Tampere University

Thomas Szymkowiak

FPGA-BASED PROTOTYPING OF A MODERN MPSOC

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Prof. Timo Hämäläinen
Arto Oinonen, M.Sc.
December 2023

ABSTRACT

Thomas Szymkowiak: FPGA-Based Prototyping of a Modern MPSoC
Master of Science Thesis
Tampere University
Information Technology - Embedded Systems
December 2023

The complexity of computing systems has been increasing exponentially since the invention of the integrated circuit in the 1950s. This increase in complexity has led to the creation of modern computer architectures such as the Multi-Processor System-on-Chip (MPSoC). The development of MPSoCs is a highly complex and resource-intensive process, with verification forming a significant portion of the activities required to produce a viable design. FPGA-based prototyping is a crucial verification activity used to create an accurate, highly performant hardware model of the system or component subsystems.

This thesis presents an overview of FPGA-based prototyping within the context of modern MPSoC development and an analysis of the application of FPGA-based prototyping within an actual MPSoC development project. The benefits and limitations of FPGA-based prototyping in developing complex ASICs are described in detail.

Keywords: Verification, FPGA, SoC, MPSoC, ASIC, SoC-Hub, RISC-V

PREFACE

Coming to Finland and working as a part of SoCHub while studying Embedded Systems at Tampere University has been the realisation of a dream. I cannot possibly put into words how thankful I am for the experience. First and foremost, I would like to sincerely thank my family for the endless support that they have provided throughout this process. Additionally, the SoCHub team at Tampere University have been key in making the past 3 years so enjoyable. In no particular order, I would like to personally thank Jarkko Passi, Antti Nurmi, Aisha Ahmed, Arto Oinonen, Matti Käyrä, Antti Rautakoura, Timo Hämäläinen and Sakari Lahti.

Tampere, 31st December 2023

Thomas Szymkowiak

CONTENTS

GLOSSARY

AI	Artificial Intelligence
AIC	And-Inverted Cone
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
APU	Application Processing Unit
ASIC	Application Specific Integrated Circuit
ATPG	Automatic Test Pattern Generation
AXI	Advanced eXtensible Interface
BRAM	Block RAM
BSP	Board Support Package
C2C	Chip-to-Chip
CDC	Clock Domain Crossing
cJTAG	Compact JTAG
CMOS	Complimentary Metal Oxide Semiconductor
CPI	Camera Parallel Interface
CPU	Central Processing Unit
CTS	Clock Tree Synthesis
CVA	Core-V APU
DDR	Double Data Rate
DDR3	Double Data Rate Generation 3
DDR4	Double Data Rate Generation 4
DFT	Design For Test
DMA	Direct Memory Access
DPRAM	Dual Port RAM
DRV	Design Rule Violation
DSP	Digital Signal Processing
DUT	Design Under Test

EDA	Electronic Design Automation
ETH	Ethernet
FLL	Frequency Locked Loop
FMC	FPGA Mezzanine Card
FPGA	Field Programmable Gate Array
GALS	Globally Asynchronous Locally Synchronous
Gbps	Gigabit per Second
GCN	Graphical Convolutional Network
GDB	GNU Debugger
GDS	Graphic Design System
GLS	Gate Level Simulation
GNU	GNU's Not Unix
GPU	Graphics Processing Unit
HAL	Hardware Abstraction Layer
hart	Hardware Thread
HPC	High Performance CPU
HPC FMC	High Pin Count FPGA Mezzanine Card
HSTL	High-Speed Transceiver Logic
HW	Hardware
I/O	Input/Output
I2C	Inter-Integrated Circuit
I2S	Inter-IC Sound
IC	Integrated Circuit
IDE	Integrated Development Environment
ILA	Integrated Logic Analyser
IP	Intellectual Property
IPC	Inter-Processor Communication
ISA	Instruction Set Architecture
JTAG	Joint Test Action Group
LEC	Logic Equivalence Checking
LPC FMC	Low Pin Count FPGA Mezzanine Card
LPDDR	Low Power Double Data Rate

LTSSM	Link Training and Status State Machine
LUT	LookUp Table
LVCMOS	Low-Voltage CMOS
LVDS	Low-Voltage Differential Signalling
LVS	Layout Versus Schematic
LVTTL	Low-Voltage TTL
MMCM	Multi-Mode Clock Manager
MMU	Memory Management Unit
MPC	Medium Performance CPU
MPSoC	Multiprocessor System on Chip
NRE	Non-Reoccurring Expense
openOCD	Open On Chip Debugger
PCIe	Peripheral Component Interconnect Express
PLL	Phase-locked Loop
PMOD	Peripheral Module
PPA	Power, Performance and Area
PULP	Parallel Ultra-Low power Processor
QSPI	Quad-Serial Peripheral Interface
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RTL	Register Transfer Level
SD Card	Secure Digital Card
SDF	Standard Delay Format
SDIO	Secure Digital Input Output
SerDes	Serializer Deserializer
SMD	Surface Mount Device
SoC	System on Chip
SPI	Serial Peripheral Interface
SPRAM	Single Port RAM
SRAM	Static RAM
STA	Static Timing Analysis

SV	System Verilog
SW	Software
SWD	Serial Wire Debug
SysCtrl	System Control
TAP	Test Access Port
TAU	Tampere University
TCL	Tool Command Language
TTL	Transitor-Transistor Logic
TUNI	Tampere Universities
UART	Universal Asynchronous Receiver Transmitter
uDMA	Micro-Direct Memory Access
URL	Uniform Resource Locator
USB	Universal Serial Bus
UVM	Universal Verification Methodology
VHDL	VLSI Hardware Description Language
VIO	Virtual Input/Output
VIP	Verification IP
VLSI	Very Large Scale Integration

1. INTRODUCTION

The complexity of modern MPSoCs can result in complex development life cycles, with multiple streams of coupled development tasks being performed in parallel. Verification activities require a high percentage (40% - 50% [37]) of total engineering resources within projects and contain multiple complimentary activities. One of the commonly used but rarely analysed activities is FPGA-based prototyping. FPGA-based prototyping is typically used to provide an accurate, highly performant hardware model, which can interface to real components within a lab without incurring high material costs. This thesis documents an overview of FPGA-based prototyping methodologies, a review of the current related technologies and an analysis of a real-life application within the SoCHub project. Measuring the effectiveness of FPGA prototyping is a challenging task, as it is an activity which is mainly used to complement adjacent, metric-driven verification activities. This thesis attempts to evaluate the strengths and weaknesses of FPGA prototyping by establishing high-level objectives, which guide the implementation and testing scope, and a qualitative analysis of the activity is performed.

This thesis is composed of six sections. Section 2 provides an overview of modern SoC development and the related verification challenges. Section 3 delivers an overview of FPGA prototyping methodologies, current technologies and related work. The context of the FPGA prototyping analysis is given within Section 4, in which the Ballast SoC project, architecture and prototyping objectives are detailed. Section 5 follows this by presenting the detailed implementation of the FPGA prototyping configurations used as a part of the Ballast SoC verification activities. A review of the results and identified limitations of the prototyping activities are provided within Section 6. Section 7 contains the conclusion and suggestions for future work items that could build upon this thesis's content.

2. SOC DEVELOPMENT

2.1 Overview

Following Moore's Law, the component density of an IC has approximately doubled every two years and historically the complexity of computing systems has increased at an equivalent rate. Computing architectures have evolved to satisfy the functional and performance requirements placed upon computing systems to manage this complexity [26]. A major milestone in the evolution of computer architecture design was the introduction of the SoC concept in the 1980's. An SoC can be summarised as a heterogeneous VLSI system that utilises a traditional processor core in combination with ASICs contained within a single IC. This combination leverages the optimised performance that a custom hardware solution provides and the operational flexibility that a general-purpose CPU running software offers [25].

Another significant evolution of architecture, conceived in the early 2000s, was the MPSoC. An MPSoC builds upon the traditional SoC concept by introducing multiple, independent, programmable processors connected to memories and other system components via a dedicated interconnect. The addition of multiple CPU instances within an MPSoC increases the achievable performance and flexibility of the software running on the system. This combination makes MPSoCs suitable for use in embedded multimedia and high-speed data communications applications, where there is a need for high performance that can be implemented to meet strict quantitative goals [55]. Figure 2.1 represents a basic MPSoC architecture.

2.2 Verification Challenges

During the development of hardware for ICs, the design must be sufficiently tested to ensure that all bugs are removed before tape-out, as the time/cost impact of fabricating another chip iteration is significant and often unacceptable. Verification is the most demanding task in terms of engineering effort, with typically 40-50% of project resources allocated to functional verification on a hardware development project [36].

For a modern MPSoC, the typical development life-cycle is highly complex. Due to the demands of project schedules and multiple activities being coupled, several activities are

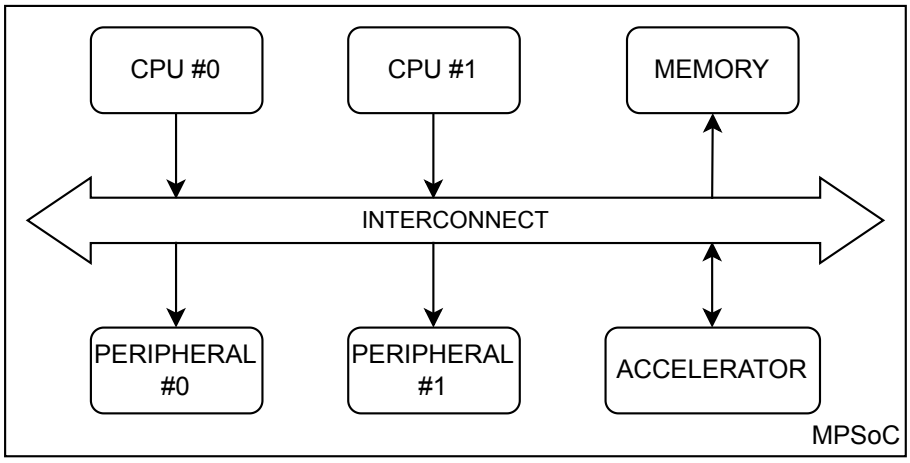

Figure 2.1. *Example of a Basic MPSoC Architecture*

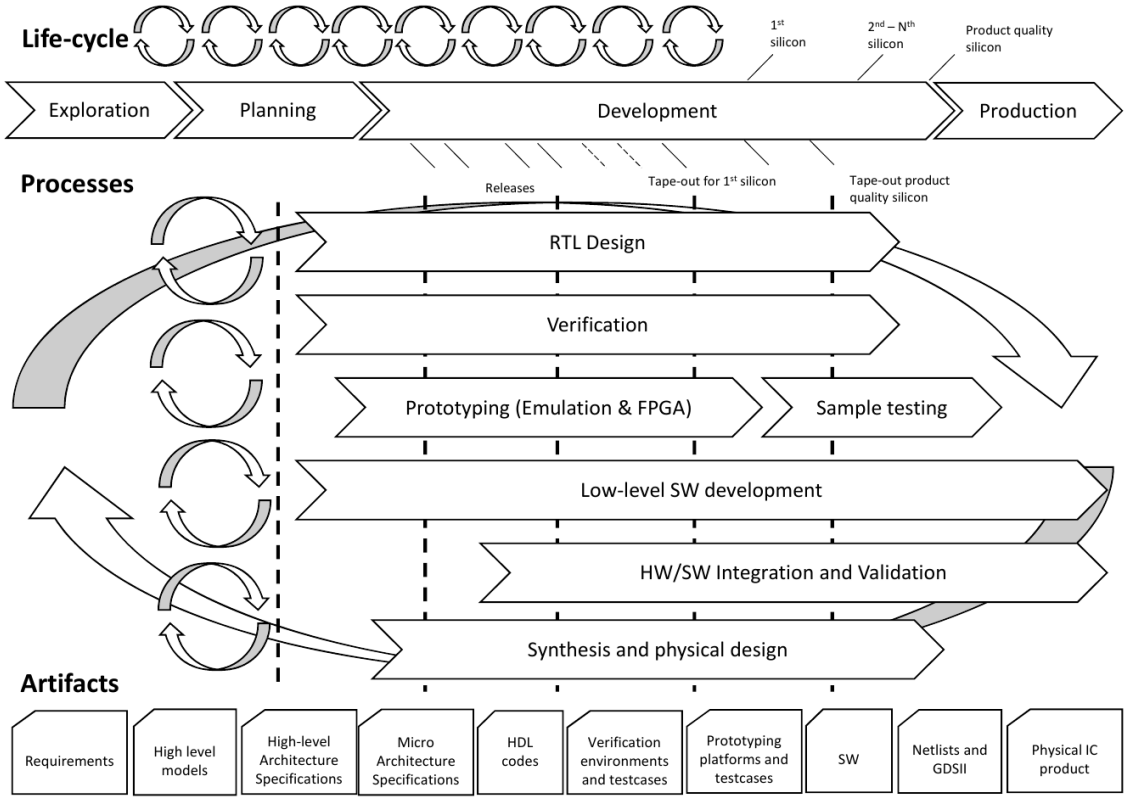

Figure 2.2. *MPSoC Design Life-cycle, processes and artefacts [44]*

often executed in parallel. These include RTL design, SW development, Physical Design and Verification [44]. Figure 2.2 illustrates the complex and parallel nature of modern MPSoC project execution. It is imperative to use design and verification techniques to accelerate the de-risking of hardware and software as early as possible in the life-cycle to minimise the penalties associated with refactoring software/RTL and reduce the time to market.

3. FPGA-BASED PROTOTYPING

3.1 Motivation for Logic Emulation

Simulating RTL is the most widely used method for performing functional verification of modern ASIC designs. As the size of the simulated designs grows, the time it takes to complete the simulation and the amount of host storage/processing required increases. It becomes impractical to run simulations of designs containing multiple processing cores or accelerator blocks for a long enough time to perform realistic use cases.

Hardware emulation (or prototyping) is a design-phase verification technique that integrates a hardware design into a reconfigurable prototyping platform to enable functional testing of the Design Under Test (DUT). It allows the hardware and the low-level software to be evaluated in a realistic performance setting [35]. Hardware emulation platforms are applied within ASIC development to satisfy the following typical requirements [18]:

- Providing a platform which can be used to run the designs at emulation speeds higher than what can be achieved through the use of RTL simulation.
- Being able to emulate the entire system on a single platform.
- Maintaining similarity between the design artefacts used on the delivered solution and the artefacts used in emulation.
- Supplying a high degree of visibility in the emulation for debug purposes.
- Enabling the ability to offer low-cost copies of the prototype for use in hardware-software co-design and field testing.

3.2 FPGA-based Emulation

FPGAs are an example of an effective hardware emulation technology that can execute the design at speeds that would not be possible in RTL simulators [46]. FPGAs have been used in this way for over 30 years. However, until recently, several weaknesses associated with the platforms limited their applicability for prototyping (e.g. low compile time and low area) to small and simple designs. The larger sizes and improved tool performance of the more recent generations of commercial FPGAs have made them more suitable for the task [15][35].

FPGA logic emulation is typically used to replace or augment traditional CPU-based logic or gate-level simulation. The logic circuits within an FPGA are implemented and run in parallel. The parallel performance of CPU-based simulators is limited by the number of ALUs available to execute instructions. In contrast, the performance of FPGA-based emulation is only limited by the amount of resources available on the FPGA. The result is that FPGA-based emulators run logic circuits faster than the equivalent logic executed using a CPU-based simulator. The execution performance of FPGA-based emulators is good. However, there are a number of limitations that exist when emulating a design using FPGA [51]:

- FPGA-based emulators can only support cycle-accurate logic evaluation that is synchronised to the clock of the FPGA design.
- The technology used to implement a design on FPGA differs significantly from the technology used for implementing the ASIC. Therefore, reliable post-synthesis/post-routing timing information cannot be retrieved by running the design on FPGA.
- Debugging designs on an FPGA during runtime is significantly more complex than debugging a design in simulation. Typically, dedicated resources must be reserved within the design to allow in-circuit analysis of the signals within the FPGA.

Depending on the application's requirements, FPGA emulators can be implemented on a single FPGA or across multiple FPGAs. Furthermore, using the FPGA emulator in a standalone, in-circuit or co-simulation configuration is possible. The following subsections provide a summary of each configuration.

3.2.1 Single-FPGA Emulation Platform

The basic form of emulator is a design running on a single FPGA. The design is translated to remove/replace any design structures that cannot be synthesised on FPGA. Additionally, architectural features in modern ASIC design, such as multiple asynchronous clock domains, complex reset trees, test/debug interfaces and multi-port memories complicate the FPGA compilation process. Therefore, modifications are made to the design to simplify these aspects and simplify implementation without compromising the RTL functionality of the DUT [27].

Once this is complete, the RTL of the design is synthesised using the FPGA-specific tooling and mapped to the platform. This approach is the simplest for logic emulation, but the size of the available FPGA platform limits it. Figure 3.1 (left) illustrates a single-FPGA emulation platform.

3.2.2 Multiple-FPGA Emulation Platform

For larger ASIC designs which cannot be prototyped using a single FPGA platform, it is possible to partition the design and implement each partition onto single FPGAs which are connected to each other. Modern multi-FPGA emulation systems are complex systems containing, in some instances, hundreds of FPGAs/memory chips, high-speed interfaces and logic analysers. Typical systems contain multiple boards, each of which contains multiple interconnected FPGAs. The boards are connected together using fixed connections or across a back-plane bus. The design flow for such a configuration is significantly more complex than the single FPGA as in addition to the logic implementation required on each FPGA locally, the implementation also need to be analysed locally across multiple FPGAs. This analysis includes the following steps [51]:

1. Translation of ASIC design into the FPGA circuit representation.
2. Partitioning and global placement of design such that each design partition is able to fit onto a single FPGA.
3. Global routing of signals between partitions.
4. Assignment of interconnected signals to pins on each FPGA. It is common that there is not a sufficient number of pins available to support dedicated connections between the partitioned design, thus time-based multiplexing of signals on each pin is used to enable the design to function correctly.
5. Local synthesis, place and route and bitstream generation for each FPGA on the system.

Figure 3.1 (right) illustrates a multi-FPGA emulation platform connected using a mesh topology.

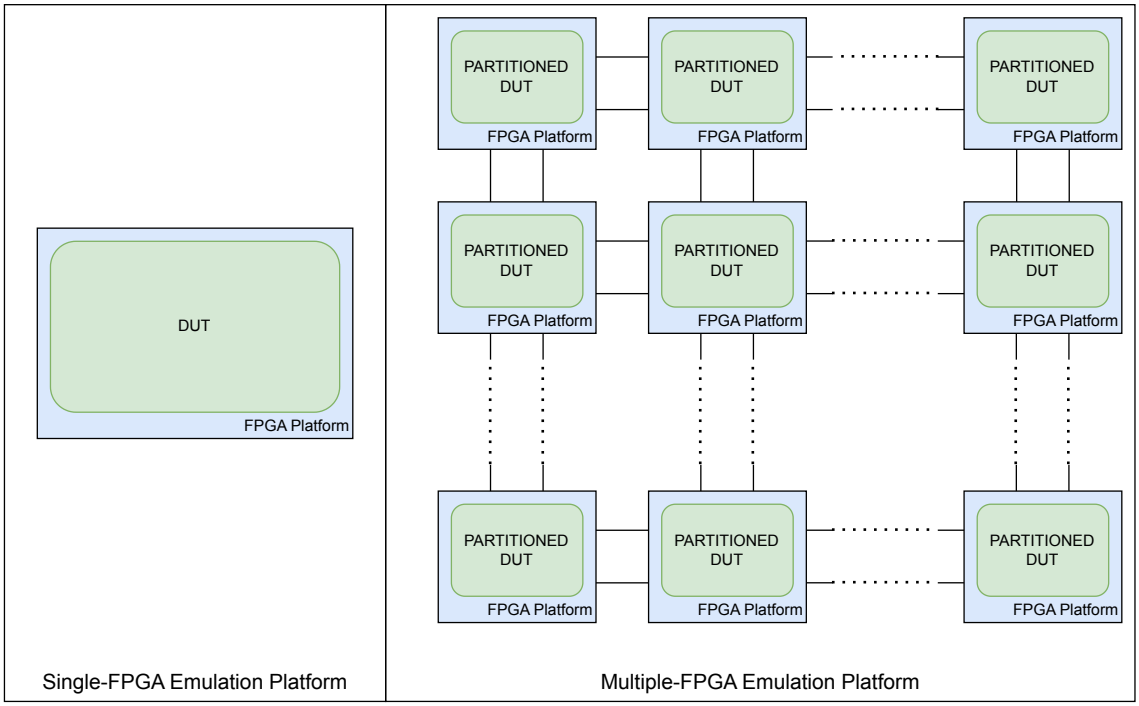

Figure 3.1. *A Single-FPGA Emulation Platform (left) and a Multi-FPGA Emulation Platform Connected in a Mesh Topology (right)*

3.2.3 Standalone FPGA Emulation Platform

As a standalone platform, the design is implemented onto a single or multi-FPGA platform and driven by sources independent of the target system within which the design will be integrated. Figure 3.2 illustrates a typical setup when using a standalone FPGA-based emulation platform. The design can be monitored using integrated debug components or external test equipment. Similarly, stimuli can be sent to the DUT using test logic which has been included within the FPGA design, external dedicated hardware to assist with testing or using test vectors from a host machine, which is usually connected to the FPGA platform using a standard data interface e.g. USB or PCIe. Depending on the testing requirements, a selection or all monitoring/stimulus sources can be used during emulation.

3.2.4 In-circuit FPGA Emulation Platform

For in-circuit emulation, the DUT is implemented on an FPGA-based emulation platform and connected to the target system within which the final design is intended to reside. The stimuli to the emulation platform can then come from the surrounding system components [51] as is shown within Figure 3.3.

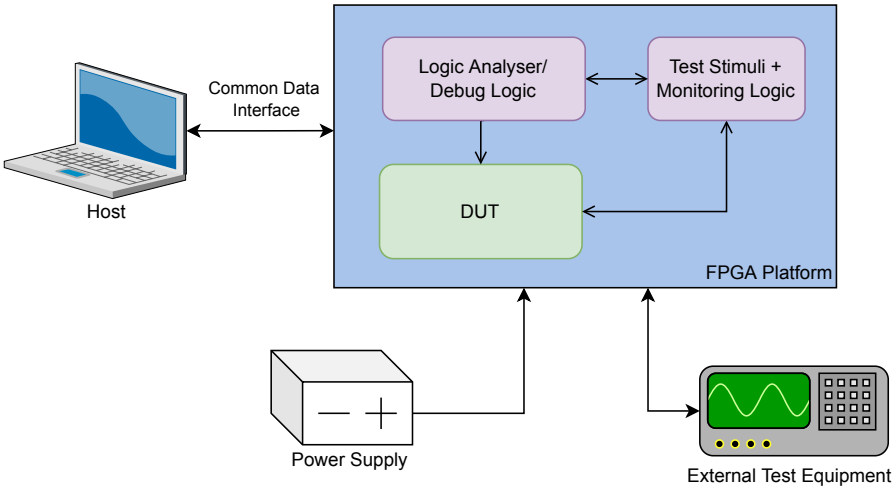

Figure 3.2. *Typical Setup of a Standalone FPGA Emulation Platform*

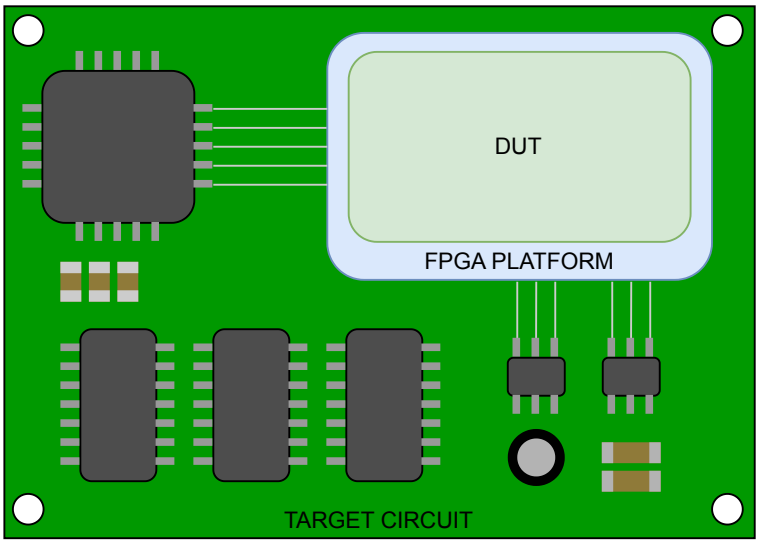

Figure 3.3. *Illustration of an In-circuit Emulation Platform*

3.2.5 Co-simulation FPGA Emulation Platform

It is possible to use an FPGA-based emulation platform as a simulation accelerator. In this configuration (FPGA co-simulation/co-verification), the emulator is connected to a host computer using a common data interface. The host computer simulates the components connected to the design running on the emulator, transferring stimuli between the simulation and the emulator over the data interface [51].

3.3 FPGA Technology Overview

Following the creation of FPGAs in the 1980s, due to the technological limitations of the time, the initial applications of FPGAs were mainly limited to implementing custom 'glue' logic that could be used to connect components. The programmability of FPGAs meant

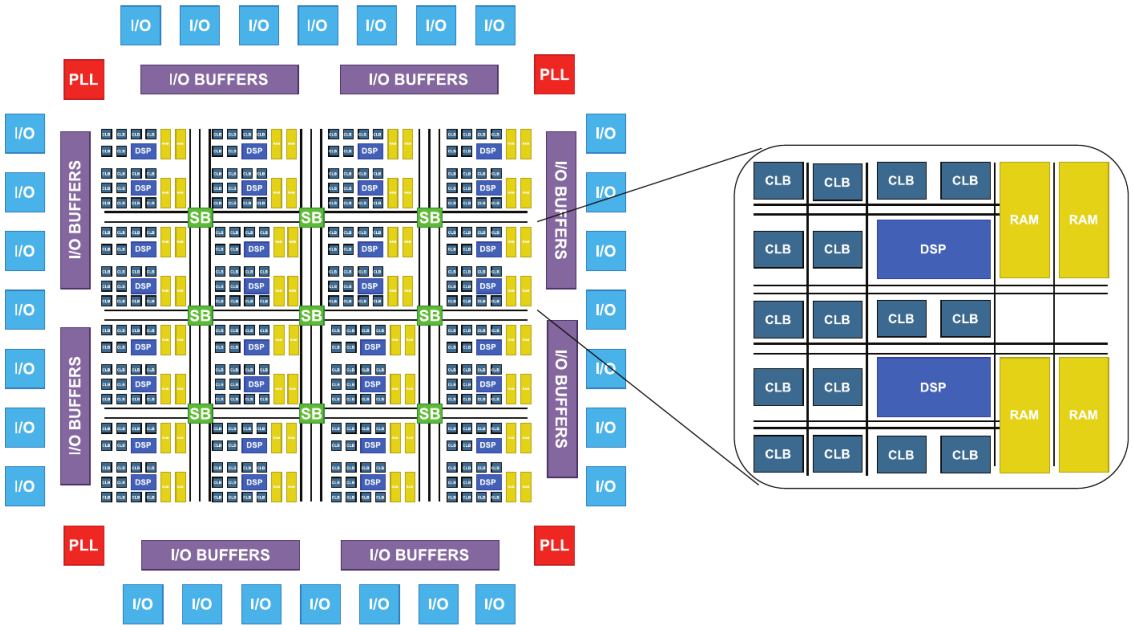

Figure 3.4. *Generic Example of a Modern FPGA's Internal Structure [14]*

that any design errors recognised at a late stage in the development lifecycle could be quickly resolved with minimal impact on schedule or cost. Over time as digital technology advanced and the cost of the technologies reduced, the capabilities of FPGAs and the range of problem domains they can be applied to has increased [56].

The typical architecture of a modern FPGA can consist of a mixture of monolithically embedded components:

- Logic blocks,
- I/O blocks,
- Embedded memories,
- DSP blocks,
- High-speed data transceivers,
- Clock management blocks,
- Signal and clock routing networks,
- Processor cores.

A generic example of the internal structure of a modern FPGA can be seen in 3.4.

The following section provides an overview of the most common components and features of modern FPGAs, focusing on AMD-Xilinx FPGAs.

3.3.1 Logic Blocks

FPGA logic blocks contain a combination of n-bit LUTs, interconnecting logic components and D-type storage elements. The LUTs are composed of n-bit input, 1-bit output SRAM blocks that are used to perform arbitrary combinatorial operations. The interconnecting logic enables routing between the logic blocks and to the I/O components on the FPGA [57]. The storage elements enable sequential operations and, in modern FPGAs, can be configured as latches or flip-flops. In addition to the basic functions some FPGA vendors use logic blocks to implement additional functionality such as shift registers, distributed RAM, LUT-based ROM, arithmetic operations and multiplexing [30]. The terminology used to reference logic blocks varies between vendors. For example, AMD-Xilinx refers to logic blocks using the term "configurable logic block (CLB)", while Intel uses the term "adaptive logic module (ALM)". The exact contents of these blocks vary between vendors and architectures.

3.3.2 IO Blocks

FPGAs use I/O interfaces to route signals in and out of the chip. Modern FPGAs support several different I/O standards and the specific standard assigned to an FPGA pin is typically statically configurable. CMOS, LVCMOS, LVTLL HSTL and LVDS are examples of some of the I/O standards supported by modern FPGAs [9]. In addition to the standard, the FPGA pins can be configured as input, output or bidirectional. The FPGA I/O interfaces and configuration logic are contained within a block referred to as the IO Block on AMD-Xilinx FPGAs. In addition to the configurable aspects already mentioned, the IO Blocks also include additional resources, such as voltage translators, buffers, registers and resistors [48]. The aforementioned configurable I/O properties allow an FPGA to communicate with external components using various interfaces.

3.3.3 Memory Blocks

A memory block, or "Block RAM" (BRAM), is a discrete, fixed-sized block of SRAM memory within an FPGA. It can often be configured to operate in various modes depending on the user requirements, e.g. FIFO, SPRAM, DPRAM. These blocks are typically provided to enable fast and local memory access. BRAMs can be combined to provide the capability of a larger memory without the need for a larger dedicated memory component. There are often many instances of BRAM available within the FPGA. However, the number of blocks and the capacity of each block vary between platforms, and they are typically well-distributed across the FPGA to aid with signal routing. To demonstrate the variable BRAM usage within FPGAs, at the time of writing this thesis, the Xilinx Vitex Ultrascale+ series advertises a rather large BRAM capacity of 94.5Mb [29], while the Lattice iCE40 LP640

offers a relatively modest 64kb [47].

In addition to 'traditional BRAMs', some modern FPGAs now include multiple embedded memory options to help address the increasing need for fast local memory access in FPGA designs. An example is the availability of 'UltraRAM' within the Xilinx Ultrascale range of FPGAs. These are larger (288kb) but less well-distributed blocks of memory to provide an intermediate between the fast/small BRAMs and large/slow off-chip memories such as DDR [31].

3.3.4 Routing Networks

A programmable interconnect is used to flexibly connect all of the components mentioned above and create a complete circuit. A typical routing interconnect comprises an array of wires and switch matrices, the layout of which is architecture-dependent.

One of the challenges EDA vendors face is to ensure that the signal routing distances between the elements within a circuit are as small as possible to minimise propagation delays. Arrangements of logic and routing resources are selected in an attempt to address this. For example, a typical routing style used within modern FPGAs is the "Island style", in which horizontal and vertical channels connect the FPGA functional elements [57] (see Figure 3.4). Furthermore, it is typical for an FPGA to use dedicated routing for wires for the clock pins of synchronous elements. These are specially designed to minimise propagation delay, clock skew and jitter. In modern FPGAs using small technology nodes, routing is the factor which limits the FPGA performance. As a result, some modern devices have started to include register pipelines within routing paths to guarantee better data throughput [23].

3.3.5 DSP Blocks

To perform high-speed arithmetic operations on values with large word lengths, hard DSP blocks are implemented within the FPGA fabric. These blocks typically contain pipelined configurable arithmetic units that allow users to efficiently perform complex arithmetic which is standard in applications such as digital signal processing [16].

3.3.6 High-speed Transceivers

Modern FPGAs also include high-speed data transceivers to provide a SerDes capability. The operation and performance of the transceiver is platform specific. For example, AMD-Xilinx Virtex Ultrascale devices contain GTY-transceivers to provide 400G networking capabilities which would otherwise be not be possible [8].

3.3.7 Embedded Processor Cores

If there is a requirement to have a high degree of configurable control within a design, it is common to implement one or more "soft" processor cores within an FPGA design. The operating frequency and performance achieved by such soft processor cores on an FPGA is lower than can be achieved by an equivalent CPU core implemented as an ASIC due to the reasons outlined within section 3.4. To try and address the continuous need for improved efficiency and performance, some vendors offer hybrid solutions within which a "hard" CPU core is embedded within the FPGA chip alongside the traditional FPGA fabric. For example, the AMD-Xilinx Zynq-7000 SoC contains a dual-core Arm Cortex-A9 processor block that can be interfaced to the FPGA fabric using AXI connections [23].

3.3.8 Other Components

In addition to the main components listed above, modern FPGAs can contain other hardware peripherals such as media codecs, network controllers, external memory controllers (e.g. DDR4 and HBM), high-performance ADCs and DACs. However, the existence of these components is highly variable, depending on the FPGA platform. A detailed description of these components is beyond the scope of this thesis and therefore they will not be covered in detail.

3.4 FPGA and ASIC Comparison

3.4.1 Technologies

The mainstream approach that is used to perform the physical design of SoCs is called semi-custom design. Within this approach, the cells of a standard cell library are used to construct the logic of the design. The cells within the library are composed of logic gates, flip-flops, latches, IO pads, clocking components, SRAM memory cells and other components [23]. During the physical design stages of an ASIC, the cells which make up the design can be freely placed and connected. This freedom can be used to optimise the design implementation in a way that isn't possible within the fixed programmable logic contained within an FPGA (see Section 3.3). Furthermore, the semi-custom approach enables the construction of complex clock and reset tree structures, fine-grained power management logic and memory customisations within ASICs which are not feasible within modern FPGAs due to the fixed nature of the underlying hardware. The result of this is that a typical semi-custom ASIC can achieve a clock frequency which 3-15 times faster, with a 50% reduction in area when comparing to an equivalent FPGA implementation [23].

While the achievable performance of an ASIC is higher than that of an FPGA, the devel-

opment costs and time associated with an ASIC solution are significantly higher than an FPGA implementation. This can be mainly attributed to the high NRE cost that must be paid to develop the mask set which is used to fabricate the ASIC. In contrast, the NRE cost for FPGA development is essentially zero as the programming of the design onto the platform is a straightforward activity [23].

3.4.2 Development Flows

The development flow is a term used to describe the progressive stages of development which must be completed to progress from an FPGA or ASIC design specification to a fully functioning and tested end-product. ASICs and FPGAs share the same high-level steps to create a design from RTL, but significant practical differences need to be considered in the development of each.

The general development flow for an FPGA can be seen in Figure 3.5. Most of the tasks in the flow are completed using the same tool, typically provided by the same vendor that produces the FPGA.

In contrast, the equivalent flow for a modern ASIC can be seen in Figure 3.6. The main difference can be seen in the number of discrete steps required for verification and physical design, mainly because the FPGA is already fabricated and therefore does not need to be verified to such a high standard before the design is released. Additionally, it can be seen that the ASIC flow is separated into front-end and back-end stages. The transition between the front-end and back-end stages occurs over the logic synthesis stage, with the front-end only containing activities independent of the selected fabrication technology or ASIC chip instance [23].

3.5 Related Work

Within [33], the implementation of a single-board FPGA prototype (described within 3.2.1) for the LEON3 SoC is presented. In the final prototype, the entire SoC is implemented using a single Xilinx XC3S1500 FPGA that is mounted on a PCB. The PCB is shared with several peripherals connected to the FPGA pins, enabling the testing of the prototype SoC peripherals using representative hardware. The prototype verification methodology of the SoC is presented and shows that the individual prototype subsystem IPs from the SoC were initially verified as standalone designs. Once this was complete, the IPs were integrated, and the entire SoC prototype verification was performed. The details of how the ASIC-specific components, such as IO pads, PLLs and SRAMs, were modified to allow FPGA synthesis are provided. Once the SoC prototype was available, it was possible to perform HW/SW co-design with ease, as the platform was able to run a Linux-based operating system.

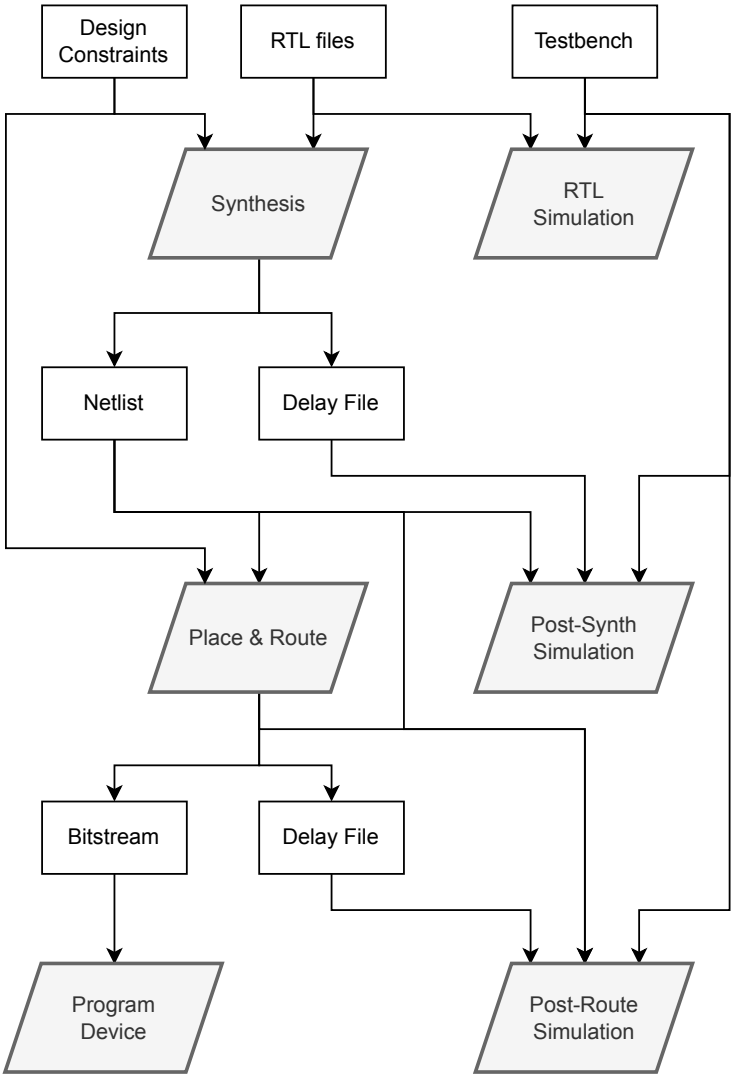

Figure 3.5. *Typical FPGA Development Flow*

An example of a large SoC implemented using multiple FPGAs (described within 3.2.2) can be found in [21]. The focus of the work is the creation of a high-bandwidth, low-latency, scalable interconnect which can be easily re-purposed for other designs requiring multi-FPGAs prototyping platforms. The SoC design is logically partitioned, and each partition is placed onto physically separated FPGAs. In addition to the DUT, logic is added that performs the routing and switching of data flowing in and out of the local FPGA design via a custom bridge. The routing and switching logic is designed to allow the FPGAs to communicate in direct link or routed mode, resulting in a topology-agnostic architecture. Furthermore, transparent error checking and an error correcting protocol are implemented to remove the bit-errors expected when utilising high-speed interfaces. The additional logic connects the DUT and the high-speed transceivers on the selected FPGA platforms. The external pins of the transceivers are connected using SATA cables. The work claims that using this generic interconnect, adapting a single vector processing system design from a single FPGA configuration to a 16 FPGA configuration took only a

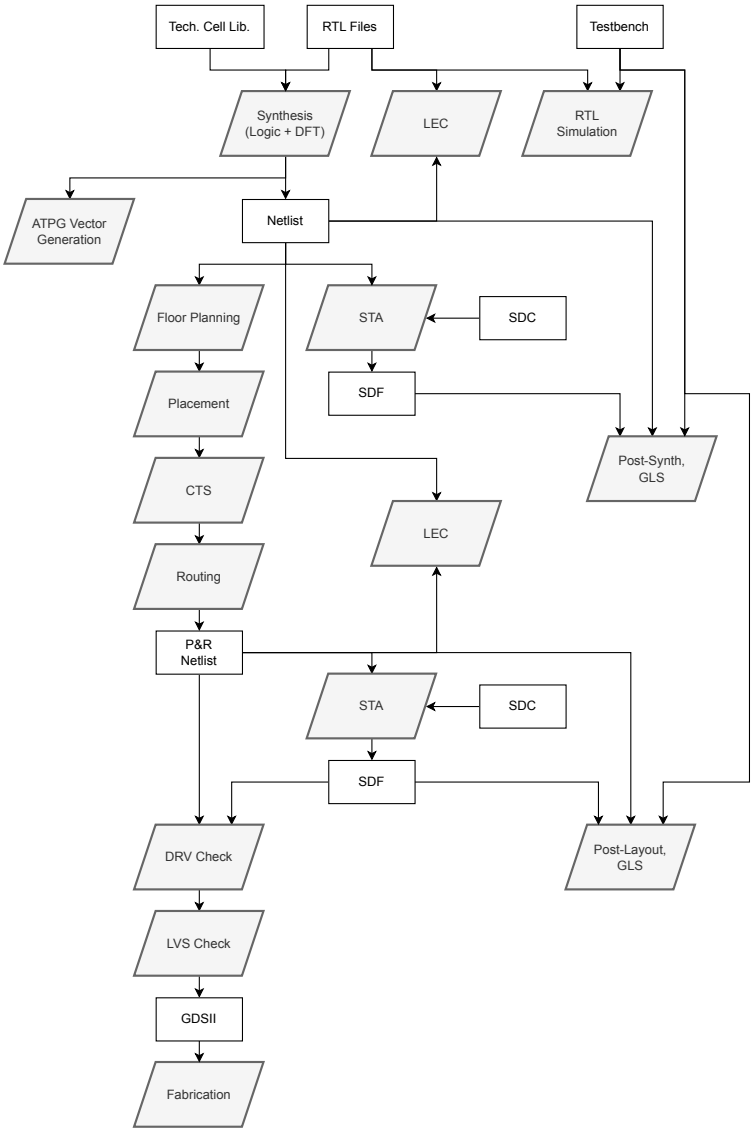

Figure 3.6. *Typical ASIC Development Flow*

single day of effort [21].

As described within 3.2.2, the partitioning of designs onto multiple FPGA prototyping configurations can be highly complex. This complexity can lead to delays and uncertainty in project schedules and create the need for experienced engineers to perform the task. [52] looks at solving the complexity associated with partitioning and mapping a single SoC onto a multi-FPGA configuration by applying deep learning methods to reduce the time to market. The work redefines the process of partitioning and mapping as a high-level task: group the SoC IPs into clusters that do not require more resources than are available on each FPGA and then ensure that the number of signals between the IP clusters is minimised. A graph convolutional network (GCN) is used to generate the FPGA-level clusters as they are proven effective for learning rich representations of nodes and edges in a graph [52]. The SoC IPs can be represented as nodes, and the connections between them can be described as edges of a graph, making a GCN suited for this task. Addi-

tionally, the FPGA logic resource constraints are fed into the constraining of the clusters. Once this is complete, a constrained greedy approach is used to map the DUT to a multi-FPGA platform and the clusters with the most connections between them are placed close together. This approach was applied to map three different SoCs containing 100-150 IPs. In parallel, the same mapping task was also completed manually to act as a control. The results show that the automated partitioning and mapping process performed the mapping more efficiently than the manual attempt for each SoC, with an estimated saving of one month of engineering effort [52].

Compared to simulation-based functional verification of hardware, measuring the functional coverage achieved when using FPGA prototyping to verify a design is complex. [38] looks at implementing a synthesisable active agent and coverage collecting component, which can be integrated into an FPGA prototype to improve input stimuli quality and perform test coverage calculations.

The work focuses on the hardware implementation of a USB3.0 IP core, which is to be verified using an FPGA prototype. The prototyped IP core is driven using an external USB3.0 host device and the output of the IP core is captured using a CPU core, which is embedded within the FPGA prototyping platform and connected via an AXI interface. In this configuration, it is impossible to gain functionality coverage within the USB3.0 link layer, such as the Link Training and Status State Machine (LTSSM), as properties such as signal timing cannot be configured on the driving external USB3.0 host. This work implements an active agent component between the DUT and the external USB3.0 host, which can be configured to set the signal timing of the link layer and therefore, freely control the stimuli. The embedded CPU which collects the DUT output data is also connected to the active agent to configure the timing parameters. A coverage collector application is executed on the CPU, creating a closed-loop system that can then be used to thoroughly test the time-sensitive aspects of the link layer and report on the coverage achieved. The active agent is designed to minimise resource consumption on the FPGA and the resulting design utilised 13% of LUTs and 6% of the registers when compared to the DUT [38].

4. BALLAST DEVELOPMENT AND ARCHITECTURE

4.1 SoC Hub

SoC-Hub is a research ecosystem created by Tampere University to bring together stakeholders from academia and industry in Finland to boost research and education in the field of SoC design. One of the targets of SoC-Hub is to produce SoCs at a cadence of one each year. Ballast is the name of the first SoC that was developed as a part of this initiative [43].

4.2 Ballast Architecture

Ballast is an edge-capable, heterogeneous MPSoC which has been taped out on 22nm technology and has a maximum frequency of 1.2GHz. There are seven subsystems contained within the SoC, namely:

- System Control (SysCtrl) subsystem,
- High Performance CPU (HPC) subsystem,
- Medium Performance CPU (MPC) subsystem,
- Chip-to-Chip (C2C) subsystem,
- Artificial Intelligence (AI) subsystem,
- Ethernet (ETH) subsystem,
- Digital Signal Processing (DSP) subsystem [43].

There is also a top-level interconnect which consists of three AMBA AXI4 [34] crossbar components, connected in a cascaded bus matrix topology [22], as shown within Figure 4.1.

4.3 Development Methodology

The SoC is designed using a GALS architecture so that each subsystem was intended to operate within its own clock domain. To help ensure that the development of the SoC was as efficient as possible, each subsystem (and the interconnect) used the same top-

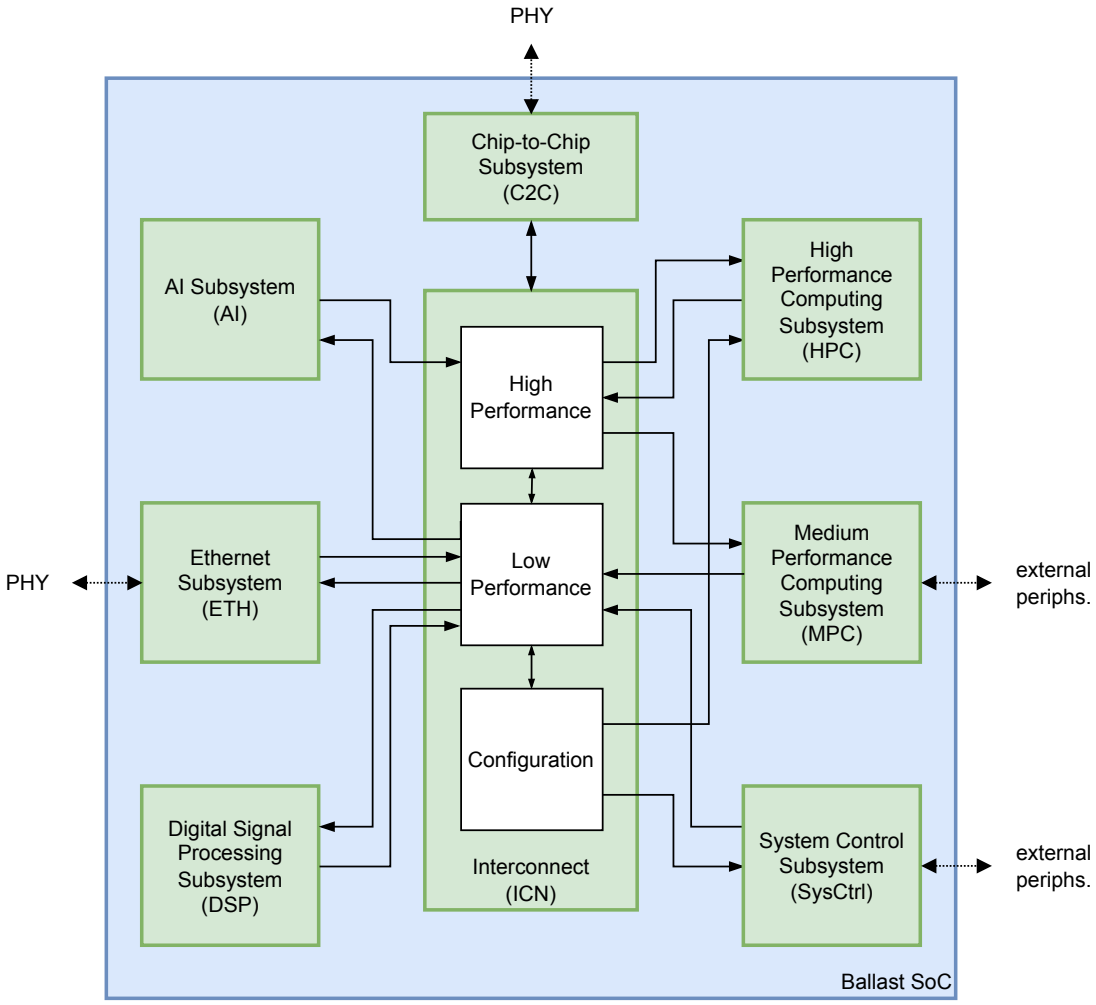

Figure 4.1. *Ballast High-Level Architecture*

level architecture template. This template contained generic definitions for the AXI CDC interface(s), control/status signal synchronisers and clock/reset controls (see Figure 4.2. It enabled the use of a hierarchical ASIC design flow, allowing for concurrent development of subsystems and reduced complexity of the design environment [43].

In addition, the use of this template provided a functional partition between subsystems and allowed each subsystem to be developed/tested in isolation from the other subsystems on the chip.

4.4 Verification Strategy

Hierarchical verification was the primary strategy employed during the development of Ballast. Each subsystem was independently verified as a separate entity using the most appropriate verification methodologies. Once verified as a standalone, the subsystem was then integrated into the top-level design and integration testing was performed [43].

The verification methodologies applied at the subsystem level included basic functional

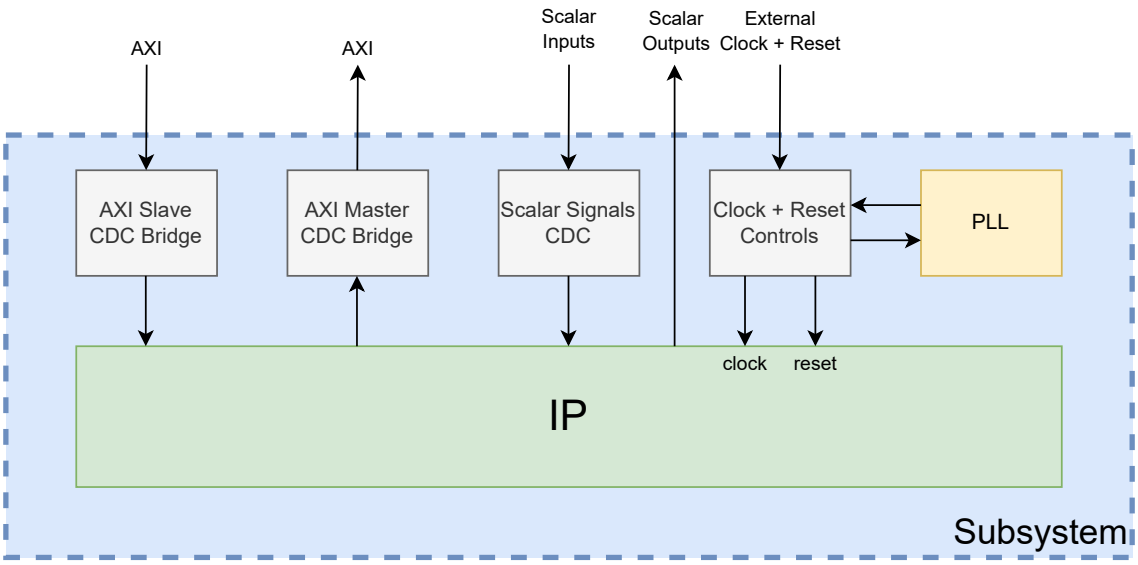

Figure 4.2. *Ballast Subsystem Template*

testbenches, UVM, HW/SW co-simulation and FPGA prototyping. This work focuses on FPGA prototyping; therefore, only this will be covered in detail.

4.4.1 Functional Coverage

When performing functional verification of a design, one must have access to a metric which has a direct relationship to the functionality of the design to determine at which point the design has been sufficiently tested. To define such a metric, a fault model of the design is created that is described at the functional level and is independent of the implementation details [37]. A fault analysis was performed during the planning stages of the Ballast SoC and was used as an input during the generation of the verification planning artefacts. The details of the verification plan creation and its contents outside of the FPGA prototyping testing objectives are beyond the scope of this thesis.

4.4.2 FPGA Prototyping

The Ballast FPGA prototyping objectives were dictated by the Ballast verification planning activities as follows:

1. Create an extensible and re-usable build flow which can be applied across the FPGA prototyping configurations required by Ballast,
2. Validate the SoC boot design,
3. Validate the SoC debugging architecture,
4. Validate SoC peripheral functionality through the use of real components,
5. Provide a platform for development of the SoC BSP and tools used for the SoC wake-up

activities.

6. Validate the functionality of the asynchronous C2C interface.

4.5 Subsystems of Interest

Ideally, creating an FPGA prototype of the entire Ballast SoC would be possible. However, this is not feasible due to hardware resource constraints on modern FPGAs. Different approaches can be taken to partition the SoC into smaller modules, which can then be individually prototyped on separate platforms. For example, [46] outlines an approach that uses the resources required to implement each module of the SoC on an FPGA and implements an algorithm that partitions the design into sections that can be implemented on a pre-defined FPGA platform. Due to project constraints, a more conservative approach was taken in the design partitioning of Ballast. Rather than aiming to prototype the entire SoC, it was found that to meet the FPGA prototyping objectives, only a subset of the subsystems and functionality of Ballast needed to be implemented on the same FPGA. Each subset (a prototype configuration) is described in more detail in Section 5.1.

As the selected prototype configurations only contain a fraction of the total subsystems on Ballast, only the subsystems relevant to the Ballast SoC prototyping configurations are described in detail here. It is possible that for some subsystems, FPGA prototyping was completed when developing the subsystem IPs. However, a prototype was not created as a part of the Ballast SoC development and integration activities. For example, the DSP subsystem was an IP developed independently of the SoCHub project and subsequently integrated into the Ballast SoC. It is known that FPGA prototyping of this subsystem was performed, but this was independent of the development of the Ballast SoC and is therefore not covered within this thesis.

4.5.1 SysCtrl

The System Control (SysCtrl) subsystem is adapted from the Pulpissimo SoC [45], initially developed by the University of Bologna. The SoC contains a collection of peripherals which are standard in contemporary microcontrollers (e.g. UART, SPI, timers, etc.), accessed using a low-power uDMA [42]. SysCtrl utilises the zero-RISCY variant of the RISC-V core, which is a minimal, 32-bit, 2-stage, in-order core; it implements the 'I', 'M' and 'C' extensions of the RISC-V ISA [53][17].

The primary purposes of the SysCtrl subsystem are to manage the boot sequence of the Ballast SoC, to control the clock/reset inputs to the other subsystems and to configure the SoC. Full details of the boot control can be found within [39]. The subsystem is not designed to be computationally powerful and, therefore, is not a main target for running application code. However, it contains several peripherals, which make it suitable to per-

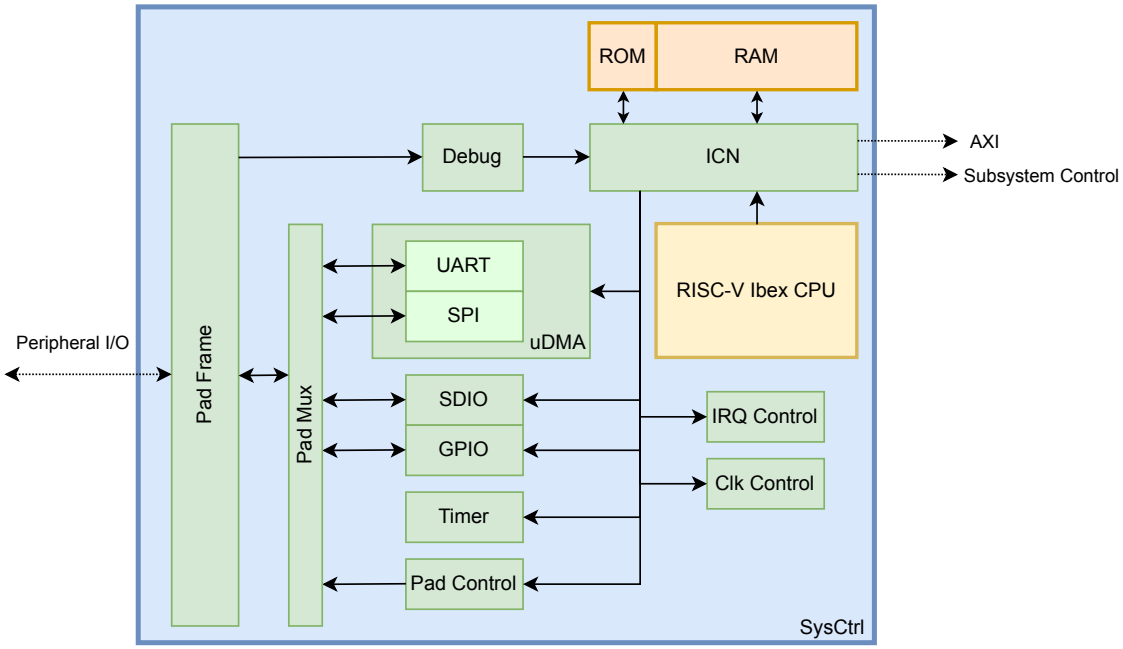

Figure 4.3. *High-level View of SysCtrl Architecture*

form supporting tasks such as acting as a system monitor for waking up from a low power state.

The following modifications were made to the Pulpissimo IP to create the SysCtrl subsystem:

- To enable a resilient boot process, the SDIO peripheral was removed from the uDMA subsystem and refactored to be accessed via the main system bus.
- Removal of unnecessary peripherals such as I2C, I2S and the parallel camera interface.
- Removal of the internal FLL and associated internal clock generators, as the clock signals of SysCtrl are generated by an external reference clock.
- Addition of external interrupt sources to enable IPC.
- Modifications to memory layout and reduction of ROM/RAM memory size.

A high-level view of the SysCtrl subsystem architecture can be seen in Figure 4.3.

4.5.2 MPC

Similar to SysCtrl, the MPC subsystem of Ballast is also created from a modified version of the Pulpissimo SoC. However, rather than using the low-power zero-RISCY variant, MPC utilises the RISCY core. The RISCY core is a 32-bit, in-order, 4-stage RISC-V core with the 'I' and 'M' extensions implemented. It has been further extended to increase performance and can be used within DSP and edge applications.

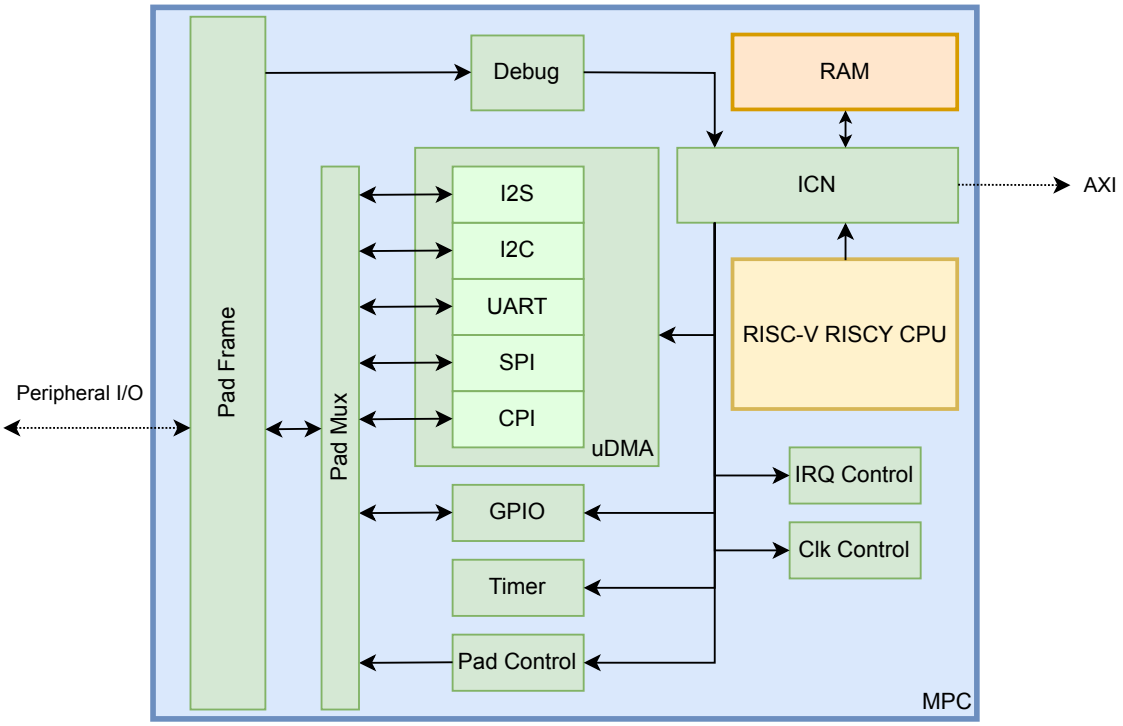

Figure 4.4. *High-level View of MPC Architecture*

The following modifications were made to the Pulpissimo IP to create the MPC subsystem:

- Removal of unnecessary peripherals such as SDIO as this is provided by SysCtrl.
- Removal of the internal FLL and associated internal clock generators, as the clock signals of SysCtrl are generated by an external reference clock.
- Addition of external interrupt sources to enable IPC.
- Removal of ROM.

A high-level view of the MPC subsystem architecture can be seen in Figure 4.4.

4.5.3 HPC

The HPC subsystem is designed for to handle computationally intensive applications on Ballast. It is implemented using two CVA6 RISC-V cores. The CVA6 is a 64-bit, single-issue, in-order RISC-V core which implements the 'I', 'M', 'A' and 'C' extensions. Additionally, it has Machine, User and Supervisor privilege levels implemented [54]. Furthermore, the CVA6 provides complete hardware support for MMU translation. These features enable the CVA6 to operate as an application class core and provide the infrastructure required to run popular operating systems such as Linux [58].

As well as two CVA6 cores, the HPC subsystem implements a 32kB 8-way L1 cache per core and a 256kB 8-way L2 cache, which is coherently shared between the cores.

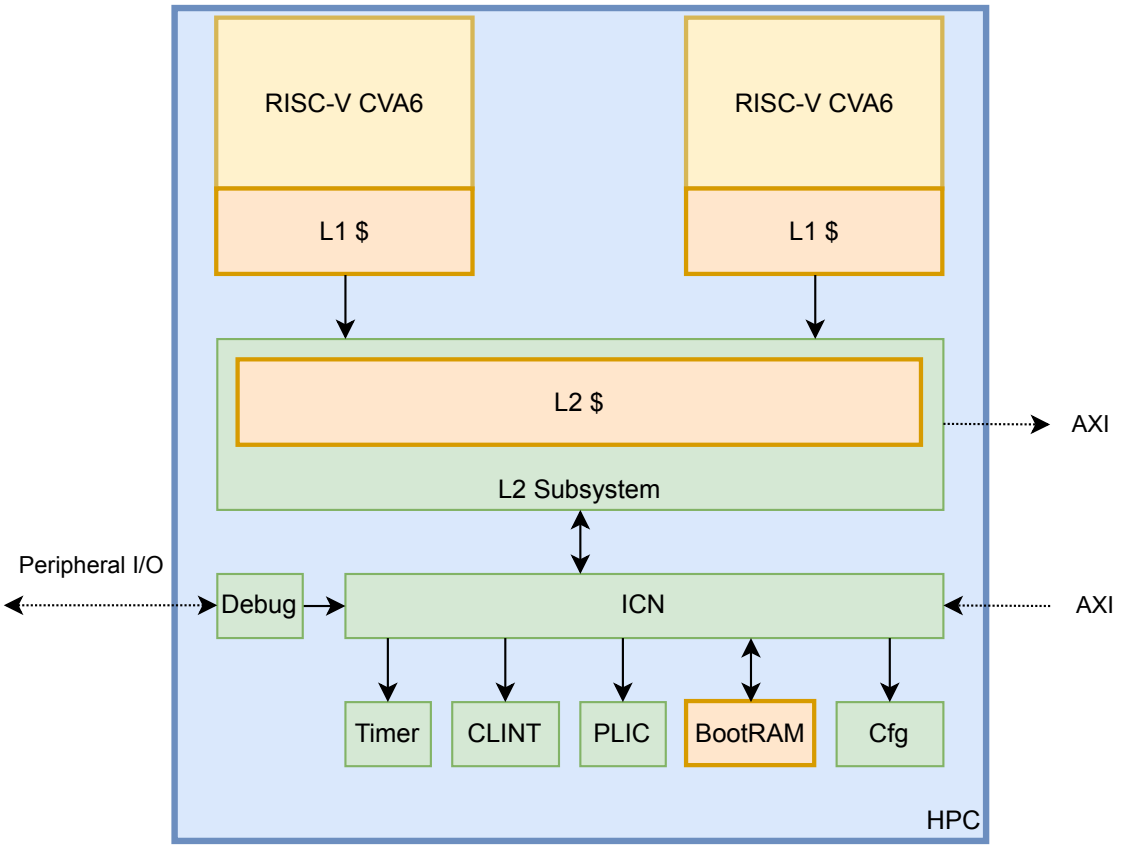

Figure 4.5. *High-level View of HPC Architecture*

Moreover, HPC includes an interrupt controller and timer peripherals.

A high-level view of the HPC subsystem architecture can be seen in Figure 4.5.

4.5.4 C2C

The C2C subsystem allows Ballast to be functionally extended by exposing a bidirectional physical interface. The interface implementation performs data transfer at a higher frequency than conventional serial communications protocols (e.g. SPI or I2C) without significantly increasing the pin count of the chip as would happen when using a parallel interface. The C2C interface enables efficient communications between Ballast and off-chip memories, sensors or another instance of Ballast.

The subsystem is connected to the Ballast interconnect via an AXI interface. It takes AXI transactions from the interconnect, converts them to an asynchronous serial protocol and then transmits them off-chip. Similarly, it receives serial data asynchronously over the external interface and converts it to an AXI transaction, which is then sent over the Ballast interconnect. Data synchronisation between the external interface and the interconnect is performed using asynchronous FIFOs and handshaking over the physical interface. With a clock frequency of 200MHz, a uni-directional bandwidth of 3.2Gbps can be achieved

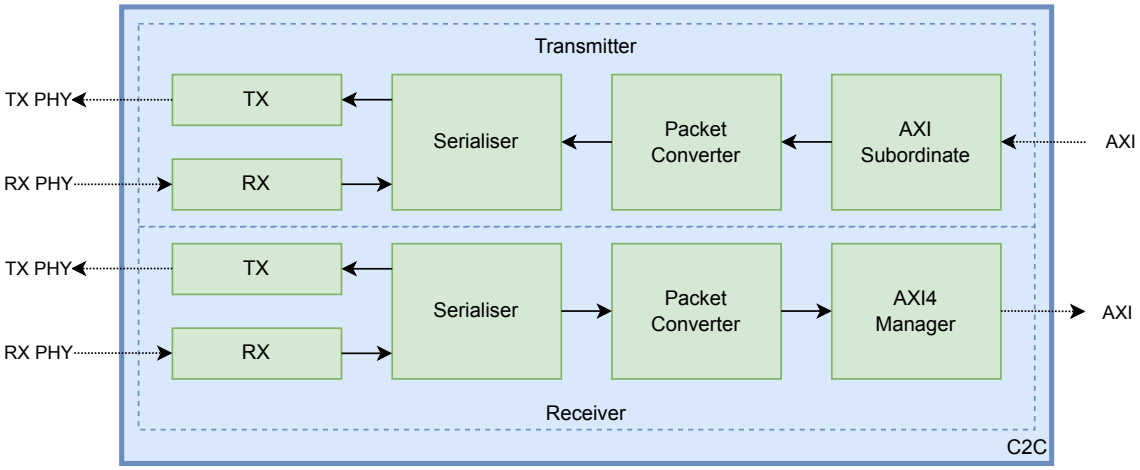

Figure 4.6. *High-level View of C2C Architecture*

[43]. A high-level view of the HPC subsystem architecture can be seen in Figure 4.6.

4.6 Debugging Architecture

4.6.1 JTAG

As integrated circuits became more complex and SMD components became more prevalent in solutions, it became more challenging for engineers to test the ICs by directly probing pins: JTAG was developed in 1985 as a solution to this problem. JTAG defines that a small number of pins and testing infrastructure are to be added to the design of each IC in a solution (see Figure 4.7. The pins of each IC can then be connected and presented to the user as a single JTAG port [14].

The JTAG standard defines a protocol which can be used to control the testing infrastructure (JTAG TAP) on each IC. The protocol controls a state machine that can store/load data to/from the IC [49]. With a few basic instructions, it is possible to control/monitor the IC functionality or perform boundary scan tests to verify the integrity of the circuits implemented on the chip. Additionally, the scope of how JTAG can be used has expanded to include more advanced functions, such as programming FPGA bitstreams into devices and integrating with external logic analysers to present the internal operations of an IC onto a display [14].

Alternatives to JTAG have been developed such as cJTAG and SWD with the goal of reducing the pin-count and area of the testing/debugging infrastructure.

4.6.2 Debug and Trace

With the complexity of modern SoCs, it is of utmost importance that mechanisms are provided to allow engineers to understand software behaviour and quickly identify software

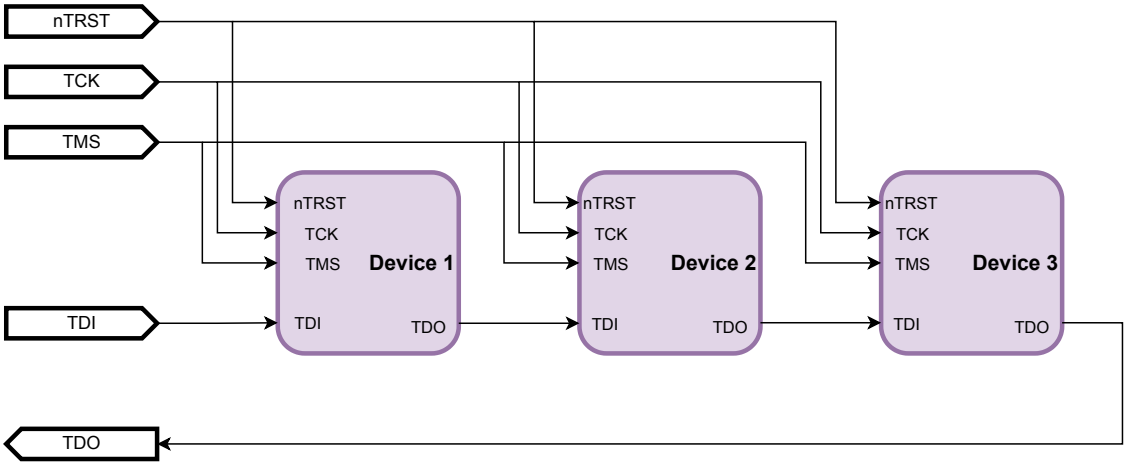

Figure 4.7. *Example of Daisy-Chained JTAG Bus for Test/Debug of Three Devices*

issues. Exactly how this is achieved can vary from chip to chip. However, the implementations generally aim to provide either debug capabilities, trace capabilities or some combination of both.

Debug capabilities typically include:

- Controlling execution of a core to enable the pausing of execution and stepping through the instruction execution.
- Accessing core registers.
- Performing reads/writes either directly from the debug infrastructure or using a selected core to perform the operation.
- Setting/clearing watchpoints/breakpoints within the code such that system events can be generated when particular conditions are met.

When debugging some code of interest on an SoC, manipulating the execution state can cause undesired secondary effects in the system's state. Therefore, it can become difficult to replicate an issue using debug capabilities alone. Trace capabilities complement this by allowing the user to stream configurable data or events to a dedicated memory or IO port. This data can be extracted and reviewed externally to allow the user to analyse the core or system behaviour without interfering with the execution [23].

4.6.3 OpenOCD

OpenOCD is an open-source software project which can be used to debug an application on a remote device. OpenOCD runs as a server and can be targeted by the GDB and can then communicate to a hardware debugging interface to send commands to a remote chip using a protocol such as JTAG or SWD. It supports the debugging of multiple platforms and can be configured to allow the debugging of multiple cores within a heterogeneous

Table 4.1. *Debug modules contained within Ballast and their compliance with the RISC-V Debug Specification.*

Subsystem	Debug Module	Specification Version
SysCtrl	PULP Debug Module	Not Compliant
	RISC-V Debug Module	v0.13.1
MPC	PULP Debug Module	Not Compliant
	RISC-V Debug Module	v0.13.1
HPC	RISC-V Debug Module	v0.13.1

SoC [41].

4.6.4 Ballast Debugging

The Ballast SoC is designed with the capability to debug the subsystems containing CPU cores and use these cores to debug the remaining systems on the chip indirectly. No trace capability is included to minimise the design complexity. Each of the HPC, MPC and SysCtrl subsystems is implemented with a RISC-V debug module and JTAG TAP. In addition, the SysCtrl and MPC subsystem also contain a custom implementation of a debug module, which was used by the PULP project for internal testing.

The RISC-V debug module specification has not yet been formally ratified and is in a draft state. As a result, the specification is updated frequently. Each of the HPC, MPC and SysCtrl IPs were initially forked from separate existing project repositories. The repositories for each subsystem used implementations of the RISC-V debug module, which are aligned to version v0.13.1 of the draft specification. The versions used by each subsystem can be found in Table 4.1.

Version 0.13.1 of the RISC-V Debug Specification the RISC-V module can be found in [20]. A sample of the most relevant supported features are:

- RISC-V hart registers can be read/written.
- Ability to debug all harts in the hardware platform.
- Each hart can be debugged from first instruction execution.
- Memory access from the system bus or hart.
- Breakpoint support.
- Hardware single-step can execute one instruction at a time.

The Ballast SoC has a single set of JTAG ports on the top level. As is typical in modern SoC architectures, these ports for the JTAG bus and the JTAG TAP of each subsystem with debug capabilities are connected to this bus, as can be seen in Figure 4.8.

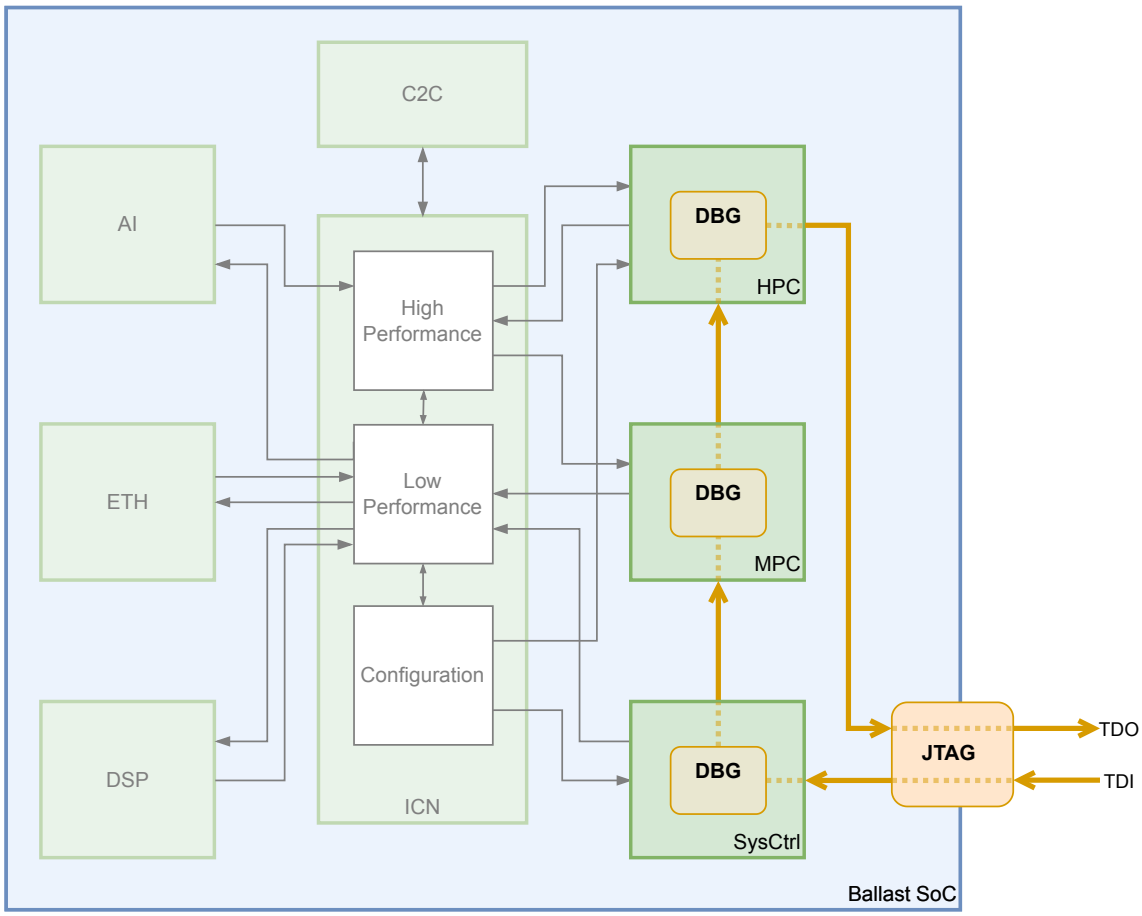

Figure 4.8. *Ballast JTAG Bus Connections*

5. FPGA PROTOTYPE IMPLEMENTATION

5.1 Prototype Configurations

The objectives for the Ballast FPGA prototyping were outlined within Section 4.4.2. The list below re-states the objectives as requirements and describes the FPGA prototype configurations that were suitable to satisfy them.

1. Validate the SoC boot design:

   Ballast boot is controlled by the SysCtrl subsystem. It utilises the JTAG, SDIO and QSPI peripherals. An FPGA prototype containing at least the SysCtrl subsystem with those interfaces exposed externally is required to verify the boot design.

2. Validate the debugging architecture:

   As Section 4.6.4 mentions, the Ballast debugging infrastructure is connected to the SysCtrl, MPC and HPC subsystems. An FPGA prototype containing these subsystems and the debugging infrastructure is required to test the chip debugging capabilities.

3. Validate peripheral functionality through the use of real components:

   Both SysCtrl and MPC are the designated subsystems to perform operations with peripherals. An FPGA prototype of at least the SysCtrl and MPC subsystems (together or standalone) with exposed peripheral IO pins is required to achieve this objective.

4. Provide a platform for development of the SoC BSP and tools used for the SoC wake-up activities:

   The main focus of the software development plan for Ballast was using the RISC-V cores to develop a Rust-based HAL. Therefore, FPGA prototypes of the HPC, MPC and SysCtrl subsystems were a priority for this. These could be in a standalone or combined platform.

5. Validate the functionality of the asynchronous C2C interface:

   As mentioned in Section 4.5.4, the C2C subsystem is designed to be used as an asynchronous interface. Therefore, to fully validate this aspect of the design, a prototype configuration of two connected C2C subsystems would need to be created. Each subsystem would use a clock which is asynchronous to each other.

The image in Figure 5.1 illustrates how the Ballast architecture was partitioned for prototyping. Each partition is labelled with the ID of the requirement it aimed to satisfy. Some requirements are partially satisfied by multiple partitions. The remaining subsystems were not included within the scope of SoCHub FPGA prototyping due to them already being prototyped independently of the SoCHub project.

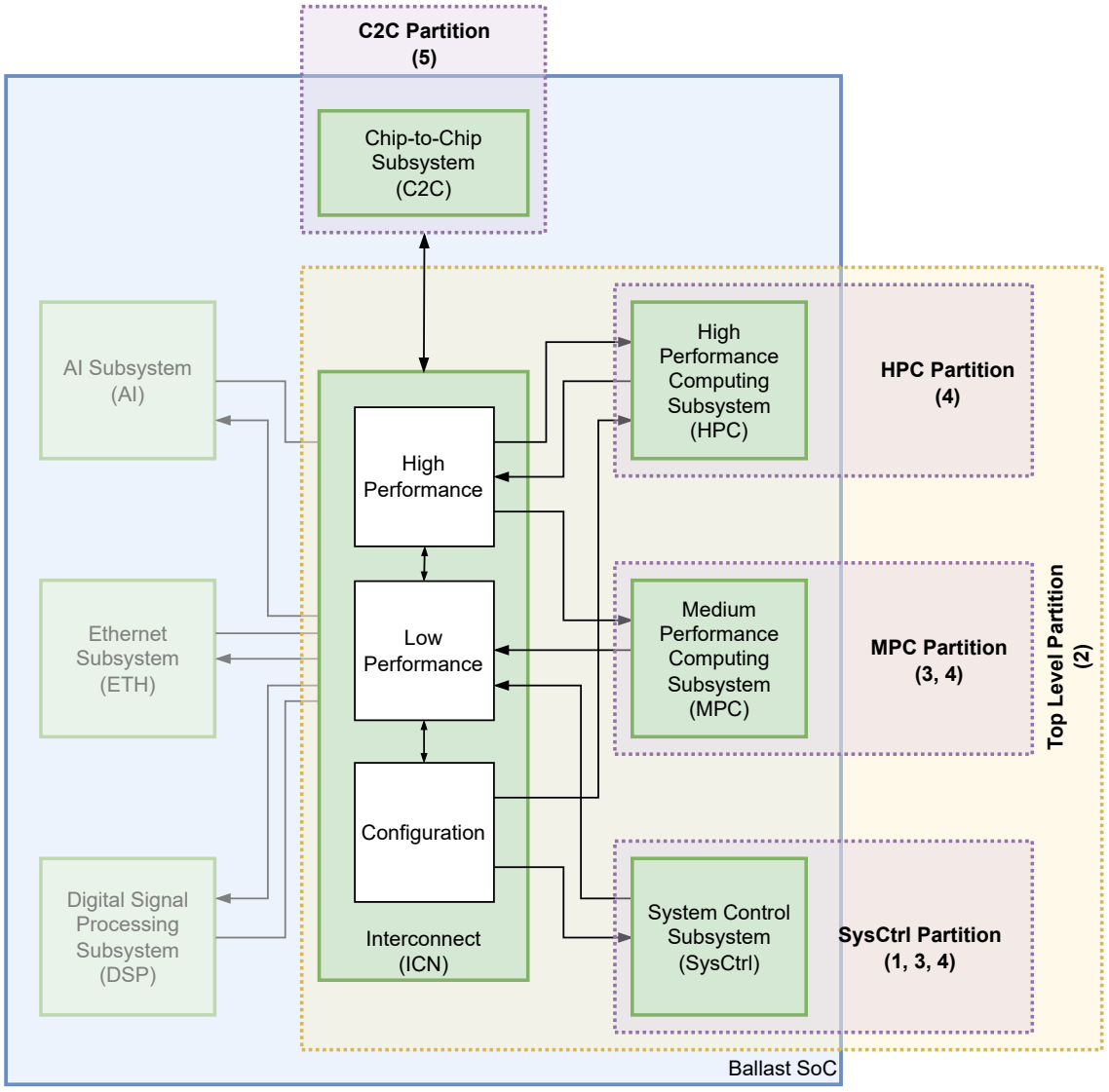

Figure 5.1. *Partitioning of Ballast Architecture for Prototyping*

5.2 Platform Hardware Selection

The primary characteristics that were considered when selecting a platform for prototyping were the size of the FPGA fabric, functional features offered by the FPGA and number of available IO pins. This led to the selection of three platforms which are described in the following sections.

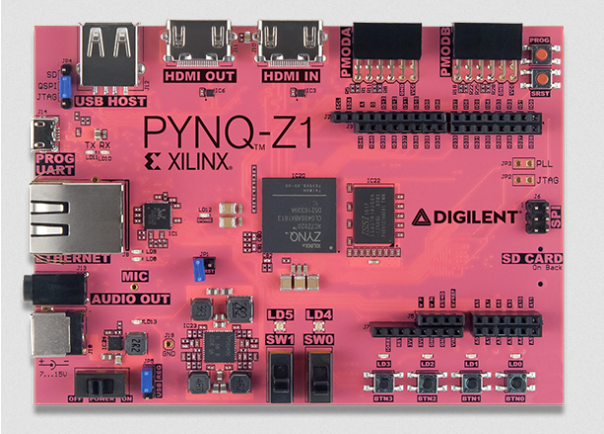

Figure 5.2. *Digilent Pynq-Z1 Development Board [19]*

5.2.1 Digilent PYNQ-Z1

The PYNQ-Z1 (Figure 5.2) is an FPGA development board supplied by Digilent, which is designed to be used with the PYNQ project, an open-source project from AMD, to make it easier to develop adaptive computing platforms by using Python to develop programmable logic for selected AMD platforms [1].

The board itself houses an AMD Zynq (XC7Z020-1CLG400C) SoC, which contains a dual-core Arm Cortex-A9 processor, a DDR3 memory controller, high-bandwidth peripheral controllers, low-bandwidth peripheral controllers, with Artix-7 programmable logic having the resources shown in Table 5.1[19]. The PYNQ-Z1 board exposes 49 external IOs, making it an appropriate choice for implementing smaller designs with many external peripherals.

5.2.2 AMD Zynq UltraScale+ MPSoC ZCU104

The ZCU104 (Figure 5.3) is an evaluation kit supplied by AMD-Xilinx targeted for use in embedded vision applications. It houses a Zynq Ultrascale+ (XCZU7EV-2FFVC1156) MPSoC, which contains a quad-core Arm Cortex-A53 APU, a dual-core Cortex-R5 real-time processor, GPU, video codec, DDR4 memory controller, a rich set of both high-bandwidth peripheral controllers and low-bandwidth peripheral controllers, with Ultrascale programmable logic containing the resources shown in Table 5.1[3]. The ZCU104 board exposes three PMOD interfaces and an LPC FMC socket, making it an appropriate choice to implement medium-sized designs with a small number of external peripherals.

5.2.3 AMD Virtex UltraScale+ FPGA VCU118

The VCU118 (Figure 5.4) is an evaluation kit supplied by AMD-Xilinx targeted for high-performance FPGA designs. It houses a VCU118 XCVU9P-L2FLGA2104E FPGA, which

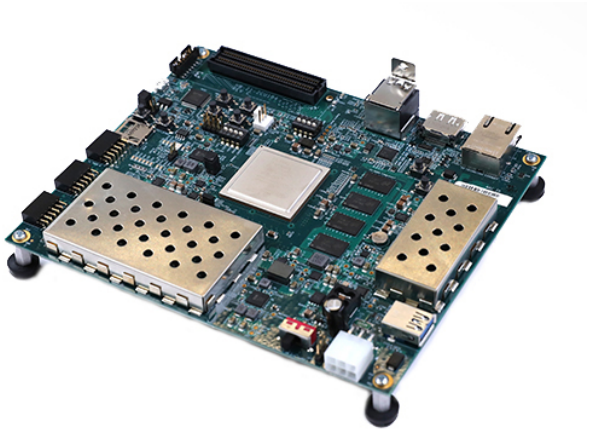

Figure 5.3. *AMD-Xilinx ZCU104 Development Board [3]*

Figure 5.4. *AMD-Xilinx VCU118 Development Board [2]*

contains a DDR4 memory controller, a selection of high-bandwidth peripheral controllers, with a Virtex Ultrascale programmable logic having resources shown within 5.1[2]. The VCU118 board exposes 2 PMOD connectors and 2 HPC FMC sockets, making it an appropriate choice for large, high-performance designs with few external peripherals.

Table 5.1. *Comparison of Programmable Logic Resources In Selected Platforms*

Resources	PYNQ-Z1 [19]	ZCU104 [3]	VCU118 [2]
Logic Cells	13.3k	504.0k	2,586.0k
Memory (Mb)	5.04	38.0	345.9
DSP Slices	220	1,728	6,840

5.2.4 Prototype Configuration to Platform Mapping

Using the platform information and configuration requirements detailed within the previous sections, Table 5.2 indicates a suitable mapping between the prototype configurations and the selected hardware platforms.

Table 5.2. *Mapping of Prototype Configuration Against the Most Suitable Hardware Platforms*

Configuration	Design Size	IO Count	Platform
SysCtrl	Small	High	PYNQ-Z1
MPC	Small	High	PYNQ-Z1
HPC	Medium	Low	ZCU104/VCU118
C2C	Medium	Low	ZCU104/VCU118
Top Level	Large	Low	VCU118

5.3 Prototype Build Flow Development

5.3.1 GNU Make

GNU Make is an easy to use, but powerful build tool which allows Users to define commands (referred to as Make targets), which can then be used to perform complex command sequences and generate output artefacts [32]. It is traditionally used to construct C/C++ software projects but can also be used in hardware to drive tools.

5.3.2 TCL

TCL is a general-purpose interpreted scripting language which is commonly used to drive the operation of modern IC development tools. It was developed in the late 1980s to attempt to unify the various command line tools used to drive the various IC development tools that existed at the time. It was designed to be easily extensible so that on top of the base language, each vendor could extend and add custom functions which could be used to drive their tool efficiently [40].

5.3.3 Vivado IDE

As AMD-Xilinx FPGA platforms were selected for prototyping, the AMD-Xilinx Vivado IDE was used to develop the prototypes. The Vivado IDE can be operated using a GUI or driven using TCL [11]. Version 2019.2 was chosen to maintain compatibility with the example FPGA flows by the open-source IP used within Ballast.

5.3.4 Synthesis Flow

As described in Section 3.4.2, the synthesis flow for FPGA designs contains multiple artefacts and stages. There was a desire to develop a flow for the Ballast prototyping aligned with the ASIC RTL simulation flow. An aligned flow would ensure that users

who are familiar with the existing RTL simulation flow, but might be unfamiliar with FPGA development would be able to quickly generate a bitstream for running tests on FPGA without any need to modify their environment or use unfamiliar tools.

The entry point for building and simulating the Ballast RTL was a top-level Makefile containing multiple targets. The general approach to align the FPGA build process with the existing RTL simulation process was to create a top-level Makefile within a sub-directory dedicated to the FPGA build. The Makefile contained targets to fully build a prototype configuration, build selected IPs for a specific prototype configuration, or clean the environment of all files generated during the build process. Once the main build target is called, the build completes the FPGA build efficiently and returns to the entry point. The specific prototype configuration and target platform are selected through variables assigned when calling the target (FPGA_CONF and FPGA_BRD, respectively). The static configuration data for each FPGA prototype is stored within separate files, which are included depending on the value of the FPGA_CONF and FPGA_BRD variables. Listing 5.1 shows an example of how the build process for the HPC prototype configuration on the ZCU104 platform could be initiated:

Listing 5.1. *Example call of FPGA build*

```
make all FPGA_CONF=HPC FPGA_BRD=ZCU104
```

An abstract illustration of how the FPGA build flow operates can be seen in Figure 5.5.

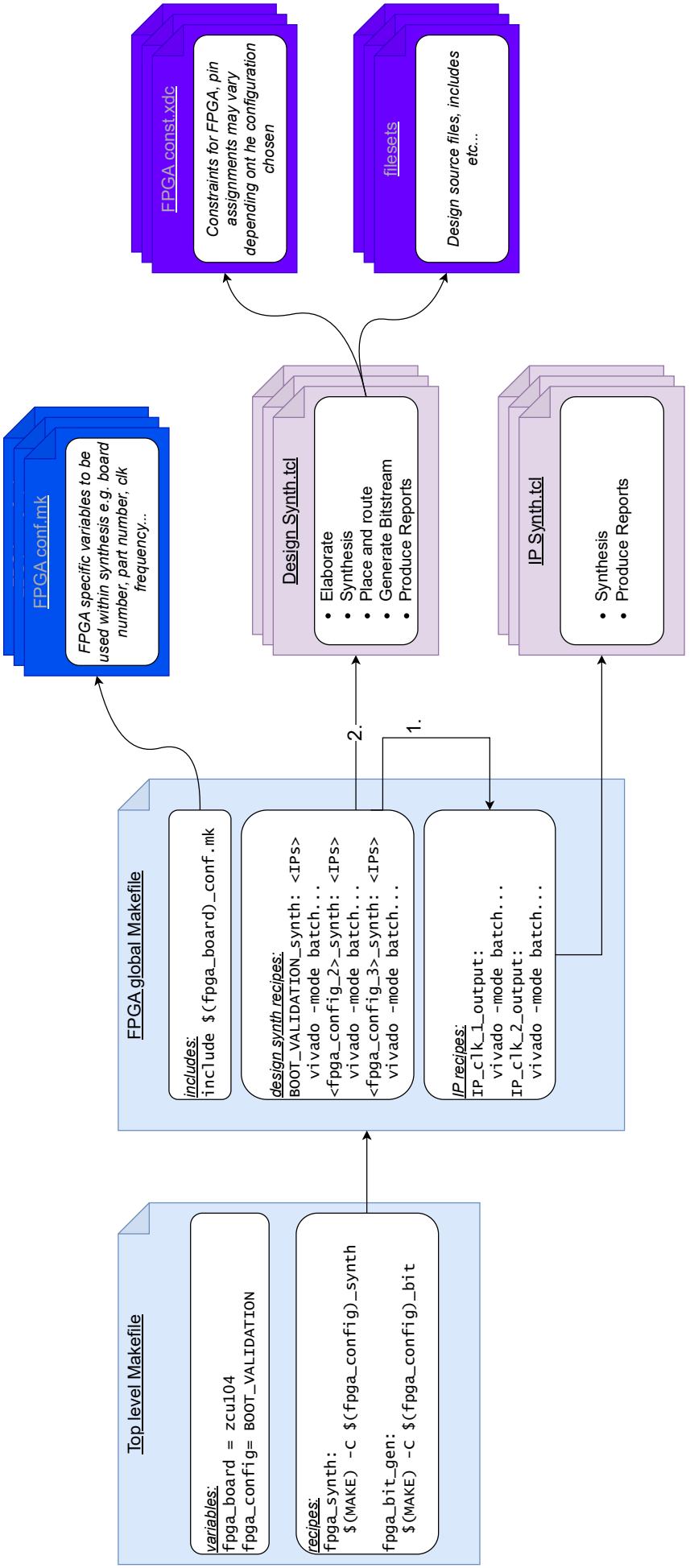

Figure 5.5. *FPGA Build Flow used for SoCHub Prototyping*

5.4 General FPGA Prototype Implementation Strategies

Modifications to the ASIC design were required to implement the FPGA prototype configurations. As described within [24], the focus of this exercise was not to completely rewrite the design as this would contradict the aim of using the same sources for ASIC and FPGA prototype, but to optimise critical design parts either for performance or resource utilisation. The scope of this section is limited to prototype implementation modifications applied to all of the prototyping configurations in Ballast. Subsystem-specific considerations can be found in Sections 5.5 to 5.8.

5.4.1 RTL Partitioning

When prototyping a portion of a complex SoC, it is necessary to ensure a suitable partition is drawn between the parts of the design required for functional testing of the target design and the rest of the SoC design. The process of functionally partitioning the SoC into prototyping configurations is outlined in Section 4.5. This section describes the common low-level approach to partition each prototyping configuration from the rest of the SoC.

Each subsystem of Ballast was initially developed as a standalone IP. The IP was then integrated into a wrapper component, which contained components and interfaces common to each subsystem on the SoC. This wrapper component was then instantiated on the SoC top level and connected to the other subsystems on the SoC. For the FPGA prototyping of the individual subsystems, the wrapper component is replaced with an FPGA wrapper equivalent, which removes the unwanted ports to the rest of the SoC and contains required FPGA-specific components (see Figure 5.6).

For example, the SysCtrl wrapper module port definition includes ports for the AXI4 interface, clock controls, PLL configuration and peripheral IO signals. The SysCtrl FPGA wrapper module removed the AXI4 interface, clock controls and PLL configuration. It should be noted that some of the external signals removed from the original wrapper are inputs to the SysCtrl IP. These signals were driven by constant values within the wrapper to prevent build issues or bugs in the FPGA design. In addition to the port modifications, the SysCtrl wrapper module contained CDC logic for the signals travelling between clock domains on the SoC. As described within Section 5.4.2, to simplify the prototyping design, each design was implemented within a single clock domain. Therefore, the CDC components within the wrapper were removed. Furthermore, to control the clock generation in the FPGA design, a clock management IP was instantiated in the FPGA wrapper, which replaced the PLL component contained within the original wrapper.

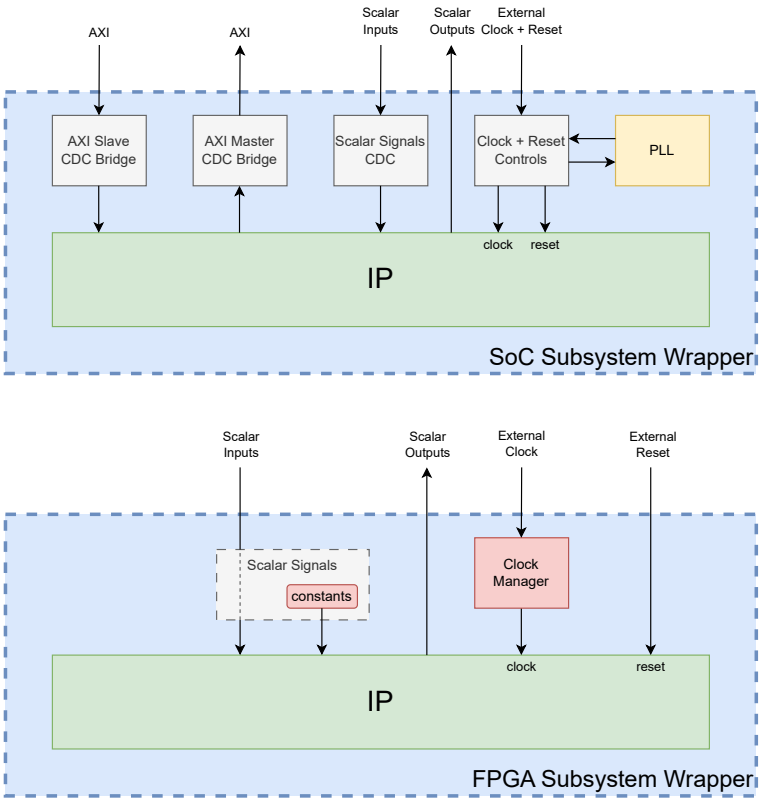

Figure 5.6. *Comparison of Top-Level Wrapper Used for ASIC and FPGA Prototyping*

5.4.2 Input Clock Architecture

The Ballast SoC was designed to allow most IPs to operate at configurable clock frequencies by including a PLL IP and clock selection logic components within the wrapper of each subsystem. It is impossible to synthesise the PLL IP used within each wrapper for FPGA, as it is implemented using mixed-signal components not present in the FPGA platforms. Additionally, as the FPGA platform uses different technologies to implement the logic compared to the ASIC, there is no way to gain useful timing information from the prototype. Therefore, a decision was made to make the clock tree in the prototype as simple as possible. As a result, clock selection logic was not included within the FPGA wrapper, leaving the design to be fed from a single clock.

While the clocking inputs to the design were simplified for prototyping, having control over the clock configuration is still valuable. As mentioned previously, the technology used to implement the design on FPGA significantly differs from the technology used to implement the ASIC design. As a result, achieving the targeted high clock frequencies of the ASIC design is often impossible. Therefore, clock management components need to be inserted into the FPGA design to reduce the frequency of the design. The AMD-Xilinx Clocking Wizard is an IP provided within Vivado, which enables the possibility of synthesising one or more clock signals with an output frequency independent of the IP's input clock frequency [4]. The frequency of the clock manager IP was defined within the

static configuration data described in Section 5.3.4 and could be changed at compile time if desired. Note that the IP can also perform dynamic clock configuration via an AXI-Lite slave interface; whilst this was not used for prototyping Ballast subsytems, it is clear that this could provide value when prototyping future designs.

5.4.3 Memory Interfaces

A common component found within modern SoCs is on-chip memory. Depending on the requirements and process selected to implement the ASIC, on-chip memory can be implemented using different technologies. The functionality of on-chip memory modules can also vary depending on the needs of the design. This difference in functionality is typically reflected in the ports on the interface to the memory module. For example, memory interfaces can use a single port or dual port configuration and contain additional control signals, such as a byte-enable signal, to mask the contents of the read memory.

As described in Section 3.3.3, FPGAs contain limited on-chip memory in the form of BRAMs. It is desirable to utilise FPGA BRAM to implement the ASIC on-chip memories within the FPGA prototype, as otherwise, the memories are implemented using registers (known as register RAM) or LUTs (known as distributed RAM). For larger memory sizes, register RAM or LUT RAM can become expensive in terms of logic resources and increase the complexity of the routing task performed by the FPGA design tools.

The memory configurations created using BRAMs on FPGA vary between FPGA platforms. Vivado offers the ability to either infer memories through specific RTL constructs (an example of a module which infers a single port memory can be seen in Listing A.1) or via instantiating BRAM IPs within the design. Each on-chip memory instance within Ballast included within the prototype was refactored to ensure that the RTL used to instantiate the memory was compatible with the pattern required by Vivado to infer the appropriate BRAM. Furthermore, due to the resource constraints of the FPGA platform, reducing the size of the memories used by the prototype configuration may be necessary. The specific modifications for each configuration are described in detail within the appropriate subsection below.

5.4.4 Clock Gating

To optimise the PPA of an ASIC design, various power reduction techniques are applied by ASIC designers, including clock gating, pipelining, parallel logic implementation and clock frequency reduction. Clock gating is the most widely used technique. It involves applying a mask to the clock signal, such that when the logic block is not in use, the clock signal (and therefore all logic transitions within the block) can be dynamically halted. Halting the clock prevents dynamic power consumption while the logic block is not in

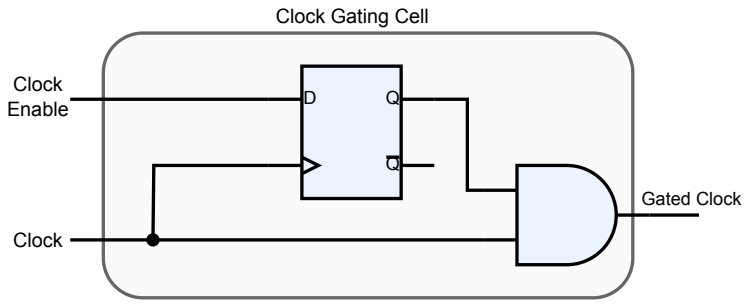

Figure 5.7. *Example of Clock Gate Component Used in ASIC Design*

use. Clock gates are typically implemented within a dedicated module, which contains a sequential element to control the enable, fed into an AND gate along with the clock (Figure 5.7) [14].

As described within Section 3.3.4, FPGAs contain dedicated routing networks for clock and logic distribution, which are configured during the programming of the device. These networks are distinct, with each clock network being optimised to minimise skew. The clock gating implementation used for ASICs would require the addition of arbitrary logic to the clock signals which is not achievable on FPGAs. As a result, the FPGA clock signal must be routed and fed into the logic network within which the combinatorial and sequential elements can be accessed. The logic output is then routed back into the clock network, from where it can feed the downstream sequential logic blocks. This re-routing of clock signals introduces a significant amount of skew onto the clock signal, and therefore, traditional implementations of clock gates are inefficient on AMD-Xilinx FPGA platforms [6]. As a result, work was required to review the use of clock gates throughout the components of Ballast, which were targeted for prototyping and to modify the design to be suitable for implementation on FPGA.

Reduction of power is not a relevant consideration in FPGA prototyping as the main focus is verifying the functionality of the design. A conservative approach was taken to restrict the application of clock gating to only those areas in which it was used to control functionality e.g. controlling external clock signals of synchronous peripherals. With this in mind, the two following logic transformations were applied to remove the timing issues introduced by clock gating:

1. If the clock gating has no functional implications (i.e., it is only for power saving), the clock gate module was replaced with an alternative implementation that removed the internal logic and created a direct connection between the input and output clock, essentially bypassing the clock gate entirely.
2. If the clock gate was used to control functionality within the design, the clock gate module was replaced with an alternative implementation which replaced the existing logic components with an instantiation of a gated clock buffer (BUFGCE). The BUFGCE

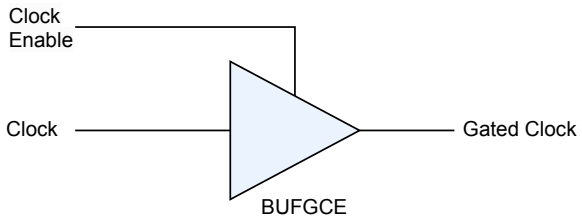

Figure 5.8. *BUFGCE Component Used to Replace Clock Gates in Prototypes [7]*

is an FPGA primitive located within the FPGA clock network (see Figure 5.8) but can be used to gate clock signals without introducing significant skew. There are a limited number of these primitives contained within the FPGA. Therefore, to ease the burden of placement and STA, these were only used when bypassing the clock gate would have functional ramifications.

5.4.5 IO Pads

The IO pads used in ASIC implementations are hard macros linked to the technology used. Within RTL simulation, these components are modelled using functional models which emulate the behaviour of the pad component. As described within Section 3.3.2, FPGAs typically contain a set of configurable IO components. The IO components available within the FPGA platforms chosen to prototype the Ballast SoC do not perfectly match the IO pad components used by the ASIC implementation. Therefore, additional logic was required to emulate as much of the IO pad functionality as possible without affecting the functional operation of the design. The result was a simplified IO pad that could be dynamically configured as an input or output. Additionally, the pad could statically be configured to apply a weak pull-up or pull-down resistor. The code used for the FPGA IO pad definition can be found in Listing A.4.

5.5 SysCtrl and MPC Specific FPGA Prototyping Implementation

A high-level architectural diagram of the SysCtrl and MPC prototype configurations can be seen in Figure 5.9 and Figure 5.10, respectively. The functional blocks which were modified for prototyping are highlighted with a hatched pattern.

5.5.1 SysCtrl BootROM

As mentioned in Section 4.5.1, one of the main functions supported by the SysCtrl subsystem is the boot management for the Ballast SoC. The bootROM for the subsystem was under development during the prototyping of the subsystem. A substitute bootROM module was created to simplify the initial testing of the emulated subsystem during this

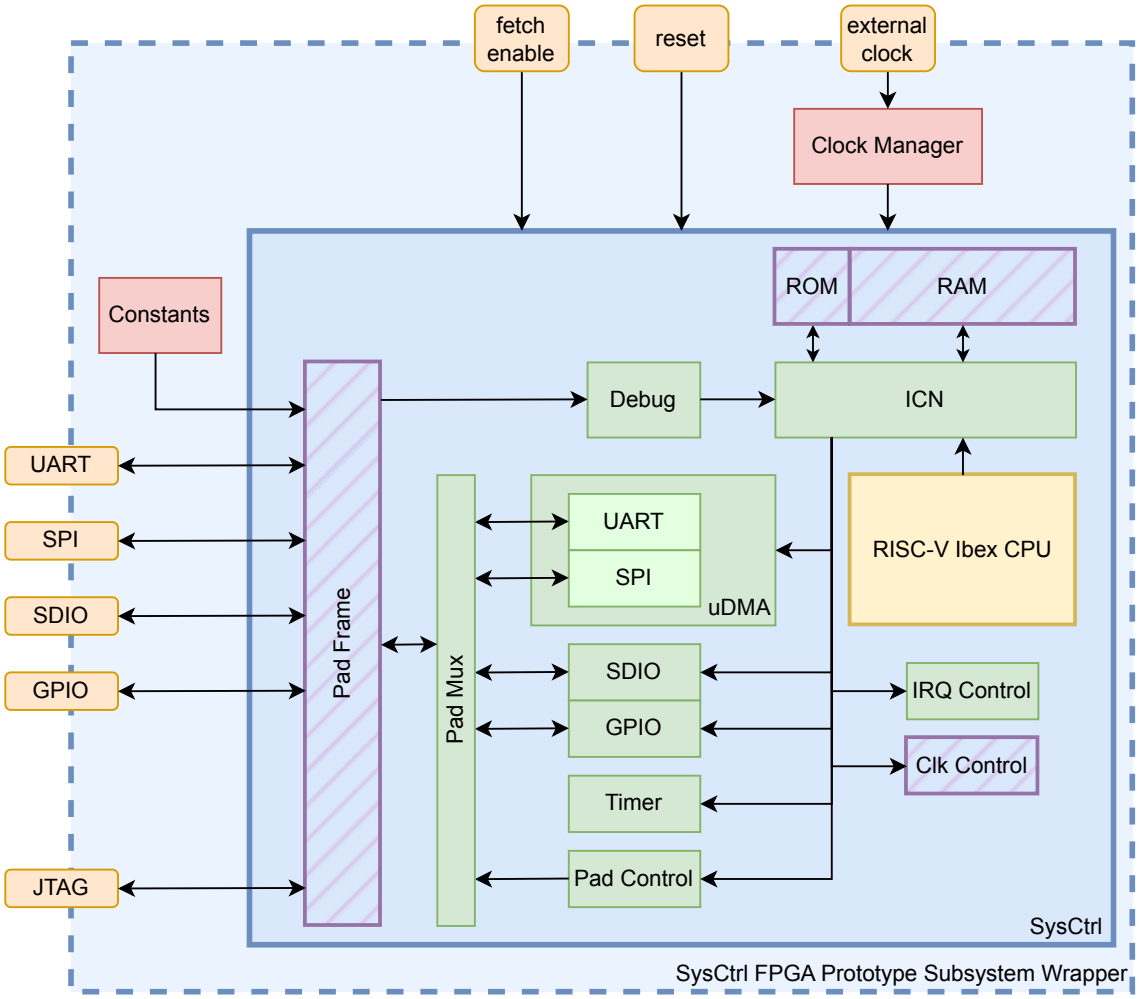

Figure 5.9. *High-level Architecture of the SysCtrl FPGA Prototype*

period. The simplified bootROM contains a single unconditional jump instruction with the offset set to zero. When the SysCtrl CPU executes from the modified bootROM, it will remain in a stable infinite loop. The simplified bootROM provides a helpful starting point for emulation from which the JTAG and debug interface could be tested. The code used for the simplified BootROM can be found within Listing A.2. This simplified bootROM was replaced with the final bootROM once it was mature.

5.5.2 SysCtrl SDIO Clock Gating

The SysCtrl boot relies on using the SDIO protocol to boot from an SD Card. The SDIO interface consists of several signal pins, namely clock (CLK), command (CMD) and four parallel data (DATA) pins [13]. The command and data signals are transmitted synchronously to the signal transmitted on the clock pin. To control the clock of the SDIO interface, the module developed for Ballast relies on clock gating. As described in Section 5.4.4, a modification to the RTL was required to transform the ASIC clock gate implementation such that it could be implemented on FPGA with the BUFGCE component. The code used to

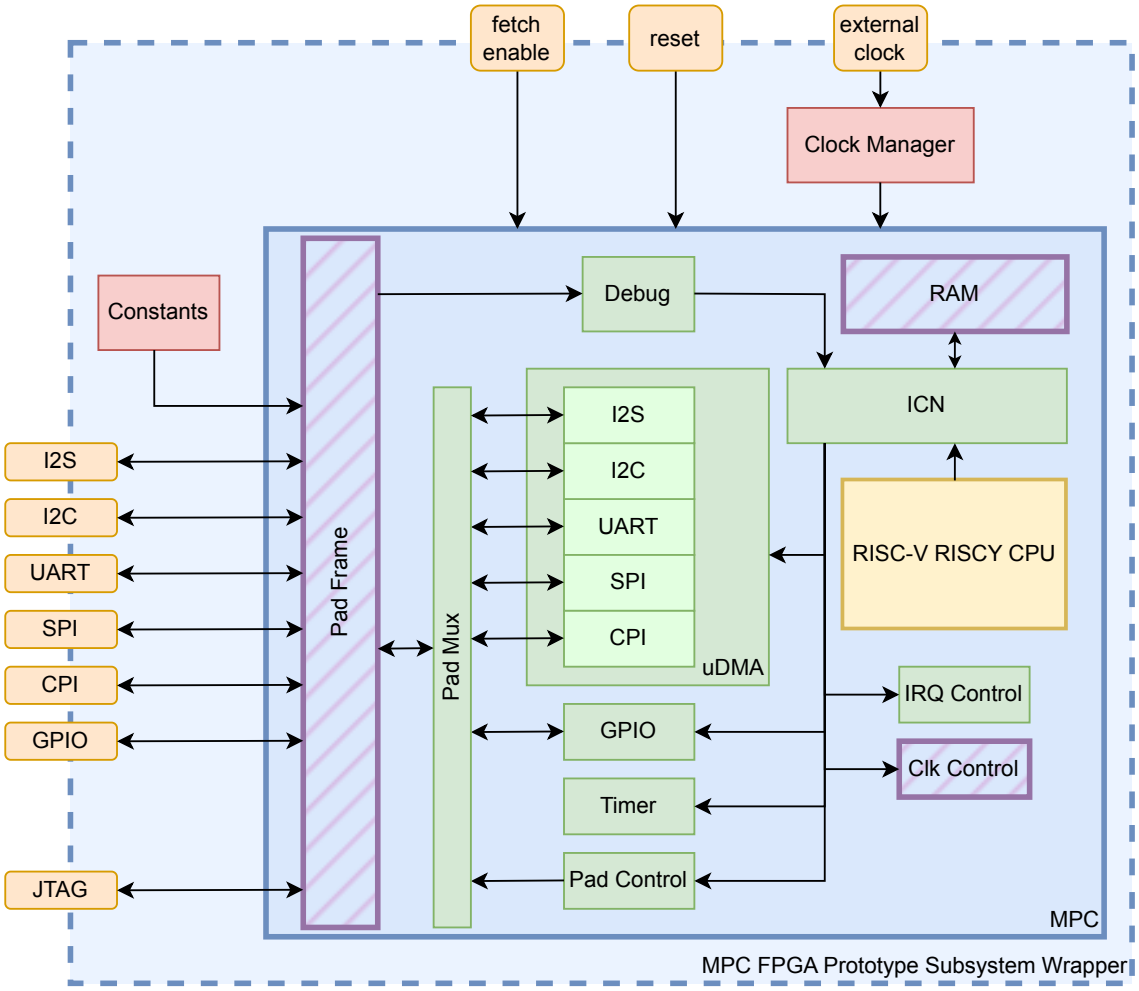

Figure 5.10. *High-level Architecture of the MPC FPGA Prototype*

implement the SysCtrl clock gating on FPGA can be found within Listing A.3.

5.5.3 Slow Clock Generator

Both SysCtrl and MPC subsystems contain timer modules that can generate interrupts at a defined rate. The timer modules require a reference clock at a set frequency of 32.768kHz. The input reference clock frequency for Ballast is fixed; therefore, implementing a divider to create the necessary frequency is trivial. However, FPGA development boards' input reference clock frequencies vary between boards. As a result, more work is required to create a generic solution that will allow the generation of the necessary 32.768kHz frequency.

The chosen solution combines an AMD-Xilinx Clock Wizard IP instance and a fixed clock divider (Figure 5.11). The input and output frequency of the Clock Wizard can be defined as variables within the TCL scripts used to build the FPGA prototype. The lower limit of the Clock Wizard output frequency is 4.9MHz; therefore, it is impossible to set the output frequency to the required 32.768kHz. As a result, the output of the Clock Wizard is set

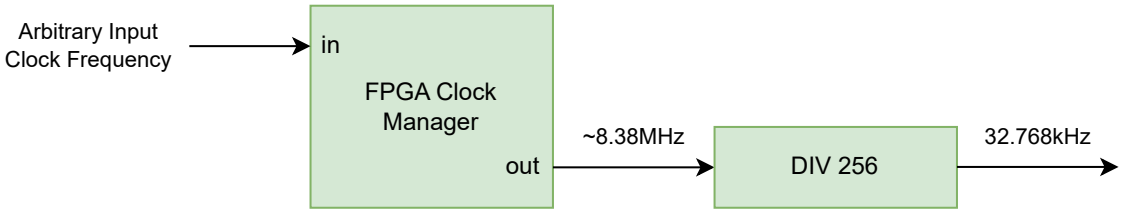

Figure 5.11. *FPGA Slow Clock*

to a value 256 times faster than 32.768kHz, and then it is divided by 256 using a clock divider implemented within the RTL immediately after. As the input frequency of the Clock Wizard can be set in the TCL scripts used to build the prototype, this design will guarantee correct frequency synthesis regardless of the input clock frequency.

5.5.4 Peripheral Clocks

The frequency of the clocks used by several of the peripherals on SysCtrl and MPC are configurable to allow them to operate at variable rates. The ASIC implementation achieves this through a combination of clock dividers and clock muxing components. Similar to the clock gating considerations stated in Section 5.4.4, efficient clock selection through multiplexers cannot be achieved in FPGA implementations without specific FPGA components. The clock mux components in the ASIC design were replaced with BUFGCTRL components accordingly.

Additionally, special consideration was given to the peripheral clock feeding the SDIO interface of SysCtrl, as the initial frequency value of the interface during SD Card initialisation needs to be set to a frequency between 0 - 400kHz [13]. The input clock frequency of SysCtrl on Ballast is 33.33Mhz, and the clock feeding the SDIO module is divided by 128 at boot, such that the SDIO clock initial frequency is 260kHz. As the input clock for the FPGA prototype was set at 10Mhz, this would result in the SDIO initial frequency being set to 78kHz. While this is within the acceptable range per the specification, experimentally, it was discovered that the SDIO initialisation was unstable when the SDIO clock frequency was below 100kHz. Therefore, the FPGA prototype was modified such that the clock feeding the SDIO module was only divided by 64, leading to an initial frequency of 156kHz. The modified clock tree contained within the SysCtrl FPGA prototype, with the default SDIO clock path highlighted, can be seen within Figure 5.12.

5.5.5 Memory Capacity

The ASIC implementation of SysCtrl contains 64kB of Private RAM in total, accessed across two 32kB blocks. Furthermore, MPC has the same 64kB of Private RAM but includes four interleaved blocks, which are 114kB each. By default, the memory of the

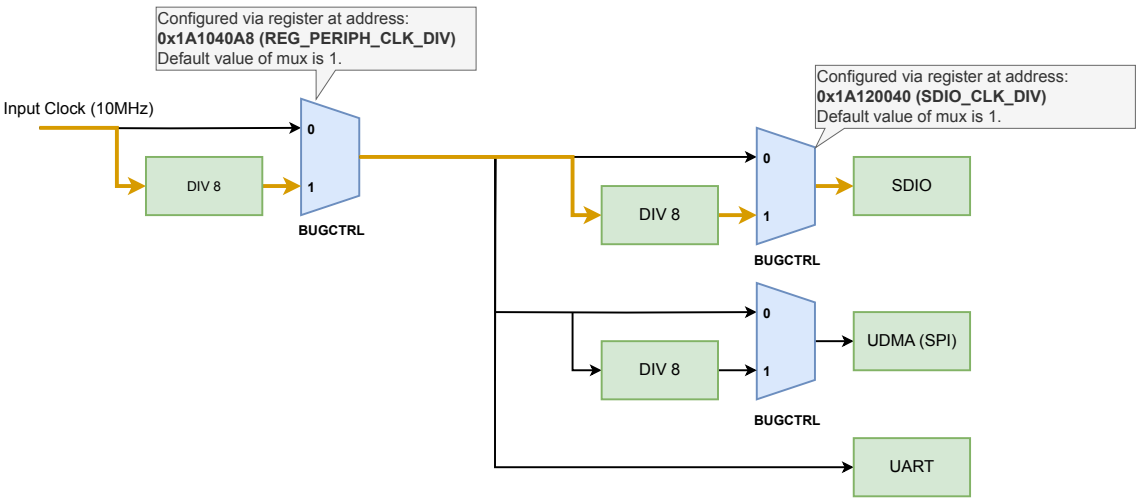

Figure 5.12. *FPGA SysCtrl Peripheral Clock Tree*

FPGA implementation for both SysCtrl and MPC is the same as the ASIC. Still, this can easily be modified by modifying variables within the build scripts if necessary.

5.6 HPC Specific FPGA Prototyping Implementation

A high-level architectural diagram of the HPC prototype configuration can be seen in Figure 5.13. The functional blocks which were modified for prototyping are highlighted with a hatched pattern. The removed (and bypassed L2 cache subsystem) is highlighted with a cross pattern.

5.6.1 L2 Cache Controller

As described in Section 4.5.3, the HPC subsystem contains a 256kB 8-way L2 shared cache. When prototyping HPC, it was found that some of the SystemVerilog structures used to implement the L2 cache could not be synthesised for FPGA within Vivado 2019.2. The L2 cache is a complex module, and project time constraints meant that finding a working fix for this issue would be a significant challenge. In addition, modifying the RTL to ensure that the L2 cache could be synthesised would create considerable differences between the ASIC design and the prototype. A review was performed to assess the benefits of fixing the issue, which concluded that it would not merit the time investment. As a result, the FPGA prototype of HPC does not contain a shared L2 cache and all memory accesses from the L1 cache of each core are performed directly upon the memory, bypassing the L2 cache.

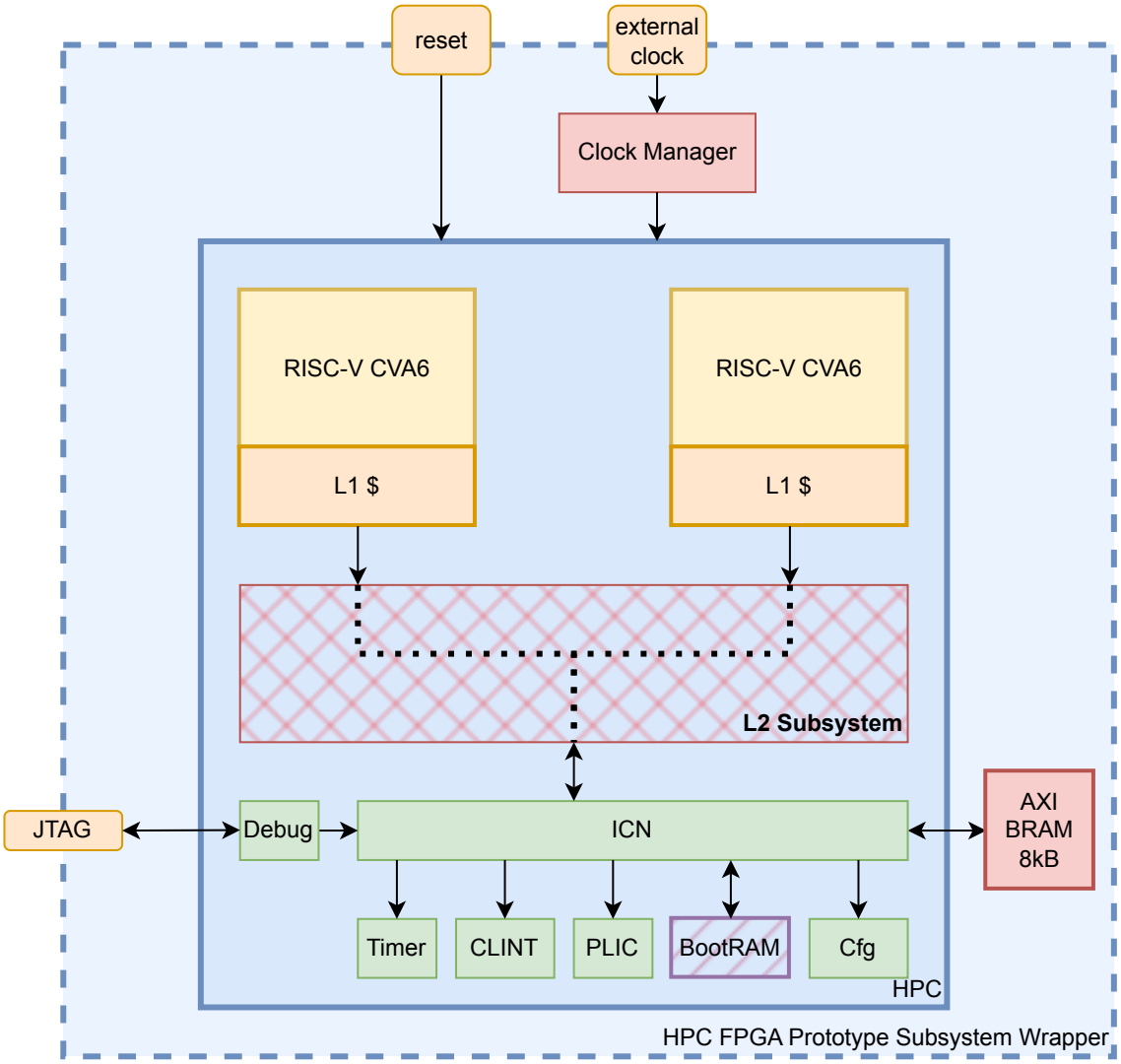

Figure 5.13. *High-level Architecture of the HPC FPGA Prototype*

5.6.2 Memory Capacity

The ASIC implementation of HPC contains 32kB internal memory and can access a 4GB external memory space. The testing of HPC on the FPGA prototype platform consisted of simple software operations. As a result, the external memory size was set to 8kB on the FPGA prototype to reduce the resources required by the prototype, thereby reducing the burden on the synthesis tools to route large memories or implement external memory controllers.

5.7 C2C Specific FPGA Prototyping Implementation

As described in Section 5.1, a requirement of the FPGA prototyping for the C2C subsystem was to validate communications using an asynchronous external interface. This would require two instances of the C2C subsystem, each implemented within a sepa-

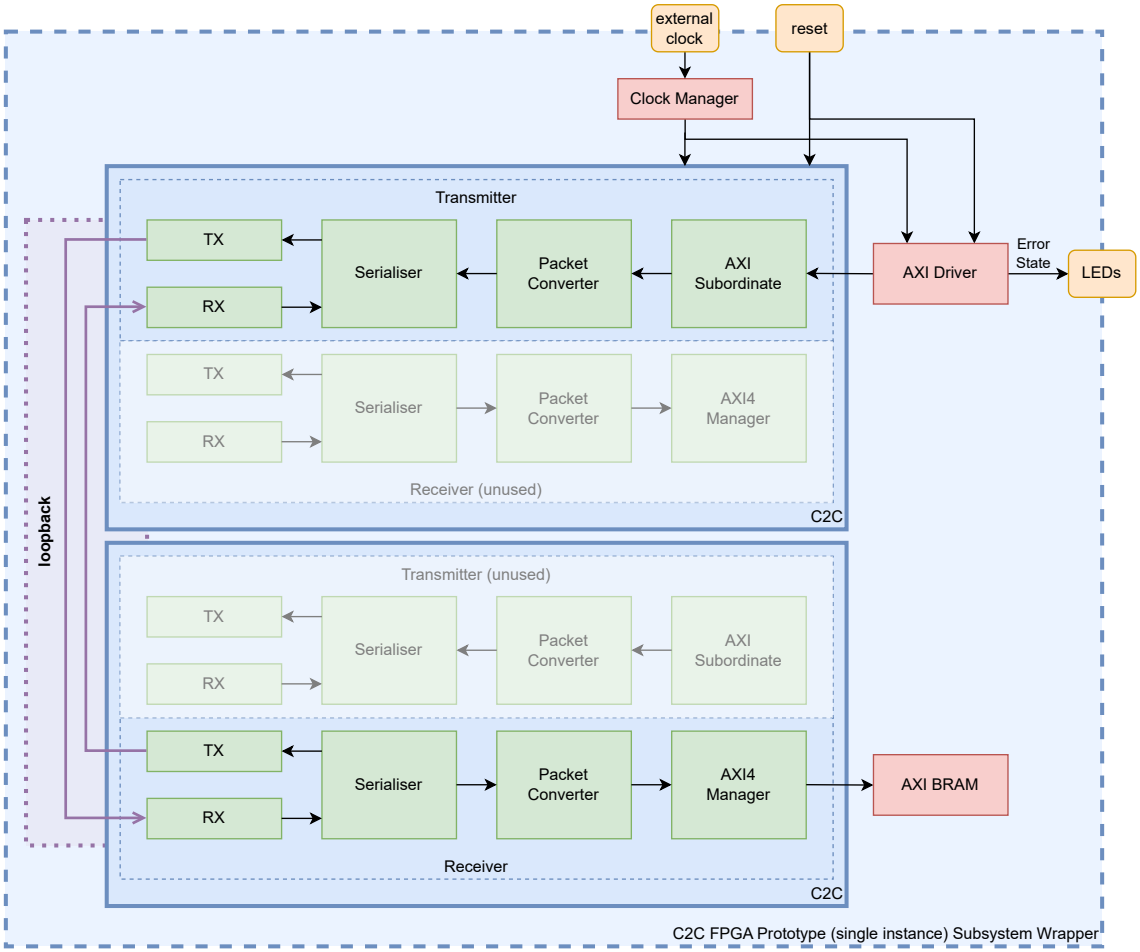

Figure 5.14. *High-level Architecture of the Single-board C2C FPGA Prototype*

rate clock domain. However, an incremental approach was taken in the implementation to minimise the number of unknowns in the design. Initially, an implementation with two C2C instances on a single FPGA (Figure 5.14), but with a loopback on the external interface, was created and validated. Only one of the two channels on each C2C instance was connected to keep the initial implementation simple. Following this, each instance was implemented on a separate FPGA, and they were connected via an external FMC connection to validate the asynchronous interface. Additionally, the C2C prototype was extended to create a peripheral bridge which could be used to support the Ballast ASIC (named Silta).

5.7.1 AXI Driver and AXI Memory modules

When implemented as a part of the Ballast SoC, the C2C subsystem is driven either by AXI transactions originating from other subsystems on the SoC or external transactions originating from the external C2C interface. To test the subsystem prototype effectively, the AXI Transaction Module was required to generate AXI transactions and exercise the subsystem. This module is connected to the AXI Subordinate interface of one of the C2C

modules and automatically starts sending transactions once the design reset is released. The AXI Transaction Module sends a write transaction followed by a read transaction, after which the read value is compared to the written value. The AXI Transaction Module is statically configurable and the following parameters can be controlled through generics:

- AXI address width,
- AXI data width,
- AXI ID width,
- The number of AXI transactions to be performed,
- Whether the address is incremented after each transaction or if the same address is used,
- The number of beats to be sent in each AXI incrementing burst transaction,
- The number of clock cycles to wait between sending transactions.

Similarly, requests received over the external C2C interface will be converted to AXI transactions and transmitted over the AXI Manager interface of the subsystem. As a result, there must be a module provided to serve these AXI requests. On the C2C prototype, an AXI BRAM IP was connected to the AXI Manager. The default size of the BRAM was set to 65kB and is statically configurable within the synthesis scripts.

5.7.2 Two Board Prototype Configuration

The C2C serial interface is designed to be asynchronous. Two connected designs must be placed in different clock domains to test this on the FPGA prototype. It is possible to achieve this using two designs on the same board, with each design using a dedicated clock manager IP configured to output a different frequency. However, this configuration would not emulate the more significant signal propagation delays on the C2C external interface. A decision was made to split the prototype over two boards (Figure 5.15), connected via the FMC interface on each board, over which the C2C interface signals would be transmitted. In addition to a higher emulation fidelity, splitting the design over two boards would create several FPGA design artefacts, which could be re-used to create Silta (see Section 5.7.3).

5.7.3 Ballast Peripheral bridge - "Silta"

Ballast was not designed with a typical high-capacity memory interface (e.g. LPDDR, flash, etc.) but included the C2C interface, which can extend the SoC. A proof of concept (named Silta) was created to demonstrate how the C2C interface of Ballast could be used to expand the memory capacity of the SoC, consisting of an FPGA with a C2C interface and an implemented DDR4 controller. When the Silta board was connected to Ballast

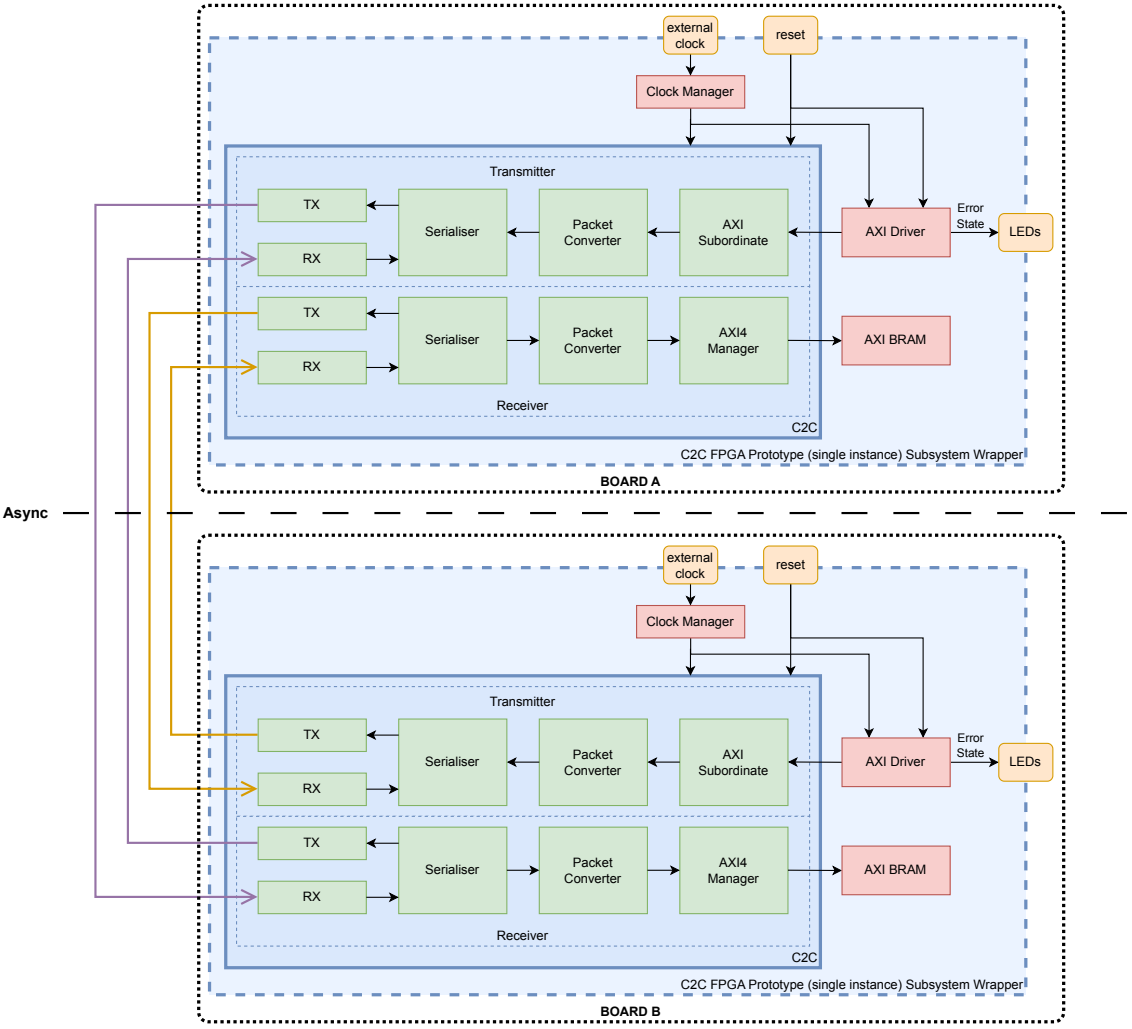

Figure 5.15. *High-level Architecture of the Two-board C2C FPGA Prototype*

over the C2C interface, the subsystems on Ballast could then access the DDR4 memory on the Silta board. The Ballast SoC was mounted onto a development board named Graniitti (Figure 5.16).

The Silta design re-used the C2C design files and constraints, which were used to develop the two-board prototype configuration. However, the AXI BRAM was replaced with an AXI DDR4 IP, and the design frequency was increased to 250MHz.

5.8 Top-Level Specific FPGA Prototyping Implementation

The main verification requirement targeted by the top-level configuration was to verify the entire JTAG chain of Ballast. This demanded a prototype configuration which contained SysCtrl, MPC and HPC. The interconnect was also included to allow communication between the subsystems in the design and to perform additional testing (Figure 5.17).

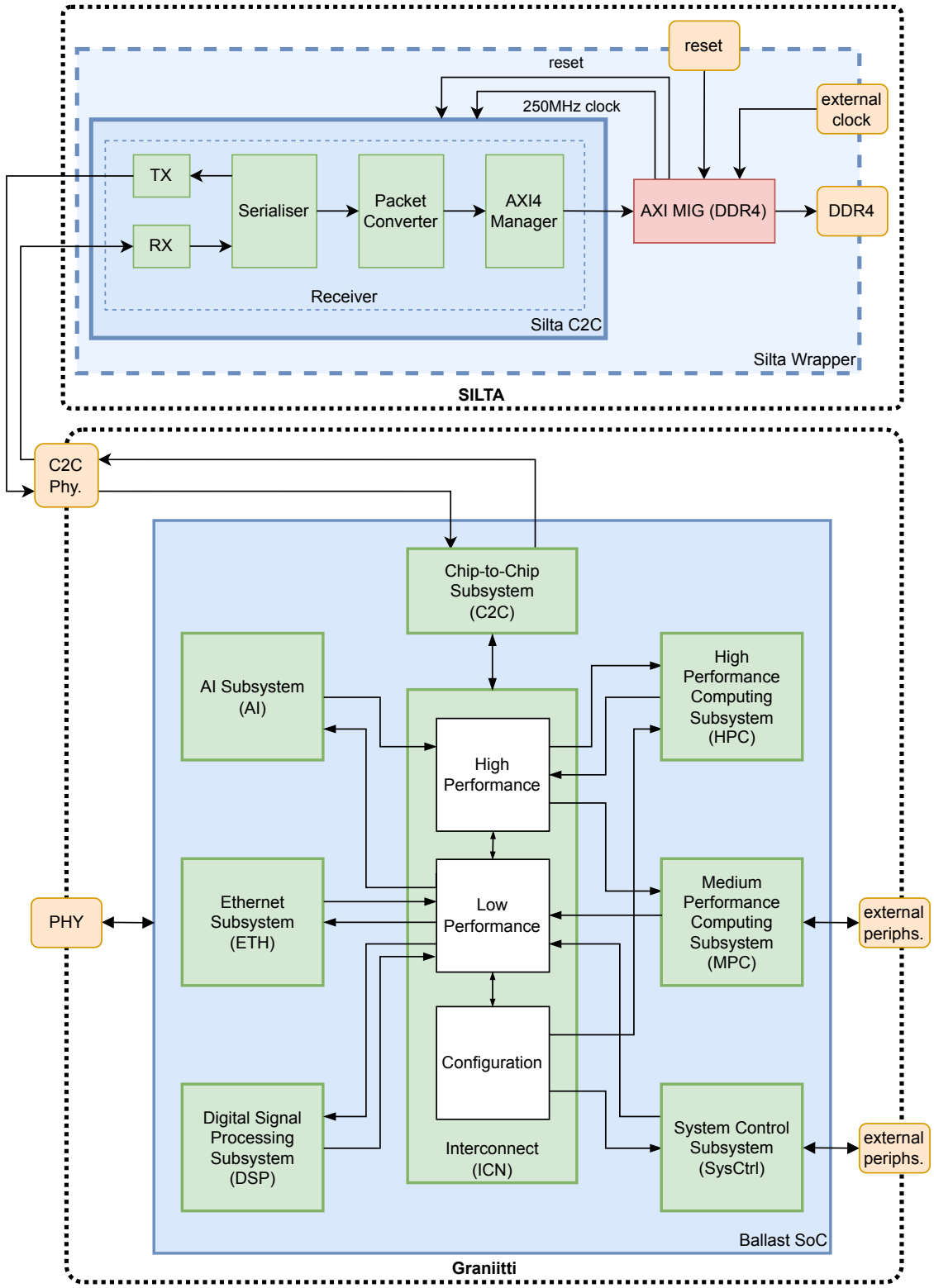

Figure 5.16. *High-level Architecture of Silta and Graniitti*

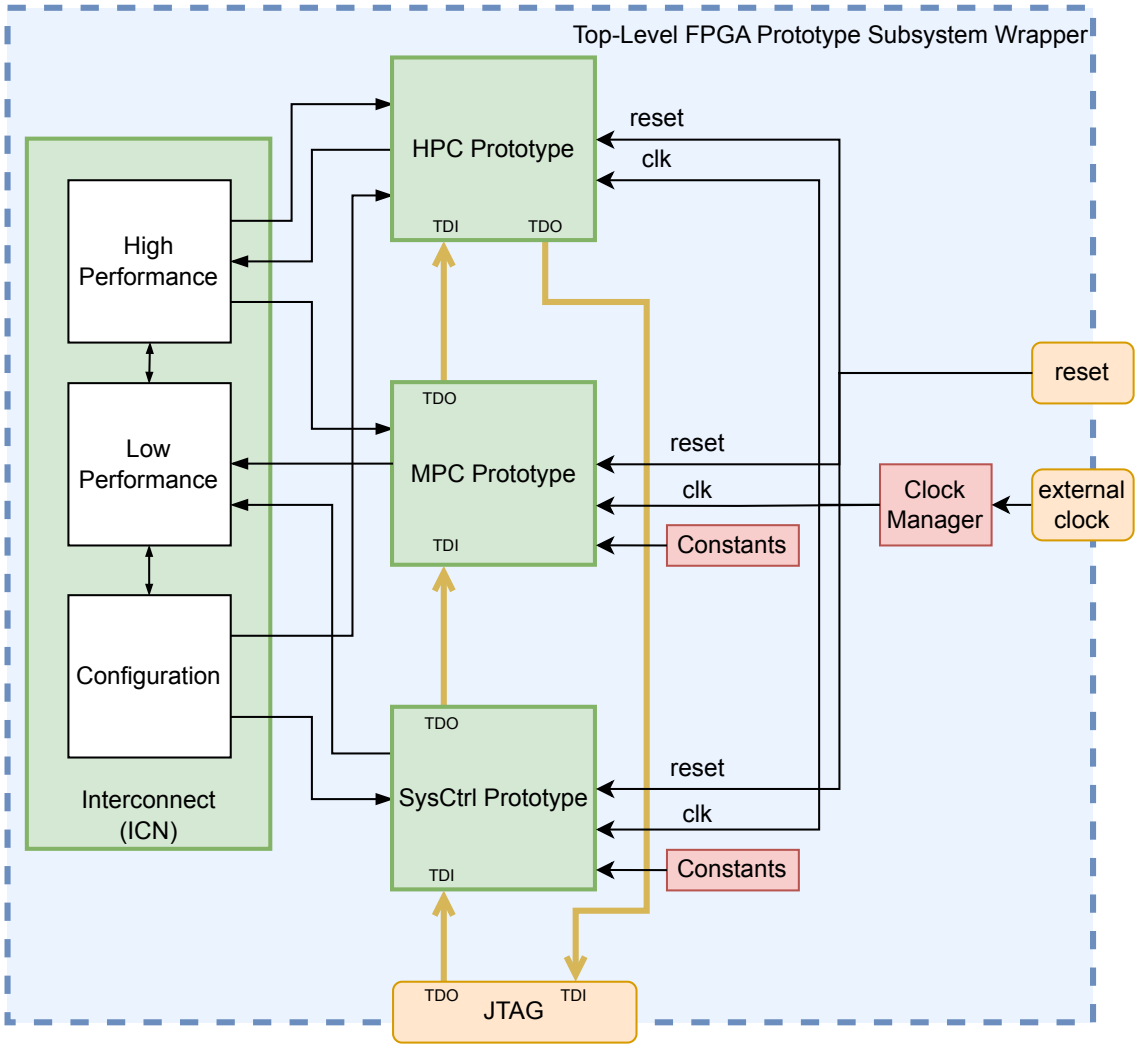

Figure 5.17. *High-level Architecture of the Top-Level FPGA Prototype*

5.8.1 Synthesis Flow

When creating the flow for the top-level configuration, an issue was identified with the synthesis flow, which had been used to prototype the individual subsystems. Some subsystems utilised design files with the same file and module name but contained different logic implementations. This is generally bad practice as it increases the risk of the wrong files being included for a specific design. Furthermore, the flow used for the other designs adds all RTL files to a Vivado project, and if files share the same name, there is no way to instruct the tool to associate a particular file with a specific module instance. This results in a non-deterministic build.

If this issue had been identified earlier in the flow, the file names would have been changed to be unique for each subsystem. However, the ASIC implementation flow was already mature by the time the top-level prototype was created and changing the file names would have created issues across the project. Therefore, an update to the flow was required. Each subsystem was synthesised independently for the top-level configu-

ration, and the netlist was exported. Once completed, the top-level project was created, and the netlist of each subsystem was imported into the top design, removing the dependency on design files and resolving the issue. In addition to fixing the conflicting file issues, this also improved the flexibility of the synthesis flow. If the design was monolithically synthesised, the entire design would need to be synthesised again if a single subsystem was updated, resulting in long synthesis times. However, with the updated flow, the top-level design can be synthesised modularly so that only the updated subsystem netlist needs to be re-synthesised. De-coupling the subsystems in this way should result in an improved synthesis time. However, this could not be measured, as building the design in a monolithic fashion was impossible due to the file name conflicts.

5.9 Verification of Implementation

As described within section 3.4.2, once the first iteration of an FPGA design is complete, it should be simulated within a testbench to verify the RTL behaviour. Once the RTL simulation is completed successfully, the design can be synthesised and the output netlist can be simulated within the testbench to ensure logical equivalence between the pre/post-synthesis design files. Following this completion, a bitstream of the design can be generated and loaded onto the FPGA platform upon which integration testing can be performed. The following sections outline the salient details of each verification step which was taken during the FPGA prototyping of Ballast.

5.9.1 RTL Simulation

Vivado was the primary development tool used to develop the Ballast prototype designs, and it comes with an integrated logic simulator. During the development process, it was found that the Vivado 2019.2 simulator did not support a number of the SystemVerilog structures used within the Ballast design. Siemens Questa had been used to simulate the RTL within the ASIC flow, and as a result, the Vivado simulation flow was reconfigured to use Questa when performing the simulation.

Within the Ballast ASIC development flow, testbenches for each of the subsystems had already been created and used to verify the subsystem RTL. These subsystem testbenches were also used to perform the RTL simulation on the prototype designs. Minor modifications to the testbench were required when running the post-synthesis simulations as the design netlist ports differed from the RTL design. Excepting this, test benches and infrastructure common to the ASIC development could be used.

5.9.2 Hardware Validation

After completing the testing of the prototype designs in simulation and the bitstreams generated, the designs were loaded onto the appropriate FPGA platform and then validated. The method used to validate each prototype configuration varied slightly between the prototypes.

If the prototype included a JTAG (all configurations apart from Silta/C2C), the initial test was to attempt to halt the CPU using openOCD and an external USB to JTAG adapter. Once this was successful, some basic firmware would be loaded onto the subsystem CPU using GDB and executed. The result of the firmware execution would be analysed using GDB, confirming that the prototype was successfully running.

For Silta and the C2C prototypes, there was no CPU contained within the design, so the validation process was different. As described within Section 5.7.1, AXI driving logic was included within the prototype configuration. The input signal to enable the AXI driver was routed to a switch, and the error-reporting output signals were routed to LEDs on the FPGA board. The design was initially synthesised and tested in the most basic configuration (single AXI bursts to a fixed address). Once this was complete, the design was incrementally modified and tested to increase the size of the feature set tested on the board. This approach was taken to ease the burden of debugging, as it would be easier to determine what feature was causing an issue.

5.10 Debugging the Hardware Design

An issue may become visible in the design when running on hardware, which was not identified during the simulation. This could be due to several reasons, including improperly constraining the physical interfaces of the design, which might lead to timing exceptions at the external interface of the design. Typical debugging methods (e.g., using a UART to print information to a terminal or using JTAG to access the debug module of a core) cannot accurately identify the root cause of an issue in hardware as they depend on the hardware design itself to work properly. The internal logic of the FPGA at runtime is not visible, and without additional tooling, it can be challenging to debug hardware issues. However, AMD-Xilinx FPGA development tools provide two IP cores, which make debugging in hardware significantly easier, namely the Integrated Logic Analyser (ILA) and Virtual Input/Output (VIO) cores.

5.10.1 AMD-Xilinx Integrated Logic Analyser

The ILA IP core can be inserted into the FPGA design selected to debug and monitor runtime signal behaviour. It allows users to store signal values over a period of time. It

provides features found in modern logic analysers, such as Boolean equations for triggering and edge transition triggers [5]. An example application of the ILA during the prototyping was to analyse the SDIO and SPI peripheral logic while developing the SysCtrl BootROM. The ability to observe the logic states accelerated the development process and allowed hardware issues to be identified very quickly.

5.10.2 AMD-Xilinx Virtual Input/Output

The VIO IP core is similar to the ILA in that it can be inserted into the target design and provides the ability to monitor signals. The VIO cannot store signal states over a period of time. However, it allows users to drive selected signals during runtime in real time [10]. The VIO core was used during the development of the Silta design to identify issues in the FPGA pin constraints. With the VIO, the resets of individual synchronous processes within the modules of Silta could be asserted and de-asserted independently of each other. Once this was in place, it was easy to identify the logic blocks generating erroneous data values and the reason for that.

6. RESULTS

This section qualifies how well the prototyping objectives outlined within Section 4.4.2 were met. Additionally, details of the hardware design issues identified as a result of the Ballast prototyping activities, the recognised limitations of prototyping and future improvements which could be applied to the prototyping implementation are also described.

6.1 Review of Objectives

6.1.1 Prototype Build Flow Development

As described in Section 5.3.4, an extensible and modular FPGA prototype synthesis flow was successfully created. This flow was applied to the standalone subsystem prototyping configurations (SysCtrl, MPC and HPC). The flow configuration is performed by changing the value of a limited number of variables and is not dependent on any configuration-specific artefacts. As a result, the FPGA prototyping synthesis flow could be easily re-used in future projects.

In Section 5.8.1, it is described that for the top-level configuration, a modified version of the flow was created to overcome an issue in file name conflicts. Alternative approaches to fixing this issue, such as re-naming files, would have allowed the synthesis flow to remain unchanged. However, this was not possible for Ballast due to project time constraints.

6.1.2 Validation of SoC Boot Design

The prototyping configuration created for SysCtrl was successfully used to test the Ballast boot process design thoroughly. The complete design and testing process of the Ballast SoC boot flow is documented within [39] and is therefore not repeated here. The main advantage of using the FPGA prototype to test the boot design was that the testing could be completed with the real off-chip storage devices (SD cards) which were to be used by the ASIC. The available RTL simulation environment SD card models were not comprehensive and could not be trusted to verify the entire boot operation. Furthermore, tests using the firmware stored within the SysCtrl bootROM could be executed quicker than

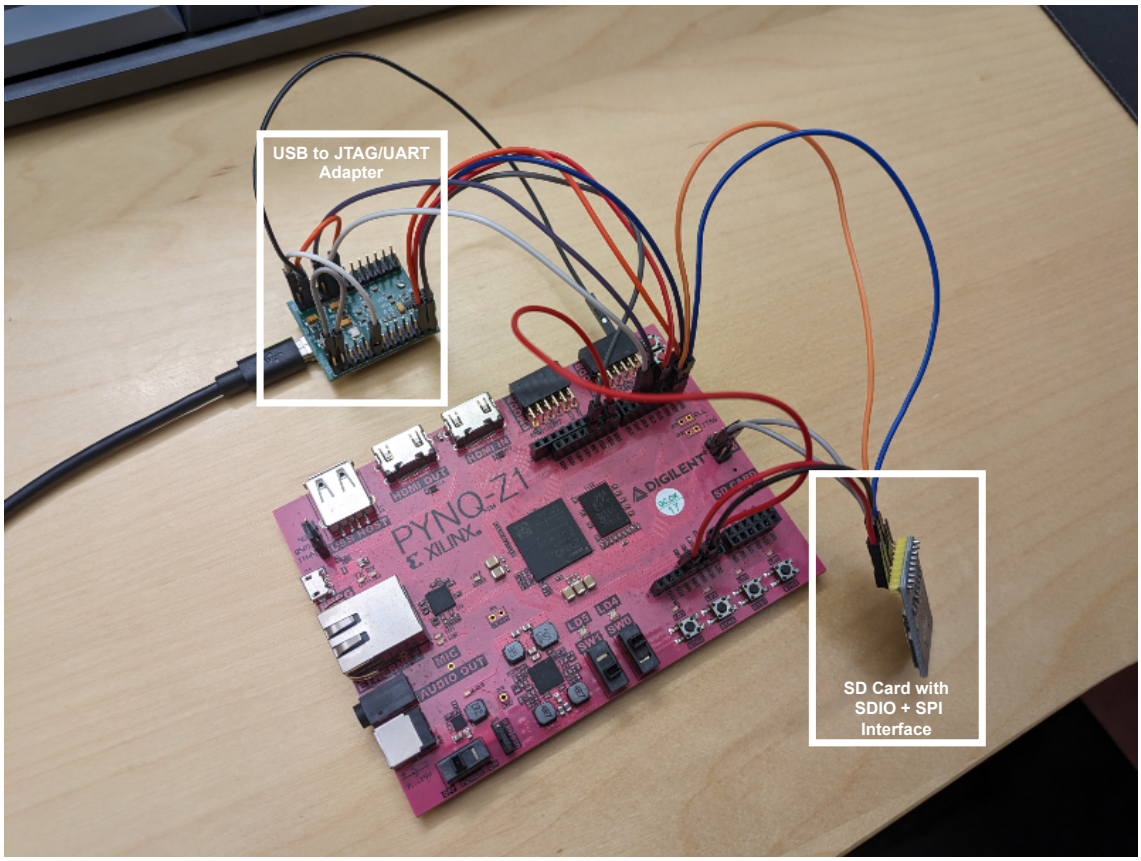

Figure 6.1. *FPGA Prototype Board Configuration Used to Validate SysCtrl Boot*

in RTL simulation, allowing for faster development times. The boot flow of Ballast was successfully tested using the sample chips after tape-out without any failures.

The physical prototyping configuration used to validate the boot testing is shown in Figure 6.1.

6.1.3 Validation of Debug Architecture

The top-level prototype configuration contained all of the Ballast SoC subsystems accessible via the JTAG interface on the chip. This configuration successfully provided a platform which could be used to interface with a debugging solution running on a separate host machine. The debugging solution used to test the JTAG chain was a combination of a USB-to-JTAG adapter hardware with OpenOCD. Tests were performed using the FPGA prototype to ensure that each subsystem could be controlled as expected via the JTAG. The tests not only validated the hardware design of the debug infrastructure but led to the creation of scripts which could be used for debugging the ASIC during the bring-up activities. After tape-out, the same debugging architecture tests were successfully performed. Figure 6.2 contains an extract of OpenOCD successfully identifying all of the JTAG taps within Ballast and subsequently halting using the SysCtrl RISC-V debug components to halt the Ibex core within SysCtrl.

```
Open On-Chip Debugger 0.12.0+dev-03051-g2427f585c (2023-09-25-13:04)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : clock speed 800 kHz
Info : JTAG tap: riscv_hpc.cpu tap/device found: 0x42acdc43 (mfg: 0x621 (Shenzhen Sooner Industrial Co Ltd), part: 0x2acd, ver: 0x4)
Info : JTAG tap: auto0.tap tap/device found: 0x10102001 (mfg: 0x000 (<invalid>), part: 0x0102, ver: 0x1)
Info : JTAG tap: riscv_mpc.cpu tap/device found: 0x249511c3 (mfg: 0x0e1 (Wintec Industries), part: 0x4951, ver: 0x2)
Info : JTAG tap: auto1.tap tap/device found: 0x10102001 (mfg: 0x000 (<invalid>), part: 0x0102, ver: 0x1)
Info : JTAG tap: riscv_sysctrl.cpu tap/device found: 0x249511c3 (mfg: 0x0e1 (Wintec Industries), part: 0x4951, ver: 0x2)
Info : [riscv_sysctrl.cpu] datacount=2 progbufsize=8
Info : [riscv_sysctrl.cpu] Examined RISC-V core; found 1024 harts
Info : [riscv_sysctrl.cpu]  XLEN=32, misa=0x40001104
[riscv_sysctrl.cpu] Target successfully examined.
Info : starting gdb server for riscv_sysctrl.cpu on 3333
Info : Listening on port 3333 for gdb connections
Ready for Remote Connections
Info : tcl server disabled
Info : telnet server disabled
```

HPC

MPC

SYSCTRL

Figure 6.2. *OpenOCD TAP Report for Top-level FPGA Configuration*

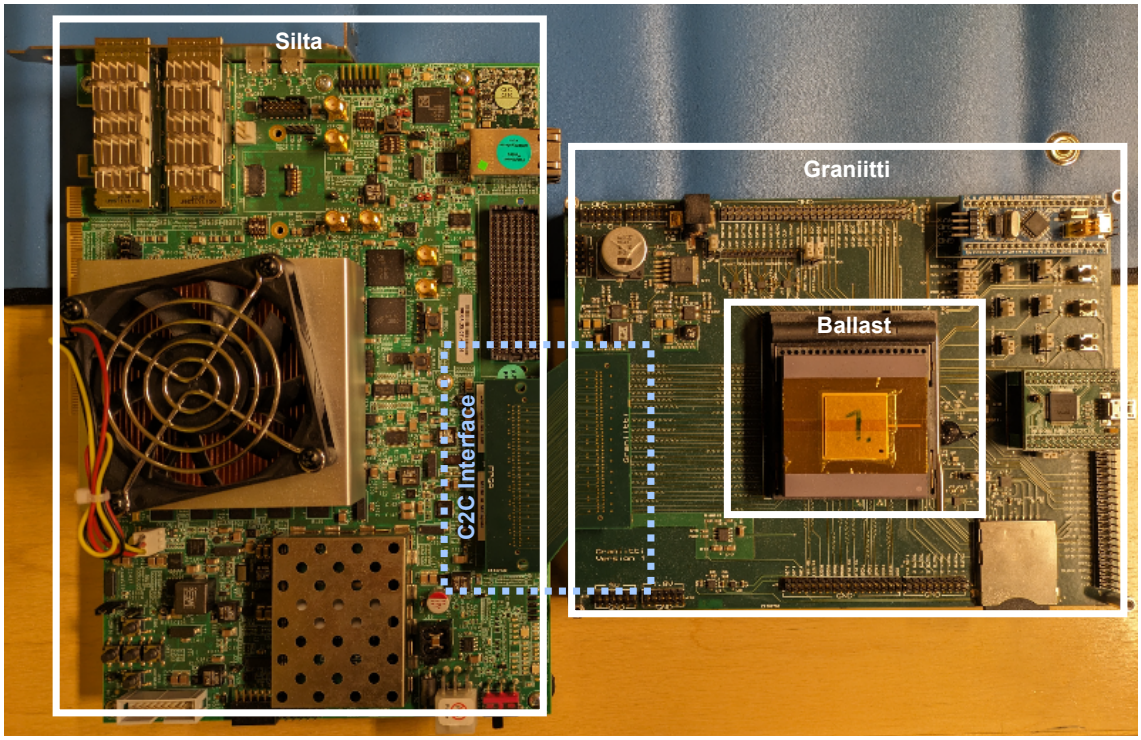

Figure 6.3. *Silta connected to Graniitti and Ballast*

6.1.4 Validation of C2C Interface

The main aim of prototyping the C2C interface on FPGA was to test the asynchronous interface. A fully asynchronous interface can be created with a C2C module on two separate FPGA boards connected by a physical connector. Additionally, the signal propagation delays across the wires in the connector improve the fidelity of the verification platform. See [28] for full details of the verification of the Ballast C2C subsystem. The C2C prototype was then used as a baseline to create the Silta board, which extends Ballast over the C2C interface (see Figure 6.3).

6.1.5 Validation of SoC Peripheral Interfaces

The SysCtrl and MPC subsystems contained several standard peripheral interfaces. Using the FPGA prototype platform for these subsystems, it was possible to test the operation of these interfaces using representative hardware. The tested peripheral interfaces were UART, CPI, I2C, SPI, SDIO and GPIO. The specifics of each test executed using each peripheral varied, but generally, testing was performed to check the configuration of the peripherals worked as expected and that communication to an external hardware component over the interface was possible.

For example, the JTAG, UART, CPI and I2C interfaces within MPC were tested using the Omnvision OV7670 camera module [12]. The camera module was configured using the I2C interface, and image data was captured using the CPI interface. Once an entire frame had been transmitted, the image data was read from MPC memory and sent over the UART to a host device. The host device captured the transmitted UART data and rendered the image data for display using a Python script.

Additionally, the SPI and SDIO interfaces were tested through the development of the bootROM described within Section 6.1.2. These interfaces were used to read/write the SD Card during boot. The JTAG interface was indirectly and continuously tested during the testing of all other peripheral interfaces, as it was the mechanism used to load code into the subsystems for testing. Therefore, all testing performed on subsystems that were running software-based tests were also validating the JTAG interface.

A list of all functional tests performed (including peripheral tests) using the Ballast FPGA prototypes are listed within Tables 6.1, 6.2 and 6.3. The external components used to test peripheral functions are listed. Tests without peripheral components listed are testing internal subsystem components and do not require external components aside from the JTAG adapter used to load/control the code. The functional tests for SysCtrl and MPC were performed using the Pynq-Z1 platform and the HPC tests were performed using the ZCU104 platform. Where possible, the code for the integration testing within RTL simulation as re-used.

The successful running of these tests provided confidence that the RTL design of the Ballast peripherals was correct. Furthermore, it was possible to reuse the same tests when testing the delivered ASIC samples.

6.1.6 Provision of Platform for BSP Development

A BSP is a software layer responsible for abstracting the hardware implementation details from higher software layers. It includes low-level code, which is used to access the hardware registers of the device [50]. For the Ballast SoC, the software written to perform the

Table 6.1. *Listing of SysCtrl Functional Tests Performed During Prototyping*

Functional Test	External Components Used
Boot	FT2232H Mini-Module
JTAG	-
Register Connectivity	-
Memory Access	-
Advanced Timer	-
APB Timer	-
GPIO	Logic Analyser
SDIO	SD Card Adapter
SPI	SD Card Adapter
UART	FT2232H Mini-Module
Local Interrupts	-

Table 6.2. *Listing of MPC Functional Tests Performed During Prototyping*

Functional Test	External Components Used
JTAG	FT2232H Mini-Module
Register Connectivity	-
Memory Access	-
GPIO	Logic Analyser
SPI	SD Card Adapter
UART	FT2232H Mini-Module
I2C	OV7670 Camera
CPI	OV7670 Camera
Local Interrupts	-

Table 6.3. *Listing of HPC Functional Tests Performed During Prototyping*

Functional Test	External Components Used
JTAG	FT2232H Mini-Module
Register Connectivity	-
Memory Access	-
APB Timer	-
Local Interrupts	-

integration tests in the simulation was written in the C programming language. However, there is ongoing research into applying the Rust programming language in embedded devices at the university. The FPGA prototyping platforms provided a suitable testing platform using both languages. The significant improvement in software execution time

when using the FPGA prototypes relative to simulation will likely reduce the BSP software development speed as new design iterations could be tested at a higher frequency. Furthermore, the ability to use real hardware when developing the driver code improved the confidence that could be gained from testing the implementation. This level of confidence could not be gained from simulation as the hardware models used to simulate peripheral functions were basic. The physical prototyping configuration used to develop the initial software artefacts for HPC can be seen in Figure 6.4.

A limitation of the FPGA prototyping platforms concerning the BSP development was caused by the differences in the hardware architecture when compared with the ASIC (see Sections 5.4 - 5.8. The FPGA prototypes were developed to try and minimise these differences as much as possible. However, simplifications to the clocking architecture resulted in small regions of functionality for which the BSP software could not be tested. For example, the PLL configuration could not be tested using the FPGA prototyping platforms because it was replaced with a simplified clocking structure in the FPGA implementation. To minimise the risk caused by the differences between the FPGA and ASIC implementations, the PLL configuration was thoroughly tested in RTL simulation and post-synthesis GLS.

After the tape-out of Ballast was complete and the sample chips were delivered, it was found that all of the BSP code developed using the FPGA prototypes could be reused to perform the wake-up testing of the SoC. The advantage of having the FPGA prototypes available early in the software development process ensured that the risk of software issues being present on the chip was reduced.

6.2 Identification of SDIO Hardware Design Issues

During the FPGA prototyping of SysCtrl, a significant bug in the hardware design relating to boot operation was identified. One of the interfaces used by the SysCtrl bootROM is the SDIO, which reads code into the Ballast SoC from an SD card during boot. To access the SD card memory via SDIO, the card must be first initialised for operation through a defined set of commands the host sends. This initialisation command sequence is managed by hardware in the SysCtrl boot design. In the early development stages of Ballast, boot operation was tested using an open-source SD card VIP. The SD card VIP's accuracy was unknown at the time, and an alternative IP could not be found. No issues were identified within the SDIO boot functionality during the RTL simulation testing of the early hardware designs using this VIP. However, when the equivalent testing was performed using the SysCtrl FPGA prototype, it was observed that the boot functionality failed to initialise the SD card for operation. Following an investigation, it was discovered that the command sequence sent by the hardware was not correct.

Additionally, it was discovered that there was a bug in the SD card VIP used, in that it

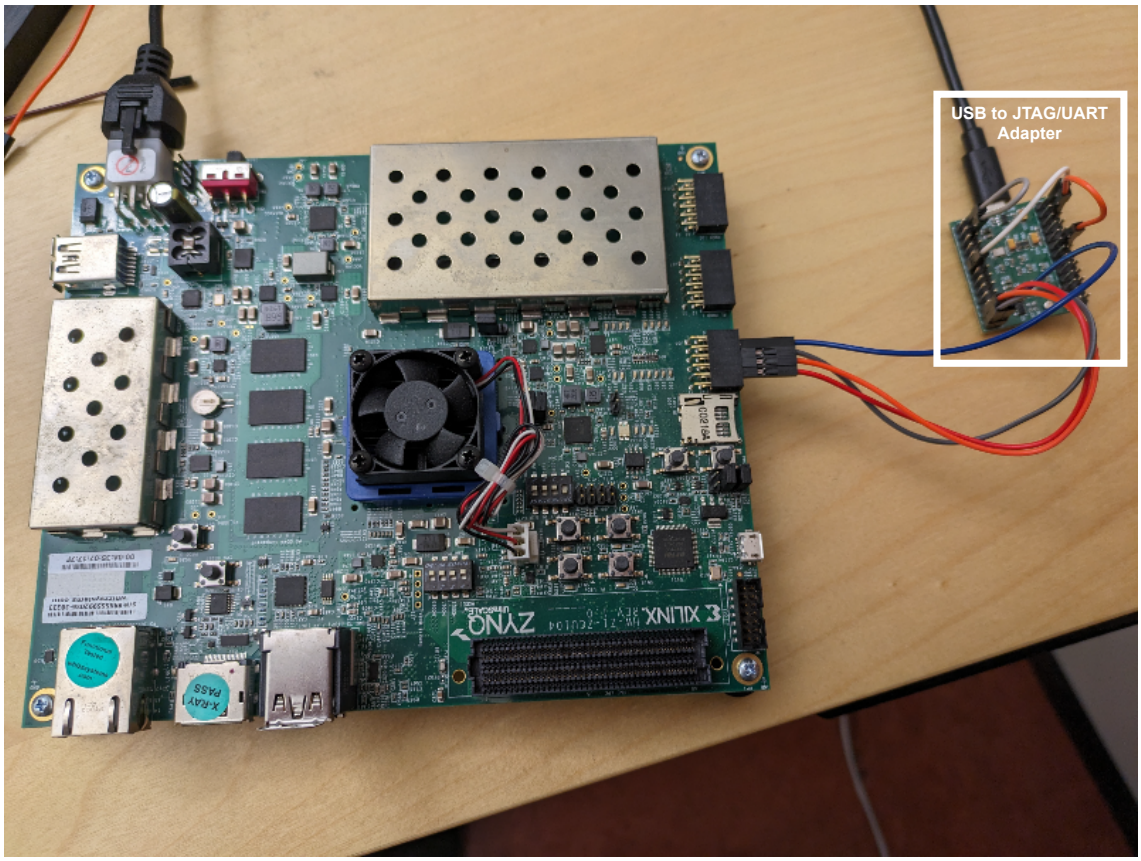

Figure 6.4. *FPGA Prototype Board Configuration Used to Develop Initial Software Artefacts HPC*

was possible to initialise the VIP without sending the correct command set. A hardware fix was immediately implemented, and the SysCtrl FPGA prototype was re-synthesised. The test was repeated using the updated design and the result confirmed that the SD card was successfully initialised.

Identifying and eliminating this bug demonstrated the value provided by the FPGA prototyping activity on Ballast. Testing the boot functionality using a prototype platform connected to the representative hardware made it possible to derisk the critical boot functions in a way that was not possible in RTL simulation. Furthermore, the use of the prototype platform enabled the ability to quickly and easily synthesise updated hardware design in order to test the fix, reducing the overall development time required.

6.3 Limitations of FPGA Prototyping

6.3.1 Development Complexity

FPGAs are complex devices requiring domain-specific knowledge to implement designs effectively. Using FPGAs to prototype ASIC designs brings advantages, such as using the same RTL design files for both the ASIC and FPGA designs. However, the technol-

ogy differences between an FPGA and an ASIC can be significant, especially when the ASIC design is a complex SoC containing multiple clock domains and external interfaces. These differences create a need to modify the design files to implement a functional prototype of the design (see Chapter 5). Making the modifications requires the engineer to understand both the ASIC design intent/implementation and the target FPGA platform being used for prototyping.

The design effort calculations presented within [43] determined that the FPGA prototyping of Ballast accounted for 21% of the effort required to develop the SoC. As already described within [43] and earlier within this section, the high amount of effort can be attributed to the complexity of the SoC design and the large number of modifications required to prototype it successfully. The FPGA prototyping team also comprised a mixture of junior and more experienced engineers. While the experienced engineers had good experience in FPGA development, they had little experience in ASIC emulation and had not previously used AMD-Xilinx tools.

6.3.2 Prototype Performance

As mentioned previously, one of the challenges of implementing an ASIC design on FPGA is that the design is not naturally optimised for the FPGA platform. While the FPGA design tools can synthesise such a design successfully, the performance (i.e. clock frequency) achieved by the design will almost certainly be reduced. It is possible to improve the performance of the prototype by modifying the design. However, great care must be taken when modifying the design. This is to ensure that the accuracy of the design being verified is not compromised, as every modification creates a delta between the prototype and the ASIC design and reduces the value of the verification activity.

On Ballast, this resulted in a maximum achievable clock frequency of 10MHz for the SysCtrl, MPC and HPC prototypes. While this speed allows for reduced test run times compared to simulation, it is still significantly slower than the targeted frequency of the ASIC. This performance did not create issues during the Ballast development as all testing was performed using simple bare-metal software tests. However, this performance restriction could become problematic if testing more complex software requires an operating system or high-level software libraries.

6.3.3 Technology Differences

When prototyping an SoC, key areas of functionality are unavoidably affected by the technology differences between an ASIC and an FPGA. Section 5.4 outlines the functional areas affected during the prototyping of Ballast. These changes must be factored into the verification plan to ensure that the gaps in coverage which are created by modifying the

design are satisfied using alternative verification techniques.

An example of improperly considering the verification coverage gaps within FPGA prototyping can be found within the DSP subsystem implemented within the Ballast SoC. As mentioned within Section 4.5, the DSP subsystem was developed as an independent IP and integrated into the Ballast SoC. The design was verified using an FPGA prototype and in RTL simulation. However, following tape-out, a bug was discovered within the internal memory interface of the DSP, which prevented the memory from being accessible on the ASIC. This issue would not have been found using the FPGA prototype because the memory interface was replaced with an FPGA-compatible module.

Furthermore, this issue was not identified within RTL simulation, as a simplified memory model was used to simulate the subsystem. The issue was identified when the subsystem was simulated using accurate memory models within a post-synthesis or post-layout netlist. However, due to human error, the wrong memory file was specified within the file lists used to tape-out the ASIC. This demonstrates that multiple verification techniques, in addition to FPGA prototyping, must be used in combination to achieve the verification coverage required.

6.3.4 Verification Coverage

It is typical to use several coverage measures during the validation of an ASIC design to determine how thoroughly the verification activity has been performed [37]. Functional coverage and code coverage are examples. These coverage metrics can be used to drive RTL simulation based verification, as the tester has full visibility of the logic during simulation and the freedom to drive the design from an arbitrary location. However, this visibility and access to logic is not available within an FPGA prototype by default. Therefore, it prevents the tester from being able to use the same verification metrics which were used in RTL simulation to drive the testing on an FPGA prototype. As seen within [38], some has been performed to demonstrate that coverage-driven verification using an FPGA platform is possible, but a variable amount of additional effort is required to do so and no standardised methodology to apply this currently exists. As a result, a practical application of FPGA prototyping is to satisfy high-level validation objectives which compliment the coverage-driven RTL simulation results.

7. CONCLUSION AND FUTURE WORK

The development of modern MPSoCs is a complex task, with verification becoming more critical than ever as the resources required to thoroughly verify the design are increasing. This thesis presents an overview of FPGA-based prototyping methodologies, a review of the current related technologies and an analysis of a real-life application within the SoCHub project.

FPGA-based prototyping is typically used to provide an accurate, highly performant hardware model which can interface with real components within a lab without incurring high material costs. This thesis describes a set of high-level objectives to measure the effectiveness of using FPGA-based prototyping (Section 4.4.2). These objectives were derived from the Ballast SoC verification planning artefacts and describe the development and execution of the prototyping activities to satisfy those objectives. The results show that that the objectives were successfully met using FPGA prototyping, and also highlight several limitations which exist in FPGA prototyping-based verification. These results emphasise the value of FPGA prototyping and the need to use it with other verification methodologies to verify an SoC design thoroughly.

7.1 Future Work

The FPGA prototyping work conducted for the Ballast SoC focused on using standalone, single-board FPGA prototype configurations. As described within Section 3, FPGA prototypes can be created using multiple FPGAs and integrated with simulators to create a co-simulation environment. Future work could be performed to utilise these methods when prototyping a design of comparable complexity to evaluate the benefits and drawbacks. In addition, Section 3.3 describes the capabilities of modern FPGAs. It would also be interesting to assess whether or not these capabilities could be applied to FPGA prototyping activities to improve the process. For example, test automation and integration within CI tools could be achieved through the use of embedded CPUs and high-speed networking interfaces.

Moreover, further work could be performed to investigate how some of the limitations identified within this thesis could be eliminated. An example would be extending the work presented within [38] to create a generic framework that could be adapted to other

designs and enable the ability to drive FPGA prototypes using coverage metrics. An additional example would be to employ ML techniques similar to those outlined in [52] to improve the efficiency of performing the tasks within FPGA prototyping.

REFERENCES

[1] AMD. *pynq_io*. What is PYNQ? 2016. URL: http://www.pynq.io/ (visited on 05/13/2023).

[2] AMD. *vcu118_overview*. AMD Virtex UltraScale+ FPGA VCU118 Evaluation Kit. 2023. URL: https://www.xilinx.com/products/boards-and-kits/vcu118.html (visited on 05/13/2023).

[3] AMD. *zcu104_overview*. Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit. 2018. URL: https://www.xilinx.com/products/boards-and-kits/zcu104.html (visited on 05/13/2023).

[4] AMD-Xilinx. *Clocking Wizard v6.0 LogiCORE IP Product Guide*. 2021.

[5] AMD-Xilinx. *Integrated Logic Analyser v6.2 Product Guide*. 2016.

[6] AMD-Xilinx. *UltraFast Design Methodology Guide for FPGAs and SoCs UG949 (v2022.2)*. 2022.

[7] AMD-Xilinx. *UltraScale Architecture Clocking Resources User Guide*. 2021.

[8] AMD-Xilinx. *UltraScale Architecture GTY Transceivers User Guide*. 2021.

[9] AMD-Xilinx. *UltraScale Architecture SelectIO Resources User Guide*. 2019.

[10] AMD-Xilinx. *Virtual Input/Output v3.0*. 2018.

[11] AMD-Xilinx. *Vivado Design Suite User Guide UG893 (v2019.2)*. 2019.

[12] Arducam and Omnivision. *CMOS OV7670 Camera Module 1/6-Inch 0.3-Megapixel Module Datasheet*. 2015.

[13] SD Card Association. *SD Specification Part E1 - SDIO Simplified Specification Version 3.00*. SD Card Association, 2018. URL: http://applelogic.org/files/SDIO.pdf.

[14] René Beuchat et al. *Fundamentals of System-on-Chip Design on Arm Cortex-M Microcontrollers*. 2021.

[15] Abhishek Bhattacharjee, Gilberto Contreras, and Margaret Martonosi. “Full-system chip multiprocessor power evaluations using FPGA-based emulation”. In: (2008), p. 335. URL: http://portal.acm.org/citation.cfm?doid=1393921.1394010 (visited on 05/31/2022).

[16] Louise H. Crockett et al. *Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. First. Strathclyde Academic Media, 2014.

[17] Pasquale Davide Schiavone et al. “Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications”. In: (2017), pp. 1–8. URL: http://ieeexplore.ieee.org/document/8106976/ (visited on 03/25/2023).

[18] Wayne Wilson - Design Engineer Qualcomm Inc. San Diego and Calif. Michel Courtoy - Marketing Manager Aptix Corp. San Jose. “FPGA emulation speeds ASIC design”. English. In: *Electronic Engineering Times* (1996).

[19] Digilent. *pynq_z1_reference*. PYNQ-Z1_reference. 2016. URL: https://digilent.com/reference/programmable-logic/pynq-z1/reference-manual (visited on 05/13/2023).

[20] Paul Donahue and Tim Newsome. *RISC-V Debug Specification*. 2019. URL: https://github.com/riscv/riscv-debug-spec/tree/66c3117145 (visited on 08/20/2023).

[21] Paul J Fox, A Theodore Markettos, and Simon W Moore. "Reliably prototyping large SoCs using FPGA clusters". In: (2014), pp. 1–8. URL: http://ieeexplore.ieee.org/document/6861350/ (visited on 08/08/2023).

[22] Aleksei Gimbitskii. "INTERCONNECT DESIGN FOR THE EDGE COMPUTING SYSTEM-ON-CHIP". In: (2022).

[23] David J. Greaves. *Modern system-on-chip design on Arm*. Cambridge: arm Education Media, 2021. 564 pp.

[24] M. Gschwind, V. Salapura, and D. Maurer. "FPGA prototyping of a RISC processor core for embedded applications". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 9.2 (2001), pp. 241–250. ISSN: 1063-8210, 1557-9999. URL: http://ieeexplore.ieee.org/document/924027/ (visited on 05/31/2022).

[25] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th. San Fransisco: Elsevier Science & Technology, 2011.

[26] Michael Hübner and Jürgen Becker. "Multiprocessor System-on-Chip". In: (2011). URL: http://link.springer.com/10.1007/978-1-4419-6460-1 (visited on 07/25/2022).

[27] William N.N. Hung and Richard Sun. "Challenges in Large FPGA-based Logic Emulation Systems". In: *Proceedings of the 2018 International Symposium on Physical Design* (2018), pp. 26–33.

[28] Mohamed Ibrahim. "Chip-to-chip Interface Communication". In: (2022).

[29] Xilinx Inc. *UltraScale Architecture and Product Data Sheet: Overview (DS890)*. 2022, p. 50.

[30] Xilinx Inc. *UltraScale Architecture Configurable Logic Block User Guide (UG574)*. 2017, p. 58.

[31] Xilinx Inc. *UltraScale Architecture Memory Resources User Guide*. 2021, p. 138.

[32] Chase Lambert. *Makefile Tutorial*. Makefile Tutorial. 2023. URL: https://makefiletutorial.com/ (visited on 10/04/2023).

[33] Xuemei Li et al. "The FPGA Prototyping Implementation of LEON3 SoC". In: (2012), pp. 1643–1646. URL: http://ieeexplore.ieee.org/document/6322724/ (visited on 08/09/2023).

[34] Arm Limited. *AMBA AXI and ACE Protocol Specification (ARM IHI 0022H.c)*. 2021.

[35] Ashok B. Mehta. *Hardware/Software Co-verification*. Cham: Springer International Publishing, 2018, pp. 243–253. URL: http://link.springer.com/10.1007/978-3-319-59418-7_12 (visited on 05/31/2022).

[36] Ashok B. Mehta. *Introduction*. Cham: Springer International Publishing, 2018, pp. 1–4. URL: http://link.springer.com/10.1007/978-3-319-59418-7_1 (visited on 05/26/2022).

[37] P Mishra and N.D. Dutt. *Functional Verification of Programmable Embedded Architectures*. New York: Springer-Verlag, 2005. ISBN: 978-0-387-26143-0. URL: http://link.springer.com/10.1007/b137514 (visited on 08/11/2023).

[38] Dipakkumar Modi and Usha Mehta. “Coverage Driven Functional Testing Architecture for Prototyping System Using Synthesizable Active Agent”. In: *International Journal of VLSI Design & Communication Systems* 9.3 (2018), pp. 41–50. ISSN: 09761527, 09761357. URL: http://aircconline.com/vlsics/V9N3/9318vlsi04.pdf (visited on 08/09/2023).

[39] Antti Nurmi et al. “A Resilient System Design to Boot a RISC-V MPSoC”. In: (2022), pp. 232–238. URL: https://ieeexplore.ieee.org/document/9996852/ (visited on 06/17/2023).

[40] John K. Ousterhout and Ken Jones. *Tcl and the Tk toolkit*. 2nd Edition. Addison-Wesley, 2009. ISBN: 0-321-60176-9.

[41] OpenOCD Project. *OpenOCD User’s Guide*. 2022. URL: https://openocd.org/pages/documentation.html.

[42] Antonio Pullini et al. “uDMA: An autonomous I/O subsystem for IoT end-nodes”. In: (2017), pp. 1–8. URL: http://ieeexplore.ieee.org/document/8106971/ (visited on 03/25/2023).

[43] Antti Rautakoura et al. “Ballast: Implementation of a Large MP-SoC on 22nm ASIC Technology”. In: (2022), pp. 276–283. URL: https://ieeexplore.ieee.org/document/9996602/ (visited on 02/25/2023).

[44] Antti Rautakoura et al. “Kamel: IP-XACT compatible intermediate meta-model for IP generation”. In: (2020), pp. 325–331. URL: https://ieeexplore.ieee.org/document/9217651/ (visited on 07/24/2022).

[45] Pasquale Davide Schiavone et al. “Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX”. In: (2018), pp. 1–3.

[46] H. Selvaraj, P. Sapiecha, and N. Dhavlikar. “Partitioning of large HDL ASIC designs into multiple FPGA devices for prototyping and verification”. In: (2001), pp. 411–415. URL: http://ieeexplore.ieee.org/document/970504/ (visited on 05/30/2022).

[47] Lattice Semiconductor. *iCE40 LP/HX Family Data Sheet FPGA-DS-02029-4.0*. 2022, p. 54.

[48] Gina R. Smith. *FPGAs 101: everything you need to know to get started*. Amsterdam ; Boston: Newnes, 2010. 229 pp. ISBN: 978-1-85617-706-1.

[49] IEEE Computer Society. *IEEE Standard Test Access Port and Boundary-Scan Architecture*. IEEE, 2008.

[50] John Taylor and Wayne Taylor. *Patterns in the Machine: A Software Engineering Guide to Embedded Development*. Apress, 2021. ISBN: 978-1-4842-6439-3.

[51] Russell Tessier. “MULTI-FPGA SYSTEMS: LOGIC EMULATION”. In: . *Reconfigurable computing* (2007).

[52] Divyasree Tummalapalli et al. “Novel Design partitioning technique for ASIC prototyping on multi-FPGA platforms using Graph Deep Learning”. In: (2022), pp. 1–4. URL: https://ieeexplore.ieee.org/document/9970882/ (visited on 08/09/2023).

[53] Andrew Waterman, Krste Asanovic, and CS Division. *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA*. 2019.

[54] Andrew Waterman et al. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. 2021.

[55] W. Wolf, A.A. Jerraya, and G. Martin. “Multiprocessor System-on-Chip (MPSoC) Technology”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.10 (2008), pp. 1701–1713. ISSN: 0278-0070. URL: http://ieeexplore.ieee.org/document/4627532/ (visited on 05/26/2022).

[56] Roger Woods et al. *FPGA-based implementation of signal processing systems, Second edition*. eng. 2nd ed. Wiley, 2017. ISBN: 9781119077978.

[57] Haigang Yang et al. “Review of advanced FPGA architectures and technologies”. In: *Journal of Electronics (China)* 31.5 (2014), pp. 371–393. ISSN: 0217-9822, 1993-0615. URL: http://link.springer.com/10.1007/s11767-014-4090-x (visited on 08/01/2022).

[58] Florian Zaruba and Luca Benini. “The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (2019), pp. 2629–2640. ISSN: 1063-8210, 1557-9999. URL: https://ieeexplore.ieee.org/document/8777130/ (visited on 03/25/2023).

APPENDIX A: APPENDIX A

Listing A.1. *Example Verilog module definition used to infer a single port BRAM memory*

```
1  module single_port_ram_memory #(
2    parameter RAM_WIDTH = 18,
3    parameter RAM_DEPTH = 1024,
4    parameter INIT_FILE = ""
5  ) (
6    input [clogb2(RAM_DEPTH-1)-1:0] addra,
7    input [RAM_WIDTH-1:0] dina,
8    input clka,
9    input wea,
10   input ena,
11   output [RAM_WIDTH-1:0] douta
12 );
13
14   reg [RAM_WIDTH-1:0] BRAM [RAM_DEPTH-1:0];
15   reg [RAM_WIDTH-1:0] ram_data = {RAM_WIDTH{1'b0}};
16
17   generate
18     if (INIT_FILE != "") begin: use_init_file
19       initial
20         $readmemh(INIT_FILE, BRAM, 0, RAM_DEPTH-1);
21     end else begin: init_bram_to_zero
22       integer ram_index;
23       initial
24         for (ram_index = 0; ram_index < RAM_DEPTH;
25                             ram_index = ram_index + 1)
26           BRAM[ram_index] = {RAM_WIDTH{1'b0}};
27     end
28   endgenerate
29
30   always @(posedge clka)
```

```
31      if (ena) begin
32        if (wea)
33          BRAM[addra] <= dina;
34        ram_data <= BRAM[addra];
35      end
36
37    assign douta = ram_data;
38
39    //  The following function calculates the address width based
40    //  on specified RAM depth
41    function integer clogb2;
42      input integer depth;
43        for (clogb2=0; depth>0; clogb2=clogb2+1)
44          depth = depth >> 1;
45    endfunction
46
47  endmodule
```

Listing A.2. *Temporary SysCtrl BootROM*

```
1  module fpga_bootrom #(
2    parameter ADDR_WIDTH=32,
3    parameter DATA_WIDTH=32
4  ) (
5     input logic                    CLK,
6     input logic                    CEN,
7     input logic [ADDR_WIDTH-1:0]   A,
8     output logic [DATA_WIDTH-1:0] Q
9  );
10
11   assign Q = 32'h0000006f; // jal x0,0
12
13 endmodule
```

Listing A.3. *SysCtrl Clock Gate for FPGA*

```
1  // clock gating implemented using BUFGCE
2  module pulp_clock_gating_fpga (
3     input  logic clk_i,
4     input  logic en_i,
5     input  logic test_en_i, // not connected
6     output logic clk_o
```

```
7  );
8
9    BUFGCE BUFGCE_inst (
10     .O(clk_o),
11     .CE(en_i), // CE = 0, clock disabled
12     .I(clk_i)
13   );
14
15 endmodule
```

Listing A.4. *IO Pad Implementation for FPGA*

```
1  module tico_pad_functional_wrapper_fpga #(
2    // 1 = ETH, 2 = GENERAL, 20 = GENERAL_PD, 21 = GENERAL_PU
3    parameter PAD_TYPE = 20,
4    parameter PAD_CONF_WIDTH = 10
5  ) (
6    input logic [PAD_CONF_WIDTH-1:0] conf_in,
7    input logic I,
8    output logic O,
9    inout logic PAD
10 );
11   generate
12     if (PAD_TYPE == 2) begin
13       // not implemented on FPGA
14     end
15     if (PAD_TYPE == 20) begin
16       // conf_in[5] == 0: OUTPUT
17       // conf_in[5] == 1: INPUT
18
19       (* PULLDOWN = "YES" *)
20       IOBUF iobuf_i (
21         // T == 0, PAD = OUTPUT
22         // T == 1, PAD = INPUT (tristated)
23         .T ( conf_in[5] ),
24         .I ( I     ),
25         .O ( O     ),
26         .IO( PAD   )
27       );
28     end
29     if (PAD_TYPE == 21) begin
```

```
30          // conf_in[5] == 0: OUTPUT
31          // conf_in[5] == 1: INPUT
32
33          (* PULLUP = "YES" *)
34          IOBUF iobuf_i (
35            // T == 0, PAD = OUTPUT
36            // T == 1, PAD = INPUT (tristated)
37            .T ( conf_in[5] ),
38            .I ( I     ),
39            .O ( O     ),
40            .IO( PAD   )
41          );
42        end
43      endgenerate
44
45  endmodule
```