A Resilient System Design to Boot a RISC-V MPSoC

Antti Nurmi*, Antti Rautakoura, Henri Lunnikivi, Timo D. Hämäläinen
*Nokia, Finland – antti.nurmi@nokia.com
Tampere University, Finland – {antti.rautakoura, henri.lunnikivi, timo.hamalainen}@tuni.fi

***Abstract*—This paper presents a highly resilient boot process design for Ballast, a new RISC-V based multiprocessor system-on-chip (SoC). An open source RISC-V SoC was adapted as a bootstrap processor and customized to meet our requirement for guaranteed chip wake-up. We outline the characteristic challenges of implementing a large program into a read-only memory (ROM) used for booting and propose generally applicable workflows to verify the boot process for application specific integrated circuit (ASIC) synthesis.**

We implemented four distinct boot modes. Two modes that load a software bootloader autonomously from an SD card are implemented for a secure digital input output (SDIO) inteface and for a serial peripheral interface (SPI), respectively. Another SDIO based mode allows for direct program execution from external memory, while the last mode is based on usage of a RISC-V debug module. The boot process was verified with instruction set simulation, register transfer level simulation, gate-level simulation and field-programmable gate array prototyping. We received the fabricated ASIC samples and were able to successfully boot the chip via all boot modes on our custom circuit board.

***Index Terms*—RISC-V, SoC, boot, bootROM, FPGA, ASIC**

I. Introduction

A functional boot process is an essential requirement to enable an autonomous wake up for a system-on-chip (SoC) and abstract low-level system functionality to aid software development. Boot process implementations can differ in the amount of fixed hardware functionality before any central processing unit (CPU) instructions, the complexity of the boot code and the memory resources that are used to host it. Typically, after reset de-assertion, the boot CPU's program counter will point to a set of hardcoded instructions in an internal read-only memory (ROM) called the bootROM. The size and complexity of the bootROM can vary dramatically, from the simplest implementations with a single jump instruction to some form of programmable memory, to large autonomous software routines. [1]

SoCs designed for and implemented to application specific integrated circuits (ASIC) can be found in both academia and industry, with academic designs more commonly released as open source. Academic SoC designs typically favor specialization towards a limited set of metrics such as computing performance or energy efficiency, while commercial SoC designers are also incentivised to create flexible and easily usable general purpose products. In our research project SoC-Hub[1], we developed Ballast, a large general purpose multiprocessor SoC (MPSoC) comparable in complexity to commercial edge computing devices, with the aim of creating a reusable SoC template for a set of three chips, and later for other projects. The goal with Ballast is to provide a base platform for MPSoC application development, at the base of which is a flexible boot concept with support for autonomous operation and multiple alternative modes.

The functionality of a SoC boot process can be compromised at a number of different stages, which at worst can render the ASIC completely unusable. Design-time logic errors may slip past verification and may compromise the boot process, depending on their location of occurrence. Critical components are the boot CPU and the interconnects, memories and peripherals that it utilizes. The manufacturing yield of integrated circuits is never 100 %, which leaves room for small, transistor-level faults to occur. [2] As the size of a bootROM increases, the probability of such a fault corrupting an instruction word from the bootROM also grows. If no mechanism to bypass usage of the bootROM is in place, such a fault may also critically compromise the chip.

Due to the schedule and resources of our research project, respins of Ballast are not possible. This makes the boot process of the chip extremely important, as the chip must be functional enough to be evaluated and guide development of subsequent chips. This context serves as motivation to implement multiple alternative boot modes that allow for bypassing the bootROM and utilizing different combinations of hardware resources to boot the chip. This research focuses on resiliency through redundancy, while security features, performance and remote boot by network are out of scope. This paper presents the following contributions:

- a resilient design that implements multiple independent boot alternatives utilizing different hardware resources,
- an analysis of developing a large boot program to be implemented as a bootROM,
- the reproducible and generally applicable workflow that we used for developing and verifying the boot code for the SoC and
- the evaluation of our design on fabricated ASIC samples.

This paper is structured as follows: Section II explores the related work on existing implementations of RISC-V SoC

This work is a part of the SoC-Hub project and received funding from Business Finland.

[1] www.sochub.fi

boot. Section III gives an overview of the whole MPSoC that hosts the bootstrap processor (BSP) that this work is centered on. Section IV presents the proposed boot process design, while Section V reviews the boot code implementation of the base RISC-V platform, outlines the requirements for general bootROM development and presents our bootROM implementation. Section VI explores the methodologies used to evaluate and verify the correctness of our design and presents the results of ASIC sample testing. Finally, Section VII recaps the results and contributions of this paper and outlines directions for future work.

II. Related Work

The approaches open source SoCs take to the boot process range from minimal implementations to elaborate and specialized routines. PULPissimo [3], an edge computing oriented SoC, supports three implementations of its boot process. A file I/O based bootROM module can be used for simulation and allows for changes to the boot code without recompilation of the hardware. An infinite CPU execution loop is targeted at field programmable gate array (FPGA) implementations that rely on control from an external debugger. Finally, PULPissimo also provides a full boot code written in C along with scripts to format the compiled executable and linkable format (ELF) file into a SystemVerilog (SV) memory array.

Another example is Rocket-Chip [4] that is a Chisel [5] based RISC-V platform generator. It provides a simple bootROM written exclusively in assembly language that is used to jump to random access memory (RAM) upon reset for further software execution.

Most of the related work centered specifically on the boot process is focusing on secure boot protocols. Haj-Yahya et al. [6] implemented a lightweight secure boot architecture with a focus on software authentication and the integration of a code authentication unit into a RISC-V SoC. Simililarly, Dave, Banerjee and Patel [7] presented a lightweight secure boot architecture for RISC-V that featured a hardware accelerated code integrity and authentication unit. Both systems were evaluated on an FPGA platform and compared against state-of-the-art secure boot solutions.

Kumar et al. [8] presented another security oriented RISC-V SoC prototype, *ITUS*, along with implementation details on their design of the system boot process. Their bootROM implemented a minimal zero-stage bootloader with basic system information and a jump to on-chip block RAM (BRAM) or external memory to execute further bootloader software.

Our work implements a highly resilient and fault tolerant system design, which is an unexplored research direction to the best of our knowledge, and thus differentiates our approach from the related work.

III. System Overview

The MPSoC developed as a part of this research is a heterogeneous system targeted at edge computing applications. The IPs of the system are mostly complete open source blocks or generated with open source toolchains.

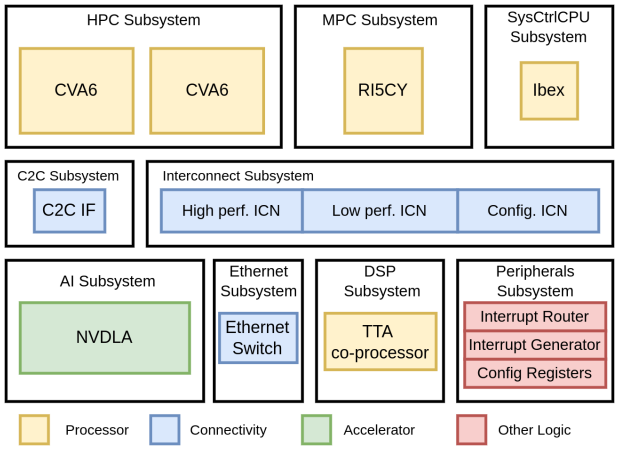

Fig. 1. The high-level architecture of the Ballast MPSoC.

A simplified block diagram of Ballast is presented in Figure 1. The system contains two 32-bit RISC-V processor subsystems based on the PULPissimo microcontroller [3], a 64-bit RISC-V processor subsystem based on the CVA6 processor [9], the open source NVIDIA deep learning accelerator (NVDLA) [10] and a transport triggered architecture (TTA) based digital signal processor (DSP) subsystem designed using the TTA co-design environment (TCE) [11]. The 64-bit dual core subsystem is labeled as the high performance computing (HPC) subsystem and provides application class computing capability to the system, while the DSP subsystem functions as a co-processor to the RISC-V based subsystems. One of the PULPissimo subsystems is left to the default design configuration with the CV32E40P processor core [12] and labeled as the medium performance computing (MPC) subsystem, while the other is configured to use the Ibex [13] processor core. The second PULPissimo subsystem is reduced in terms of memory and peripheral interfaces and is designated as the system control CPU subsystem (SysCtrlCPU) that functions as the BSP for the system. In addition to the processor subsystems, the system has a chip-to-chip (C2C) interface, a central interconnect subsystem, an Ethernet subsystem and a peripherals subsystem.

IV. Boot Strategies

The core functionality of our boot code is loading a software image from an external memory to the SysCtrlCPU static RAM (SRAM) and executing the image. With this minimal functionality in our bootROM, the responsibility of the system level boot is left to a subsequent bootloader that is contained within the loaded software image. The bootROM contains only the control flow and necessary initializations to access a connected SD card in both secure digital I/O (SDIO) [14] and serial peripheral interface (SPI) mode. To improve the reliability of this system through high redundancy in the boot process, four distinct boot modes were conceived.

The **SDIO boot mode** is the default boot mode for the system. It uses the SDIO protocol to access an SD card from a custom SDIO register interface, which is connected directly to the subsystem peripheral bus. In this mode the bootROM

initializes the SD card to store the contained software bootloader image to internal SRAM and move execution to the bootloader.

The **SPI boot mode** is similar to the SDIO boot, but accesses the SD card through the SPI. This change requires an additional program section in the boot code to perform the initialization and accesses to the SD card through the direct memory access (DMA) which connects the SPI interface to the peripheral bus.

The **external boot mode** is enabled by a hardware finite state machine (FSM) in the SDIO register interface with the ability to initialize a connected SD card and autonomously translate bus accesses to the SDIO memory space into SD protocol compliant read and write operations. This makes the SD card a continuous and directly accessible memory space for the CPU. With this configuration the CPU can be set to execute programs directly from the SD card without loading them to internal memory.

The **JTAG boot mode** is the final supported boot alternative. It uses the JTAG interface to connect a debug host to the RISC-V debug module [15] implemented in the SysCtrlCPU. In this mode the debug module has control over the halting and resuming the CPU execution as well as access to the subsystem SRAM via the peripheral bus. To execute programs, the debug module halts the CPU and fills the SRAM before releasing the CPU execution again. While this boot mode does not utilize the SDIO register interface, the SPI interface, the DMA or the bootROM, it is not autonomous due to the dependence on a debug host.

The use of hardware resources on the SysCtrlCPU in each of the four boot modes is illustrated in Figure 2. The CPU and peripheral bus form the core of every boot mode, but all other resources can be bypassed with at least one other mode. One of the three peripheral interfaces being accessible should enable booting the system, and a dysfunctional bootROM, for example, should be bypassable with the JTAG and external boot.

The four boot alternatives were conceived to be completely contained within the subsystem and therefore rely on a minimal number of hardware components to operate. A boot option using Ethernet, for example, was not implemented for this reason, as it would depend on multiple subsystems on the MPSoC to operate.

V. Development

In this section we review the default boot process of the PULPissimo and contrast it against our research goals. We outline the minimum requirements to consider when creating a hardware description of a bootROM from a compiled executable. Finally, we present the structure of our bootROM implementation.

The original boot code of the PULPissimo is formed by a C source code file, multiple platform specific header files and a minimal runtime written in assembly language. As of now, the PULPissimo boot supports four boot modes configured through two bits. The four boot modes are a

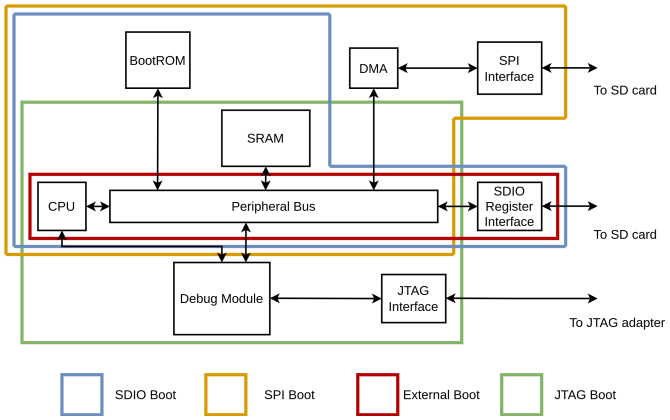

Fig. 2. Usage of the hardware resources of the SysCtrlCPU in each boot mode.

zforth shell accessed through UART, a JTAG mode accessed through an external debugger, a QSPI based flash read and a preloaded boot, which assumes the SRAM to be loaded with an executable image. The size of the compiled PULPissimo bootROM code is approximately 5 kB [3]. Notably, of the four default boot modes only the flash read can be considered completely autonomous.

A fully autonomous boot was one of the key targets in our work, which along with the heavy deviation of the SysCtrlCPU from the original PULPissimo motivated us to create a novel boot concept tailored to our platform and specific needs. While the crt0 runtime, the linker script and the compilation script from ELF to SV were reused with minimal modifications, the main functionality of the new boot code was developed from scratch.

The functionality of the bootROM code can largely be developed like any conventional bare metal program. However, certain restrictions that stem from the unique system state and execution setting will have to be considered, including runtime initalization, linking and special compilation. As the bootROM will be hardcoded on the ASIC and consume silicon area, minimizing its size is a universal goal in its development.

A. Runtime Initialization

While most of the boot code can be written in a high-level compiled language such as C, C++ or Rust, the immediate start of the boot code must perform the initialization of the main abstractions of the platform programming model. Typically, these include setting up an interrupt vector table, initializing the stack pointer and finally, calling the main program entry point [16]. These tasks are conventionally implemented by a platform-dependent file called C-runtime zero, contained in a crt0.S file. The file is automatically inserted by most C-compiler toolchains unless the contrary is specified with a flag such as GCC's `-nostartfiles`. The default start file supplied by compilers is most often excessively elaborate for a boot program and a custom start file might be preferable to minimize code size and thus save silicon area.

B. Linkage

Linking is the process of mapping the program sections of compiled object files to system memory resources to produce an executable program. Compiler toolchains typically use a generic default linker script unless specified otherwise, but the implementation of a bootROM requires a custom linker script to manage the unique memory environment. This becomes apparent when considering the four most common output sections [17]:

- **.text** contains the executable program code and must be mapped to the ROM.
- **.rodata** contains read-only (R/O) data and must be mapped to the ROM.
- **.data** contains read-write (R/W) initialized data. The requirement to write to this section means it can not be mapped to ROM and must be allocated space from a writable memory on the system.
- **.bss** contains R/W zero initialized data and must also be mapped to a writable memory area.

C. Compilation

The source code files are compiled together with the custom crt0 runtime and linker script to create an ELF-executable from the boot code. The compilation flags should be set to optimize for code size. In GCC, this is set with the `-Os` flag. [18]

After compilation [19], the relevant sections from the ELF file are formatted into an SV module using the `objcopy`-utility and scripts adapted from PULPissimo [3].

D. Results

The developed boot code has a size of approximately 3 kB when compiled to an SV memory array, making it significantly smaller than the 5 kB on PULPissimo. Figure 3 illustrates the relative sizes of of the program sections from different source files and their offset from the bootROM base address. The spi.c source file accounts for appproximately 55 % of the bootROM's size, followed by 29 % for sdio.c, 11 % for bootrom.S and 5 % for crt0.S. The C source files contain the SD card specific initalization protocols for each operating mode, while bootrom.S mainly operates the control flow of the boot code and crt0.S is the minimal runtime for the program. The large size of the C based sections is explained by the complexity of the implemented SD initialization protocols. Furthermore, the size of the SPI section in contrast to the SDIO section stems from the requirement to program the SPI mode via DMA commands, while the SDIO protocol has hardware support from the programmable SDIO register interface.

The structure and functionality of the bootrom.S source file is illustrated in Listing 1. By default, the code is structured to automatically run through all initializations supported by software until the image is successfully loaded. If the image can not be loaded, the boot code defaults to an infinite loop to leave the processor in a stable state for the debug module to seize. With this implementation the debug module should be usable even if other boot modes fail, unless the chip completely faulty. From lines 2 to 5, the I/Os are initialized

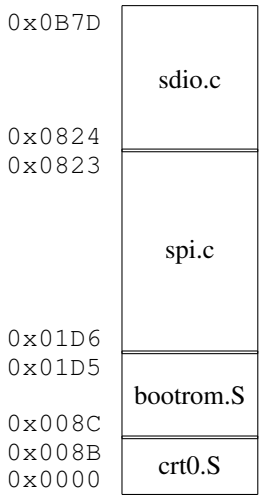

Fig. 3. The offsets of the boot code program sections from the base address 0x1A00 0000.

```
1  int main(){
2    set_schmitt_trigger( PADCFG_08_11 );
3    set_pin( GPIO9, INPUT );
4    set_padmux( PADMUX_MASK );
5    GPIO_ENABLE |= GPIO_MASK;
6    int gpio_value = PADIN_00_31;
7    if (GPIO9_MASK & gpio_value) loop();
8    if (CSN1_MASK & gpio_value){
9      spi_load();
10     if( check_control_word() )
11       goto SW_IMAGE_ENTRY;
12     else loop();
13   }
14   else{
15     sdio_read();
16     if( check_control_word() )
17       goto SW_IMAGE_ENTRY;
18     else{
19       sdio_init();
20       sdio_read();
21       if( check_control_word() )
22         goto SW_IMAGE_ENTRY;
23       else{
24         spi_load();
25         if( check_control_word() )
26           goto SW_IMAGE_ENTRY;
27         else loop();
28       }
29     }
30   }
31 }
```

Listing 1. C-pseudocode describing the behaviour of bootrom.S. Memory addresses, registers and bit masks are abstracted with macros.

to use GPIO9 and CSN1 as inputs, after which their value is read on line 6. The pin values are evaluated on lines 7 and 8, where a high value will call the `loop` or `spi_load` function, respectively. `loop` is a function to implement an infinite loop. `spi_load` performs both the card initialization and read from the SPI interface, while the SDIO interface performs these operations separately with the `sdio_init` and `sdio_read` functions. The `check_control_word` function is used to evaluate a predefined control word in the loaded software image to verify the correctness of the transferred image. The program first attemps to directly read from the SDIO interface on line 15, as the hardware FSM should have initialized the card before software execution. The program then attempts to reinitialize the card on the SDIO interface on line 19 before reading again on line 20. Finally, the program attempts to load the image through SPI on line 24 and enters the busy loop if this is not successful either. The result of each read attempt is verified with a call of the `check_control_word` function. If a read is successful, the program performs an unconditional jump to the fixed image start address, represented with the `goto SW_IMAGE_ENTRY` command.

VI. Evaluation

The BootROM will be unchangeable after fabrication. That makes its careful and exhaustive verification much more critical than for other firmware. For our goals most boot modes are deemed functional in verification if they can transfer an arbitrary software image to the SRAM and execute that image successfully. The external boot mode is the only exception, because it runs software directly from external memory.

To achieve maximal confidence in our design before it is manufactured, we evaluated our design with instruction set simulation, register transfer level simulation and gate-level simulation along with prototyping on two different FPGA boards.

A. Simulation

The boot code was simulated on a number of abstraction levels and methodologies to test its behaviour exhaustively before the design was deployed to FPGA boards for prototyping.

Instruction set simulation (ISS) was used as a low-overhead method to test the simplest functionality of the boot code and check it for runtime errors that are undetectable at compile time. Due to its focus on abstracting away hardware, ISS is of relatively low value in bootROM development where most code deals with the interactions between the CPU and the system's memory mapped peripheral interfaces. With ISS we were able to check the boot code for simple runtime errors and verify the base boot code functionality of reading and writing to and from a directly mapped memory.

Register transfer level (RTL) simulation was used for most of the simulation work during the boot code development. RTL simulation gives a cycle accurate description of the entire system and can be augmented with tracing tools [20] to help software development using processor level abstraction. With RTL simulation we were able to completely simulate the behaviour of the JTAG and external boot options, as well as test the control flow of the bootROM. A simple SD card model allowed us to simulate the functionality of the SDIO boot mode, while no model was available to simulate the SPI mode of an SD card.

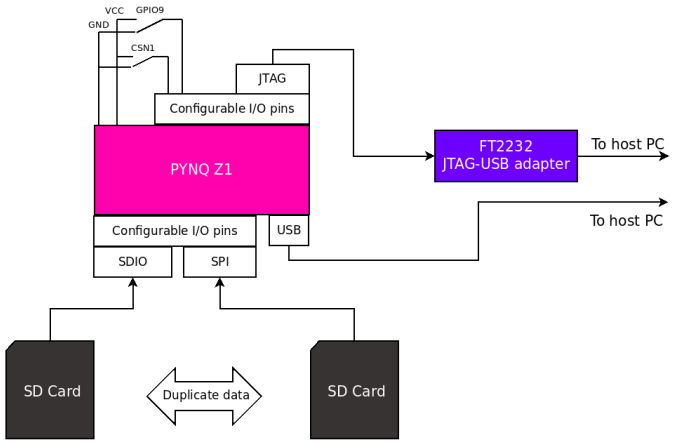

Fig. 4. The FPGA prototyping setup for the PYNQ-Z1 board.

Gate-level simulation (GLS) or netlist simulation was reserved for the end of the development cycle, as the simulation is significantly slower than RTL simulation and the design should have reached a reasonable level of maturity before deploying GLS. With GLS we were able to replicate the results achieved in RTL simulation and thus confirm the functional correctness of the bootROM was not compromised during ASIC synthesis and achieved high confidence in our boot concept and implementation.

B. Prototyping

Prototyping of the boot CPU on physical hardware is an extremely important step in verification as it allows for the system to be tested with the same real physical interfaces as the final ASIC. In this work the prototyping was started after RTL simulation had been mostly covered and was performed on two FPGA boards: the Xilinx PYNQ-Z1 and the Xilinx Zynq Ultra-Scale+ MPSoC ZCU104 Evaluation Kit for the subsystem level and system level, respectively.

The FPGA setup for the PYNQ-Z1 is presented in Figure 4. The SysCtrlCPU device under test (DUT) and its interfaces are implemented fully on the programmable logic of the board. In the prototyping setup the SDIO and SPI interfaces are connected to two SD cards with duplicated data, while on the final PCB one card will be accessed by both interfaces. The option to internally multiplex both interfaces to the same pins during the boot process was considered, but was left unimplemented to avoid the associated increase in complexity. The JTAG pins of the DUT are connected to an FT2232 JTAG-USB adapter to access the RISC-V debug module from a host PC via OpenOCD [21]. The GPIO9 and CSN1 pins are by default configured as inputs and are used in the boot control flow. In the FPGA setup they are controlled by connecting them to VCC or GND with jumpers. The setup on

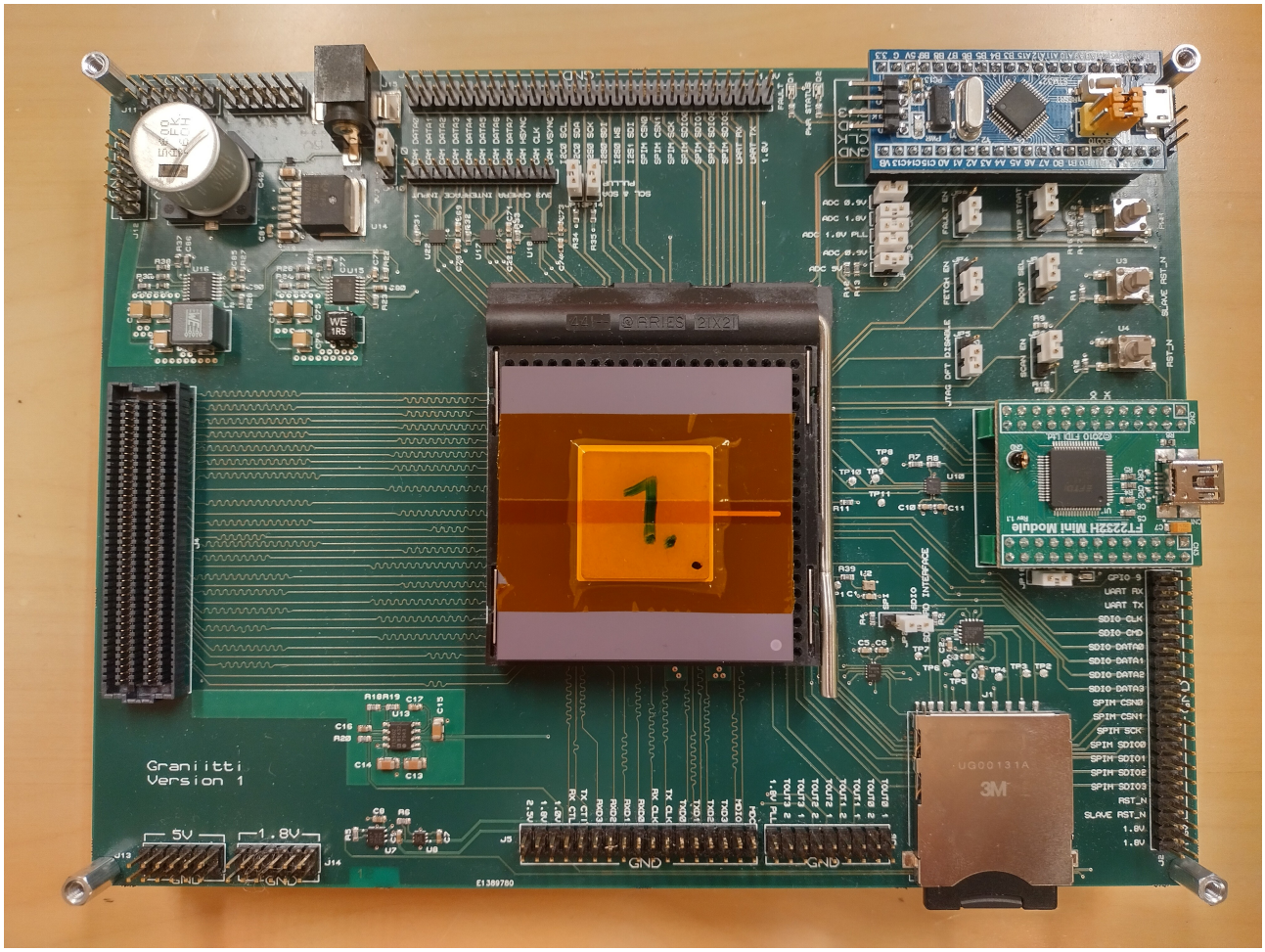

Fig. 5. The Ballast ASIC on the custom circuit board.

the PYNQ-Z1 was closely mirrored on the ZCU104 board to accommodate the full-sized system prototype.

The importance of FPGA prototyping for both development and verification in this work was amplified by the limited availability of full simulation models for SD cards. With the limited visibility on the FPGA during the boot process and the time of synthesis limiting the flexibility of making changes to the bootROM, a new approach was adapted to implement the SDIO and SPI initialization protocols. The SysCtrlCPU was implemented first with a blank bootROM to allow for the debug module to seize control without the system state being affected. The initialization protocols were then developed as conventional software that could be executed with the debug module in conjunction with GDB [22], giving full control and visibility into the system while having a negligible recompilation time. After the software could access the SD card successfully on this level, the sdio.c and spi.c files could be included in the bootROM compilation and their initialization functions called from bootrom.S during the boot process. This approach requires the designer to be especially mindful of the memory space being used, as software on the SRAM could easily overwrite itself. Even with the bootROM in its own memory space, the stack it reserves is located on the SRAM and must not be overwritten when loading the next stage bootloader from the SD card.

FPGA prototyping enabled us to verify all boot modes with the physical SD cards and interfaces that will be used to wake up the final ASIC.

C. ASIC Sample Evaluation

A custom printed circuit board (PCB) was developed to evaluate the Ballast ASIC samples upon their arrival. The PCB including the ASIC is pictured in Figure 5. The Ballast ASIC is packaged in a ceramic pin grid array (CPGA) package and is hosted in the central socket of the PCB. The board has a microcontroller dedicated to power control and monitoring and an FT2232H mini module for JTAG-to-USB connectivity. The SD card on the board can be routed to the SDIO or SPI pins, or be left unconnected with the use of the JP2 jumper. Other jumpers are also implemented to further control the system, including the boot select signal that enables the external boot mode.

The functional testing of the boot process on the ASIC was started after the chip and PCB were tested for their electrical

integrity. Functional testing was started with establishing a JTAG connection, which was successful and allowed us to gain visibility and control of the three RISC-V subsystems. After program execution via JTAG was verified, the SD card based boot alternatives were tested. The SPI boot alternative proved to be functional and reliable immediately, while the SDIO boot was initially dysfunctional. Investigation of this revealed that a very small 4 pF capacitance needed to be added between the `sdio_cmd` line and ground for the SDIO boot to work. The external boot was tested with very simple programs and achieved satisfactory results. The usage of the external boot will be limited due to its very slow execution speed.

VII. Conclusion

This paper introduced Ballast, our novel MPSoC ASIC design and the context in which it was developed. We presented four distinct boot modes based on independent hardware resources to make our design highly resilient to a number of potential faults originating from design or fabrication. The core functionality of our boot process is minimal to retain flexibility and general purpose usability in our design. We outlined the essential considerations for writing software for a bootROM and evaluated our design with ISS, RTL simulation, GLS and FPGA prototyping. Our design achieved the desired results in verification and could be sent to manufacturing with confidence. We evaluated the ASIC samples and were able replicate the results achieved in simulation and prototyping.

Future work with Ballast is centered on developing the software bootloaders and full applications that are loaded during the bootROM execution. As the general concept of the boot process has now been proven on ASIC, multiple directions exist to develop the SysCtrlCPU further. Secure boot features that were omitted from this research would be a natural direction for future development to make the system more comparable with other academic works and commercial solutions. The efficiency and utility of the different boot modes should be evaluated critically for potential savings in system area or to further improve reliability.

References

[1] P. Zhang, *Advanced Industrial Control Technology*. Elsevier Inc, 2010, pp. 601–611.

[2] C. Chiang, *Design for Manufacturability and Yield for Nano-Scale CMOS*, ser. Series on integrated circuits and systems. Springer, 2007, ISBN: 1-280-93802-1.

[3] P. D. Schiavone, D. Rossi, A. Pullini, *et al.*, "Quentin: An Ultra-Low-Power PULPissimo SoC in 22nm FDX," in *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, 2018, pp. 1–3. DOI: 10.1109/S3S.2018.8640145.

[4] CHIPS Alliance. "Rocket-chip." (2022), [Online]. Available: https://github.com/chipsalliance/rocket-chip.git (visited on 03/01/2022).

[5] J. Bachrach, H. Vo, B. Richards, *et al.*, "Chisel: Constructing hardware in a Scala embedded language," in *DAC Design Automation Conference 2012*, 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[6] J. Haj-Yahya, M. M. Wong, V. Pudi, S. Bhasin, and A. Chattopadhyay, "Lightweight Secure-Boot Architecture for RISC-V System-on-Chip," in *20th International Symposium on Quality Electronic Design (ISQED)*, 2019, pp. 216–223. DOI: 10.1109/ISQED.2019.8697657.

[7] A. Dave, N. Banerjee, and C. Patel, "CARE: Lightweight Attack Resilient Secure Boot Architecture with Onboard Recovery for RISC-V based SOC," in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, 2021, pp. 516–521. DOI: 10.1109/ISQED51717.2021.9424322.

[8] V. B. Y. Kumar, S. Deb, N. Gupta, *et al.*, "Towards Designing a Secure RISC-V System-on-Chip: ITUS," eng, *Journal of Hardware and Systems Security*, vol. 4, no. 4, pp. 329–342, 2020, ISSN: 2509-3428.

[9] F. Zaruba and L. 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," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019, ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2926114.

[10] NVIDIA Corporation. "Hardware Manual." (2018), [Online]. Available: http://nvdla.org/hw/contents.html (visited on 02/15/2022).

[11] P. Jääskeläinen, T. Viitanen, J. Takala, and H. Berg, "HW/SW Co-design Toolset for Customization of Exposed Datapath Processors," in *Computing Platforms for Software-Defined Radio*, W. Hussain, J. Nurmi, J. Isoaho, and F. Garzia, Eds. Springer International Publishing, 2017, pp. 147–164, ISBN: 978-3-319-49679-5. DOI: 10.1007/978-3-319-49679-5_8. [Online]. Available: https://doi.org/10.1007/978-3-319-49679-5_8.

[12] A. Traber, M. Gautschi, and P. Schiavone. "RISCY: User Manual." (2019), [Online]. Available: https://pulp-platform.org/docs/ri5cy_user_manual.pdf (visited on 04/11/2022).

[13] P. D. Schiavone, F. Conti, D. Rossi, *et al.*, "Slow and steady wins the race? a comparison of ultra-low-power RISC-V cores for Internet-of-Things applications," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1–8. DOI: 10.1109/PATMOS.2017.8106976.

[14] SD Association. "Simplified Specifications." (2018), [Online]. Available: https://www.sdcard.org/downloads/pls/ (visited on 02/21/2022).

[15] RISC-V. "RISC-V External Debug Support." (2019), [Online]. Available: https://riscv.org/wp-content/uploads/2019/03/riscv-debug-release.pdf (visited on 02/21/2022).

[16] Embecosm. "The C Runtime Initialization." (2010), [Online]. Available: https://www.embecosm.com/appnotes/ean9/html/ch05s02.html (visited on 02/23/2022).

[17] S. Chamberlain and I. Taylor. "The GNU Linker." (2021), [Online]. Available: https://sourceware.org/binutils/docs-2.37/ld.pdf (visited on 03/02/2022).

[18] R. Stallman. "Using the GNU Compiler Collection." (2021), [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-11.2.0/gcc.pdf (visited on 03/17/2022).

[19] RISC-V Software Collaboration. "RISC-V GNU Compiler Toolchain." (2022), [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain (visited on 03/02/2022).

[20] lowRISC. "Ibex Reference Guide: Tracer." (2022), [Online]. Available: https://ibex-core.readthedocs.io/en/latest/03_reference/tracer.html (visited on 03/04/2022).

[21] "riscv-openocd." (2022), [Online]. Available: https://github.com/riscv/riscv-openocd (visited on 04/12/2022).

[22] R. Stallman, R. Pesch, and S. Shelbs. "Debugging with GDB, The GNU Source-Level Debugger." (2022), [Online]. Available: https://sourceware.org/gdb/current/onlinedocs/gdb.pdf (visited on 03/09/2022).