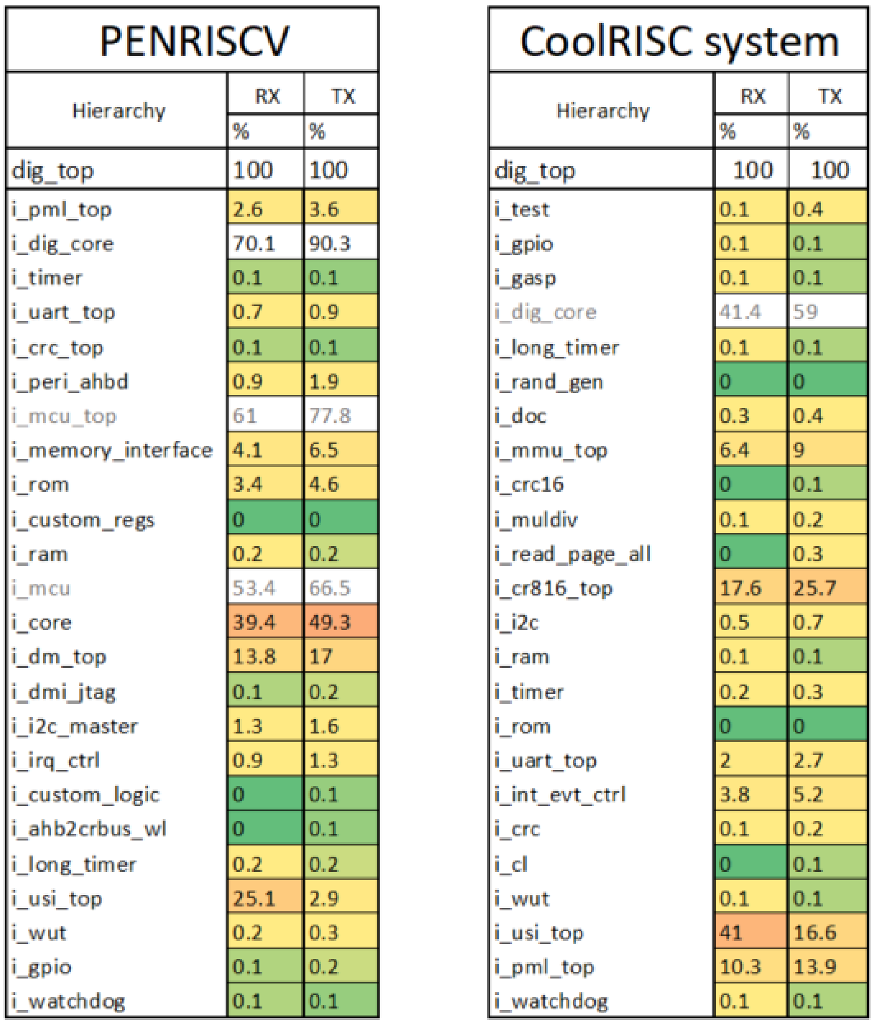
8.5.1.1 RAM access

One interesting observation I made is that the difference in CPUs used also affected the results of the other parts of the system. One of those is the RAM average power consumption. The RAM in PENRISCV has less than half of the power consumption of the RAM in the CoolRISC-based system even though it is four times the size.

Detailed analysis of this revealed that the CoolRISC-based system has to access the RAM far more often than PENRISCV. During the CoreMark run CoolRISC made close to a million RAM read/write accesses whereas PENRISCV only made around 110 000. This is most likely due to the overall shorter run of CoreMark on PENRISCV as well as the fact that CoolRISC has fewer general-purpose registers and has to access the RAM more to store temporary data.

8.5.2 penriscv_usi_01

For this scenario, I applied the same corrections as defined previously but I used the hierarchy measurements for transmission and reception. The distribution of power consumption across the hierarchy remained similar with the only noticeable difference being the USI Control Logic having higher power consumption of 41 % and 16 % of total system power for reception and transmission respectively. A detailed write-down of measured power consumption across the hierarchy can be seen in Figure 8.3.



PENRISCV		
Hierarchy	RX	TX
	%	%
dig_top	100	100
i_pml_top	2.6	3.6
i_dig_core	70.1	90.3
i_timer	0.1	0.1
i_uart_top	0.7	0.9
i_crc_top	0.1	0.1
i_peri_ahbd	0.9	1.9
i_mcu_top	61	77.8
i_memory_interface	4.1	6.5
i_rom	3.4	4.6
i_custom_regs	0	0
i_ram	0.2	0.2
i_mcu	53.4	66.5
i_core	39.4	49.3
i_dm_top	13.8	17
i_dmi_jtag	0.1	0.2
i_i2c_master	1.3	1.6
i_irq_ctrl	0.9	1.3
i_custom_logic	0	0.1
i_ahb2crbus_wl	0	0.1
i_long_timer	0.2	0.2
i_usi_top	25.1	2.9
i_wut	0.2	0.3
i_gpio	0.1	0.2
i_watchdog	0.1	0.1

CoolRISC system		
Hierarchy	RX	TX
	%	%
dig_top	100	100
i_test	0.1	0.4
i_gpio	0.1	0.1
i_gasp	0.1	0.1
i_dig_core	41.4	59
i_long_timer	0.1	0.1
i_rand_gen	0	0
i_doc	0.3	0.4
i_mmu_top	6.4	9
i_crc16	0	0.1
i_muldiv	0.1	0.2
i_read_page_all	0	0.3
i_cr816_top	17.6	25.7
i_i2c	0.5	0.7
i_ram	0.1	0.1
i_timer	0.2	0.3
i_rom	0	0
i_uart_top	2	2.7
i_int_evt_ctrl	3.8	5.2
i_crc	0.1	0.2
i_cl	0	0.1
i_wut	0.1	0.1
i_usi_top	41	16.6
i_pml_top	10.3	13.9
i_watchdog	0.1	0.1

Figure 8.3: Power consumption distribution across the hierarchy in transmission/reception scenario

During transmission, the average power consumption and energy consumed ended up at 1.56x and 2.28x respectively of the CoolRISC-based system. I investigated the significantly higher energy consumed by PENRISCV.

I found out that the length of the transmission phase is dependent on the register reads and writes of the USI Control Logic. Because the USI Control Logic uses the AHB2CR bridge every register access takes two clock cycles instead of one. The increased length of the transmission phase increases the overall total energy consumed. In the final product, the USI Control Logic IP will be re-designed to be AHB-lite compliant therefore I decided to make a correction to the duration of the test case by subtracting the duration of the wait cycle of the AHB2CR bridge multiplied by the number of wait cycles during transmission.

By applying this correction the total energy consumed by PENRISCV decreased to 1.72x of the CoolRISC-based system.

During the reception, PENRISCV ended up at 1.53x average power consumption and 1.46x energy consumed of the CoolRISC-based system.

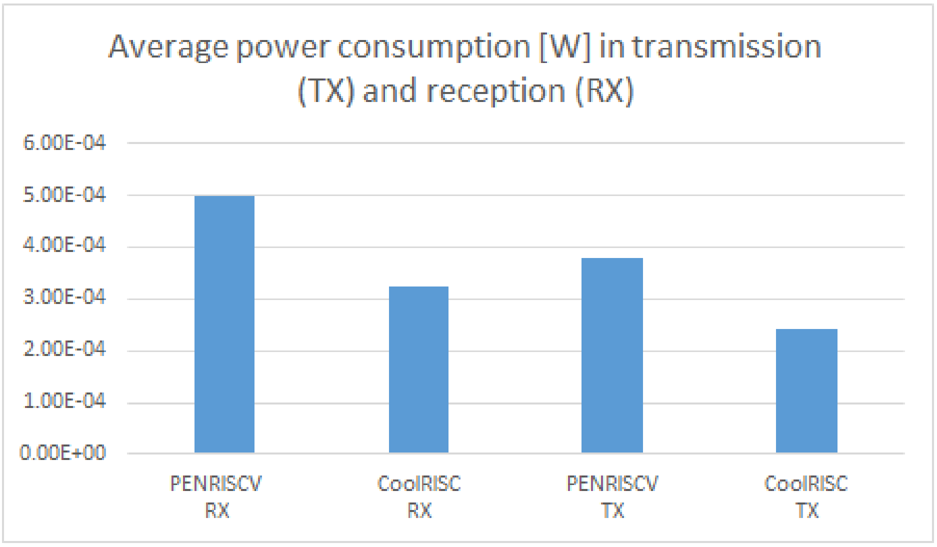


Figure 8.4: Average system power consumption during transmission and reception

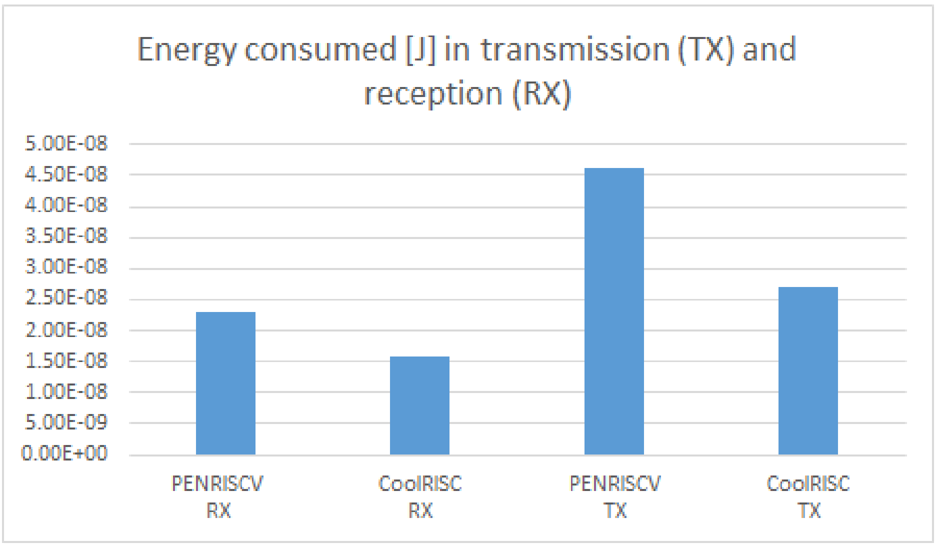


Figure 8.5: Total system energy consumption during transmission and reception

8.5.3 Power measurement conclusion

Looking at the results it is apparent that PENRISCV average system power consumption is around 1.50x-1.80x of the CoolRISC-based system in all scenarios. This comes as no surprise considering the CV32E40P CPU is nearly 10x larger than the CR816L.

However, when looking at the energy consumed results the situation changes significantly and the comparison starts being heavily dependent on the scenario. PENRISCV benefits from a more capable CPU in compute-heavy applications because it can complete the task much faster saving total consumed energy. During CoreMark PENRISCV consumed only 0.54x of the CoolRISC-based system's energy.

In applications which only consist of register reads/writes or waiting for an event PENRISCV is put at a disadvantage. During such operations, there is no way to utilize the performance of the CV32E40P CPU and save time. Meaning the runtime is going to be similar while the average power consumption is higher the whole time.

If such a system is implemented into a graphical pen there would be higher power supply requirements for it due to the higher average power consumption. However, the energy consumed during the operation of the system is more important for the battery life. This gives the RISC-V a chance to be a feasible alternative to CoolRISC if the application is properly tailored for it to utilise its potential.

Table 8.2: Final results relative to the CoolRISC-based system

PENRISCV/CoolRISC		
Scenario	Power	Energy
CoreMark	1.75	0.54
Transmission	1.56	1.72
Reception	1.53	1.46

Conclusion

The goal of this thesis was to analyse the existing CoolRISC-based pen controller SoC, create a system design of a pen controller with the RISC-V CPU platform and then implement it in RTL. The basic functionality of the new implementation was to be verified in a test bench and validated on an FPGA. Then it was to be implemented into 180 nm technology and the physical implementation was used for measurements and estimation of the power consumption of the system. The results of the measurements were to be compared to the existing measurements of the previous CoolRISC-based system. All of these goals were achieved.

Firstly the existing CoolRISC-based pen controller SoC was analysed and core features of the system were identified. Based on this analysis, a system design for the new RISC-V-based system called ”PENRISCV” was created which was then used for the RTL implementation. Part of the analysis was also a detailed comparison of both CoolRISC and RISC-V CPU platforms.

A test bench environment using UVM was created for the verification of basic system functionality. Then the system was implemented onto an FPGA with appropriate design changes to make it functional. Validation was performed on an FPGA. The result of the FPGA validation was a successful pentip reception using a real touch screen.

Physical implementation was performed into 180 nm technology provided by EM Microelectronics. The post-layout netlist was used to simulate the power consumption of the system in simulations.

The power consumption results estimates were compared to the existing results of the CoolRISC-based system. The results show that PENRISCV ends up having higher average power consumption in all tested scenarios but ends up being more energy efficient in compute-heavy scenarios. In scenarios which consist of just register reads/writes and waiting for an event, PENRISCV ends up being less energy efficient.

This thesis will serve as a deciding factor for the next generation of pen controller SoCs when considering the power characteristics of RISC-V in such a system. If RISC-V is selected as the next CPU platform PENRISCV will also serve as a base for building a fully featured SoC. After all, the only remaining design elements not implemented in PENRISCV are the RC oscillator clock, Flash controller, some peripherals and PML functionality related to analogue parts of the system.

Bibliography

1. XEMICS SA. *CoolRISC 816 8-bit Microprocessor Core* [online]. 2001. [visited on 2023-04-26]. Available from: `https : / / www . datasheetarchive.com/pdf/download.php?id=ffbdc36fbf4dc0d29fd1c730dd8933625e8153& type=M&term=CoolRISC816`.
2. Swiss Center for Electronics and Microtechnology (CSEM). *csem.ch* [online]. 2023 [visited on 2023-04-26]. Available from: `https://www.csem.ch/`.
3. RISC-V. *riscv.org* [online]. 2023 [visited on 2023-04-26]. Available from: `https://riscv.org/about/`.
4. OpenHW Group CORE-V CV32E40P RISC-V IP. *github.com* [online]. 2023 [visited on 2023-04-26]. Available from: `https://github.com/openhwgroup/cv32e40p`.
5. About Us. *openhwgroup.org* [online]. 2023 [visited on 2023-04-26]. Available from: `https://www.openhwgroup.org/about-us/`.
6. ANDREW WATERMAN, KRSTE ASANOVI, SIFIVE INC. *The RISC-V Instruction Set Manual* [online]. 2017. [visited on 2023-04-26]. Available from: `https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`.
7. SILICON LABS, INC. *OBI 1* [online]. 2022. [visited on 2023-04-26]. Available from: `https://github.com/openhwgroup/obi/blob/188c87089975a59c56338949f5c187c1f8841332/OBI-v1.5.0.pdf`.
8. RISC-V Instruction-Set Cheatsheet. *itnext.io* [online]. 2022 [visited on 2023-04-26]. Available from: `https://itnext.io/risc-v-instruction-set-cheatsheet-70961b4bbe8`.
9. The D Latch. *allaboutcircuits.com* [online]. 2021 [visited on 2023-04-26]. Available from: `https://www.allaboutcircuits.com/textbook/digital/chpt-10/d-latch/`.

10. Design of Low Voltage D-Flip Flop Using MOS Current Mode Logic (MCML) For High Frequency Applications with EDA Tool. *researchgate.net* [online]. 2015 [visited on 2023-04-26]. Available from: `https://www.researchgate.net/figure/D-Latch-design_fig5_318295016`.
11. D Flip-Flop (edge-triggered). *barrywatson.se* [online]. 2000 [visited on 2023-04-26]. Available from: `http://www.barrywatson.se/dd/dd_d_flip_flop_edge_triggered.html`.
12. CENTRE SUISSE DELECTRONIQUE ET MICROTECHNIQUE SA CSEM. *D-type master-slave flip-flop*. Christian Piguet, Jean-Marc Masgonty, Claude Arm. May. 1999. U.S. Patent 6323710B1. United States. Available also from: `https://patents.google.com/patent/US6323710B1/en?oq=US6323710`.
13. Time borrowing in latches. *vlsiuniverse.blogspot.com* [online]. 2019 [visited on 2023-04-26]. Available from: `https://vlsiuniverse.blogspot.com/2016/08/time-borrowing-in-latches.html`.
14. Ride7. *raisonance.com* [comp. software]. 2023 [visited on 2023-04-26]. Available from: `https://www.raisonance.com/ride7.html`.
15. RISC-V Software Ecosystem. *wiki.riscv.org* [online]. 2022 [visited on 2023-04-26]. Available from: `https://wiki.riscv.org/display/HOME/RISC-V+Software+Ecosystem`.
16. Ibex RISC-V Core. *github.com* [online]. 2023 [visited on 2023-04-26]. Available from: `https://github.com/lowRISC/ibex`.
17. ARM LIMITED. *AMBA 3 AHB-Lite Protocol Specification* [online]. 2006. [visited on 2023-04-26]. Available from: `https://www.eecs.umich.edu/courses/eecs373/readings/ARM_IHI0033A_AMBA_AHB-Lite_SPEC.pdf`.
18. XILINX, INC. *7 Series FPGAs Clocking Resources* [online]. 2018. [visited on 2023-04-26]. Available from: `https://docs.xilinx.com/v/u/en-US/ug472_7Series_Clocking`.
19. XILINX, INC. *LUT* [online]. 2018. [visited on 2023-04-26]. Available from: `https://www.xilinx.com/htmldocs/xilinx2017_4/sdaccel_doc/yeo1504034293627.html`.
20. How do I reset my FPGA? *eetimes.com* [online]. 2011 [visited on 2023-04-26]. Available from: `https://www.eetimes.com/how-do-i-reset-my-fpga/`.
21. Xcelium Logic Simulator. *cadence.com* [comp. software]. 2023 [visited on 2023-04-26]. Available from: `https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html`.

22. CoreMark. *eembc.org* [comp. software]. 2018 [visited on 2023-04-26]. Available from: `https://www.eembc.org/coremark/`.

23. Vivado 2019.1. *xilinx.com* [comp. software]. 2019 [visited on 2023-04-26]. Available from: `https://www.xilinx.com/support/documentation-navigation/design-hubs/2019-1/dh0013-vivado-installation-and-licensing-hub.html`.

24. Vivado Lab 2022.2. *xilinx.com* [comp. software]. 2022 [visited on 2023-04-26]. Available from: `https://www.xilinx.com/support/download.html`.

25. Clocking Wizard. *xilinx.com* [comp. software]. 2022 [visited on 2023-04-26]. Available from: `https://www.xilinx.com/products/intellectual-property/clocking_wizard.html`.

26. KiCad 6.0. *kicad.org* [comp. software]. 2023 [visited on 2023-04-26]. Available from: `https://www.kicad.org/`.

27. IXYS INTEGRATED CIRCUITS DIVISION. *NCD2400M* [online]. 2018. [visited on 2023-04-26]. Available from: `https://www.ixysic.com/home/pdfs.nsf/www/NCD2400M.pdf/$file/NCD2400M.pdf`.

28. Design Compiler Ultra. *synopsys.com* [comp. software]. 2023 [visited on 2023-04-26]. Available from: `https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html`.

29. Synopsys Design Constraints — SDC File in VLSI. *teamvlsi.com* [online]. 2020 [visited on 2023-04-26]. Available from: `https://teamvlsi.com/2020/05/sdc-synopsys-design-constraint-file-in.html`.

30. Unified Power Format (UPF) and Beyond: How to Expand Low-Power Signoff. *blogs.synopsys.com* [online]. 2021 [visited on 2023-04-26]. Available from: `https://blogs.synopsys.com/from-silicon-to-software/2021/10/05/unified-power-format-low-power-design/`.

31. Standard Parasitic Exchange Format. *wikipedia.org* [online]. 2023 [visited on 2023-04-26]. Available from: `https://en.wikipedia.org/wiki/Standard_Parasitic_Exchange_Format`.

32. PrimeTime. *synopsys.com* [comp. software]. 2023 [visited on 2023-04-26]. Available from: `https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html`.

33. Standard Delay Format. *vlsi.pro* [online]. 2013 [visited on 2023-04-26]. Available from: `https://vlsi.pro/standard-delay-format-sdf-file/`.

34. PrimePower. *synopsys.com* [comp. software]. 2023 [visited on 2023-04-26]. Available from: `https://www.synopsys.com/implementation-and-signoff/signoff/primepower.html`.

Appendix A

List of abbreviations used

CPU Central Processing Unit

SoC System on Chip

FPGA Field Programmable Gate Array

RTL Register Transfer Level

DCDC Direct Current to Direct Current conversion

SPI Serial Peripheral Interface

CRC Cyclic Redundancy Check

PLL Phase-Locked Loop

ISA Instruction Set Architecture

ALU Arithmetic Logic Unit

MIPS Million Instructions Per Second

IDE Integrated Development Environment

LED Light Emitting Diode

ASIC Application Specific Integrated Circuit

IO Input/output

PCB Printed Circuit Board

GUI Graphical user interface

XML Extensible markup language

Contents of the attachments

```
readme.txt ......................... Summary of attachments contents
src
  penriscv ..................... PENRISCV implementation source files
    design .4 ipgen ........ XML definition scripts for IPGEN script
      mcu ............................ RTL for mcu_top and CPU
      periphs ............................... RTL for peripherals
      pml ........................................ RTL for PML
      top ................................. dig_top and dig_core
    fpga .4 bin ........................................ Scripts
      constraints ........................... FPGA constraints
      src ................................... FPGA source files
    sw .4 comp .................................. Compilation files
      coremark ............................... Coremark sources
      periphs ........................... Sources for peripherals
      riscv ........................... RISC-V control functions
    syn ..................................... Synthesis constraints
    tb ..................................... Test bench source files
  thesis ............................... Thesis source files in LaTeX
text ................................................... Thesis text
  thesis.pdf ................................... Thesis text in PDF
```