Triple-Modular-Redundancy RISC-V Demonstrator - documentation

Antmicro Ltd

2018-04-27

CONTENTS

CHAPTER

ONE

INTRODUCTION

The purpose of this document is to describe the motivation for as well as the structure and use of the Triple-Modular-Redundancy system, a demonstrator of the RISC-V technology stack prepared by Antmicro for Thales Research and Technology.

The Triple-Modular-Redundancy RISC-V demonstrator (henceforth *RISC-V demonstrator* or *demonstrator*) has been designed to show how the open RISC-V ISA architecture specification, and an open source implementation thereof, can be used to build a flexible and extendible fault tolerant voter CPU system mitigating Single Event Upset (SEU) with minimum impact on software.

1.1 Motivation

While the use case covered by the demonstrator is a practical one, and similar solutions are often implemented for space applications, the purpose of the demonstrator reaches beyond just solving the technical challenge of restoring state in a fault-inducing environment. The primary focus is rather to show how RISC-V as such and all the building blocks in its technology stack can be combined to achieve:

- maximum design freedom
- higher speed of implementation
- higher degree of reuse
- more thorough understanding of the system

The demonstrator project employes several technologies which have made technical innovations like this possible:

- *RISC-V* (the major focus here)
- *Rocket Chip*
- *Chisel*
- *UltraScale+* FPGA SoC
- *Linux*
- *Zephyr*

1.1.1 RISC-V

RISC-V, being an open specification backed by over 100-members of the Foundation driving it (which includes companies like Google, NVIDIA, Antmicro, Western Digital, Qualcomm, Microsemi, Samsung, and Berkeley amongst its Platinum members), has generated a radically different ecosystem than any other competing proprietary ISAs or even previous open source efforts.

As an open spec with a strong legal and institutional foundation, coming out of world's leading institution in the field (Berkeley), co-designed by the creator of the RISC-V (Dave Patterson) and already adopted by Silicon Valley titans like NVIDIA and Western Digital, RISC-V combines the rallying power of open source and open specifications with a full readiness for industrial adoption.

Thanks to a sharing and inclusive ecosystem, RISC-V attracts best minds in the research and industry and opens new fields where ASIC design was normally not an option. With the computer architecture classic student handbooks updated to use RISC-V in the new edition, students are being taught RISC-V as a de-facto standard worldwide, and it is predicted that it will be the default choice for new designs, especially in niche applications.

1.1.2 Rocket Chip

Coming from UC Berkeley, the open source Rocket Chip Generator is framework allowing the user to generate a highly-parametrisable family of CPUs, including both 32, 64 and 128-bit solutions for various applications. This is one of the earliest and most mature large-scale RISC-V implementations, and the 32-bit chip coming out of the Rocket generator is SiFive's FE310, the first commercially available RISC-V chip and the base for Antmicro's RISC-V SoM.

The Rocket Chip Generator is written in the *Chisel* HDL and is an off the shelf CPU generator solution which can be adapted for specific purposes, saving years of design time.

Rocket Chip is Apache-licensed and is under active development.

1.1.3 Chisel

Chisel (Constructing Hardware in a Scala Embedded Language) is a high-productivity, Scala-based Hardware Description Language (HDL) developed in UC Berkeley for more structured development of FPGA and ASIC IP (it is a project that was developed in parallel to, but is not obligatory for, RISC-V). Chisel compiles to Verilog, which makes it easy to use other traditional tools in the ASIC/FPGA designer's workflow, but thanks to Scala's high-level and dynamic capabilities it makes the activity of developing hardware blocks much simpler and more predictable.

Many of the creators of RISC-V underline that without Chisel, RISC-V as we know it today would not have been possible. They explain the relation between Chisel and Verilog through an analogy to the relation between C and assembly - while it is possible to write very complex programs in assembly, over time the vast majority of real life programs have been using C which provides better portability and productivity through a higher level of abstraction. Similarly, the tedious process of writing Verilog can be avoided in Chisel and a much more concise and less repetitve syntax can be used instead, where the chip/FPGA designer can focus on what are the unique characteristics of any given hardware block and abstract out/parametrize the rest.

Chisel is a different approach from HLS (High-Level Synthesis) - the latter typically uses syntax of C/C++ with macros which is familiar to programmers yet difficult to map on HDL (resulting in problems whenever tight control over the functioning of the IP implemented in HLS is needed, or indeed whenever the HLS solution needs to be debugged), while the former is a domain-specific language (DSL) newly designed specifically for the purpose of hardware development and using the advantages of its base, Scala, only opportunistically.

Traditional HLS approaches have been focusing on creating blocks which accelerate specific functionalities but not on entire blocks and systems, while Chisel is a much more general purpose concept, and allows very tight control over the IP design and the flexibility of a higher-level programming language at the same time.

Antmicro is one of the earliest adopters of Chisel, which has indeed resulted in a productivity boost and much more concise, understandable code in many of our IP designs, leading to fewer bugs and less testing. While there is a learning curve involved, the time is almost immediately saved when building any more complex design. With its adoption in a large, commercially used CPU IP project - *Rocket Chip* - Chisel is very unlikely to fade away as a HDL development language, but since it compiles to Verilog and can be freely interchanged with other HDL languages like Verilog or VHDL, there is also little risk involved in using it.

1.1.4 Linux

A de-facto standard OS for the majority of embedded and server systems on the market (including Xilinx FPGA SoC based systems), Linux is used on the UltraScale+ CPU system to allow for a wide range of open source development and debugging tools to be taken advantage of. This choice is especially useful for development purposes, and many Linux utilities and capabilities were helpful especially in the intermediate phases of the project while building up the system. Since Linux is run both on the UltraScale+ CPU and the development host, a high degree of compatibility in terms of tooling, scripring and usage can be provided.

Antmicro has very deep expertise in Linux on Xilinx Zynq/UltraScale+ since it has already developed an entire Linux Board Support Package stack for multiple UltraScale+ projects.

1.1.5 Zephyr

Zephyr is a Linux Foundation-driven project building a scalable real-time operating system (RTOS) for a broad range of architectures and embedded and IoT applications. It is optimized for resource constrained devices, and built with security in mind.

Unlike the non-RT OS space where Linux is the dominant choice, the RTOS space has traditionally been very fragmented with no clear winner - the Zephyr project, with backing from companies like Intel, NXP, Nordic Semiconductor and TI as well as the widely recognized and effective Linux Foundation, is aiming to provide a good de-facto standard choice for an open source RTOS.

RISC-V is a primary citizen in the Zephyr Project - Antmicro is officially maintaining the RISC-V port of Zephyr. There is a significant commitment to maintaining the new architecture from both the Zephyr team and Antmicro, as is is viewed to be a very good match for the open source and collaborative nature of RISC-V itself.

Recent additions to Zephyr include symmetric multiprocessor support (SMP), asymmetric multiprocessor support (AMP), POSIX-compliance and other industry-friendly features. Pre-certification in many important fields is a major focus for 2018.

1.1.6 UltraScale+

The FPGA SoC device class has emerged as a de-facto standard platform to custom architecture research: a tightly integrated CPU and FPGA with highly customizable I/O and the possibility to interchangeably use hard Processing System blocks or soft Programmable Logic IP is a perfect fit to building software-driven but hardware-accelerated solutions. With its successful Zynq FPGA SoC series, Xilinx is currently the leading provider of FPGA SoC solutions - almost universally used by companies prototyping with RISC-V - and UltraScale+ is its state-of-the-art platform. Antmicro is one of the few parties in the world working with both UltraScale+ and its predecessor, Zynq-7000 since a very early stage, collaborating with one of the leading FPGA SoC SoM vendors, Enclustra, for whom (and whose customers) we have built the entire Linux software stack for a wide array of UltraScale+ and other FPGA SoC modules.

UltraScale+, with a quad-core Cortex-A53 processing system and dual-core Cortex-R on the processing side, as well as a built-in GPU, is a powerful hardware platform that can serve as a fully capable embedded computer, offering the capability to connect fast networking, a screen, and indeed almost any I/O interface. Antmicro's UltraScale+ Processing Module used in the project, is one of the few boards on the market offering both a wide array of built-in connectors and a high number of fast I/O pins which can be used for customization/prototyping.

A large amount of FPGA logic (469,000 LUT-4 equivalents - with a possiblity to easily upgrade to 747,000) is available, and while in its current state the project is far from using the entire FPGA fabric, a large amount of logic cells is almost always required as soon as specialized processing tasks need to be prototyped, and a multi-core Linux-capable solution is potentially planned for the future iterations of the project - which task can only be achieved with a platform like UltraScale+.

Throughout the project, the built-in Cortex-A53 CPU of the UltraScale+ Processing Module running Linux was heavily used to prototype new features, greatly reducing development time - the solution could be implemented in steps and tested every time a new element was moved into the FPGA.

1.2 Summary

A wide array of technologies is employed in the demonstrator project to make the Triple-Modular-Redundancy possible to achieve in a relatively short period of time and build a future-proof development and training platform.

RISC-V lies at the base of the solution, enabling the customized CPU demonstrator to be managed with off-the-shelf, open source tools developed by the entire RISC-V ecosystem. *Rocket Chip* is used to develop the RISC-V cores without having to build them from scratch but rather focusing on the relevant functionality. *Chisel* is used for high-productivity IP development and easy-to-explain code. *Zephyr* is used on the redundant cores as a standard RTOS, to provide a software stack that is significantly more complicated than a bare-metal program but also realistic in a potential real-life application. *Linux* on Antmicro's *UltraScale+* Processing Module is used to build a very robust and future-proof development platform with lots of FPGA logic, I/O and tooling available.

CHAPTER

TWO

OVERVIEW

2.1 System structure

The block diagram of the demonstrator is presented in Fig. 2.1 below.

In the center of the system there are 3 RISC-V cores working in parallel and ensuring triple modular redundancy.

They are connected to the system bus through the Fault Detector (Voter), which merges the AXI buses from the respective cores into one. The main objective of the Fault Detector (Voter) is to detect any inconsistencies in the bus communication between all cores and the system bus and responding to them.

The Fault Injector & System Monitor and System Manager are logical blocks realized with a set of IP cores jointly called the System Monitor, and are controlled by the quad-core ARM Cortex-A53 CPU running Linux. With its ability to inspect the status of all other parts of the system it can be used for monitoring and debugging purposes.

2.2 RISC-V CPU

The main part of the demonstrator system is the RISC-V Fault Tolerant CPU. In fact the CPU is composed of three identical CPUs clocked with synchronized clock and running the same program.

The RISC-V FT CPU can access the external peripherals with one system bus. The bus is monitored with the Fault Detector IP core. The IP core is responsible for detecting any differences in the operation of the three instances of the RISC-V CPU.

The RISC-V FT CPU can access two peripherals on the system bus: DDR memory and UART.

The System monitor IP core can be used to control the execution of the RISC-V FT CPU.

2.2.1 RISC-V configuration

The RISC-V Fault Tolerant CPU is composed of three identical RISC-V CPUs. Each CPU is configured as a 32bit RISC-V IA CPU.

The following, additional components are added to the CPU core:

- Interrupt controller
- RTC clock (used to generate system ticks, clocked with bus clock)

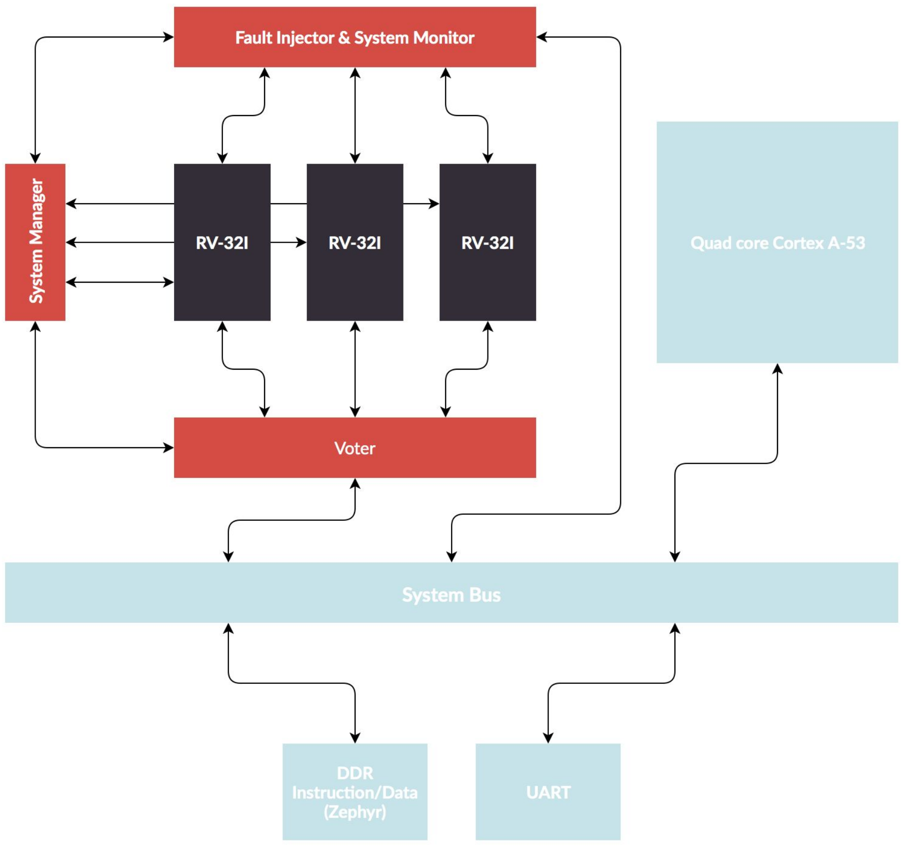

Fig. 2.1: Block diagram of the demonstrator

- JTAG DAP
- Uncached AXI4 main bus
- Uncached additional AXI4 bus used as a recovery bus
- Boot ROM containing zero stage boot loader

On normal reset the CPU starts execution of the code in Boot ROM. The boot loader placed in the Boot ROM does nothing more, but jumps directly to the first address of the DDR memory. The CPU starts executing the code placed in the DDR memory.

2.2.2 RISC-V Fault Tolerant adaptations

Beside the normal RISC-V configuration the Fault Tolerant version has been extended with the following features:

- Possibility to change the reset vector to start the code in the hang section of the boot ROM
- Possibility to change the reset vector to start the code execution in the recovery memory space
- Possibility to observe every general purpose register from an external logic block
- Possibility to observe selected special purpose registers from an external logic block

The reset vector changes are realized with two additional 1 bit inputs to the CPU. Setting the corresponding input to a logic '1' during reset changes the reset vector. Reset to recovery has a higher priority, thus setting this signal to '1' will ignore the settings on the hang line.

The CPU registers are routed from the internal core of the CPU up to the top level of the CPU. The routing includes clock domain crossing between the CPU core and external peripherals domains. The routed out registers are updated immediately when the CPU state changes.

2.2.3 RISC-V memory map

The following table presents the memory map of the RISC-V CPU:

Base Address	Range	Function
0x0	0x1000	Debug Controller
0x10000	0x10000	Boot ROM
0x50000000	0xFFFFFF	Recovery code
0x51000000	0xFFFFFF	CPU Registers copy
0x60000000	0x3FFFFFF	DDR Memory
0x70000000	0x1000	UART

Addresses below `0x50000000` are inside the CPU IPCore. The memory there is read only, and is hardcoded into the CPU during the synthesis.

Addresses between `0x50000000` and `0x52000000` are accessible through the additional recovery bus. This bus is independent in every instance of the RISC-V CPU. The code and data in this memory region is used during the recovery stage.

Addresses above `0x60000000` are accessible through system bus. This bus is shared between all the instances of the RISC-V CPUs. This bus is monitored by the Fault Detector IP core to detect any execution errors.

2.3 System bus

The system bus in an AXI4 bus allowing access to the DDR memory and UART peripherals. As mentioned above, the system bus of the RISC-V Fault Tolerant CPU is shared between all the three instances of the RISC-V CPUs. Separate busses from each RISC-V CPU are connected to the Fault Detector IP core. This IP core is responsible for merging the three separate busses into one instance and monitoring the bus accesses of each of the three CPUs. The detailed description of the Fault Detector IP core can be found in the *Fault Detector* section.

2.4 System monitor

The System monitor is a simple AXI4-Lite IP core enabling the software running on the US+ monitoring the state of the RISC-V CPU. Additionally, the System monitor IP core allows the software running on the US+ can control the execution flow of the RISC-V Fault Tolerant CPU.

The IP core can generate two types of interrupts:

- CPU fault interrupt - generated when a discrepancy between the CPUs is detected.
- CPU back online - generated when a faulty CPU has been recovered and is back online.

The IP core exposes five 32 bit wide AXI4-Lite accessible registers. The following table presents the registers offsets and purpose:

Offset	Register name	Function
0x00	Control	Control of the RISC-V FT CPU
0x04	Status	Current status of the system
0x08	CPU0 PC	Current value of the RISC-V FT CPU0 Program counter
0x0C	CPU1 PC	Current value of the RISC-V FT CPU1 Program counter
0x10	CPU2 PC	Current value of the RISC-V FT CPU2 Program counter

2.4.1 Control Register

This register is used to reset/start/stop the selected RISC-V CPU. The following table presents the functions of the bits in the register (bits not mentioned in the table are not used):

Bit	Function
0	Enable clock for CPU0 (writing ‘1’ enables the clock)
1	Enable clock for CPU1 (writing ‘1’ enables the clock)
2	Enable clock for CPU2 (writing ‘1’ enables the clock)
4	Assert reset for CPU0 (writing ‘1’ asserts the reset, writing ‘0’ deasserts it)
5	Assert reset for CPU1 (writing ‘1’ asserts the reset, writing ‘0’ deasserts it)
6	Assert reset for CPU2 (writing ‘1’ asserts the reset, writing ‘0’ deasserts it)
8	Global interrupt enable
9	CPU fault interrupt enable
10	CPU back online interrupt enable
13:12	JTAG routing control (see section *debug-access*)

2.4.2 Status Register

This register is used to monitor the current status of the RISC-V CPU. The following table presents the functions of the bits in the register (bits not mentioned in the table are not used):

Bit	Function
0	CPU Fault interuptL ‘1’ in this bit means that the fault event occured (writing ‘1’ clears the interrupt)
1	CPU Back online interupt: ‘1’ in this bit means that the faulty CPU has been recovered (writing ‘1’ clears the interrupt)
6:4	Discrepancy detected between the CPUs (see section *Fault Detector*)

2.4.3 CPUX PC

The registers from `0x08` to `0x10` are used purely for debug purposes. The values in those registers are the current values of the program counter of the corresponding RISC-V CPU.

2.5 Execution control

The RISC-V CPUs reset and clock signals are controlled by other system components - System Monitor and Fault Detector IP cores. In order to maintain the FPGA system stability and not produce gated clocks, additional components were included in the design:

- Xilinx Processor system reset IP core.
- Clock control IP core.

Xilinx Processor system reset assures that the reset signal is held long enough to reset all the system components.

Clock control IP core instantiates a `BUFGCE` component. The component allows to enable/disable a clock signal without necessity of routing it through regular (and not clock dedicated) logic. Additionally the IP core provides TCL scripts necessary for the Vivado’s DRC checker.

2.6 RISC-V debug access

For debug purposes of the RISC-V FT CPU special JTAG adapter IP core has been developed and included in the system. The IP is named Simple JTAG and is connected via AXI4-Lite to the US+ processing system. The IP core enables accessing the RISC-V DAP. Two instances of the IP core are included in the system.

Since the single CPU cores of the RISC-V Fault Tolerant CPU can be in different state, a single JTAG adapter cannot be used to access them. An additional IP core called the JTAG Router, whose purpose is to connect the Simple JTAG IP cores to a selected CPU of the RISC-V Fault Tolerant CPU, was created. The IP core is controlled with a two bit `selector` signal. The following table explains the routing depending on the `selector` value:

Selector	Routing
0	JTAG0 is routed to all the CPUs, JTAG1 is disconnected
1	JTAG0 is routed to CPU0, JTAG1 is routed to CPU1 and CPU2
2	JTAG0 is routed to CPU1, JTAG1 is routed to CPU0 and CPU2
3	JTAG0 is routed to CPU2, JTAG1 is routed to CPU0 and CPU1

The JTAG routing allows to debug the code execution in every state of the system:

- When all the CPUs are not corrupted JTAG0 can be used to access the CPU.
- In case of a single CPU fault, JTAG0 is routed to the faulty CPU, while JTAG1 is routed to a correct pair.

2.7 RISC-V recovery memory space

The recovery space of the RISC-V Fault Tolerant CPU can be divided into two parts:

- Memory holding the recovery code.
- Saved registers memory.

In the demonstrator system, the recovery code is held in a block RAM memory. The memory is also accessible from the US+ Processing system, so the recovery software is easily exchangeable. This approach allows to faster development of the recovery software and testing different approaches.

The register memory is an AXI4 IP core which observers all the internal registers routed from the RISC-V CPUs. The IP core can latch the register values and provide the latched values to a CPU in the recovery stage.

CHAPTER

THREE

FAULT DETECTOR

The fault detector is one of the key elements of the system. It's responsible for detecting any inconsistencies on the busses of the RISC-V processors, disconnecting the faulty core from the main bus, and then restoring its state.

The detector consists of an AXI forwarder which basically merges the AXI buses from all RISC-V cores into one main bus that is connected to the Processing System - Zynq UltraScale+ in this case. When one of the cores starts failing, the Fault Detector (also called Voter) indentifies the responsible core, isolates it from the bus and applies a reset without influencing the work of the remaining cores.

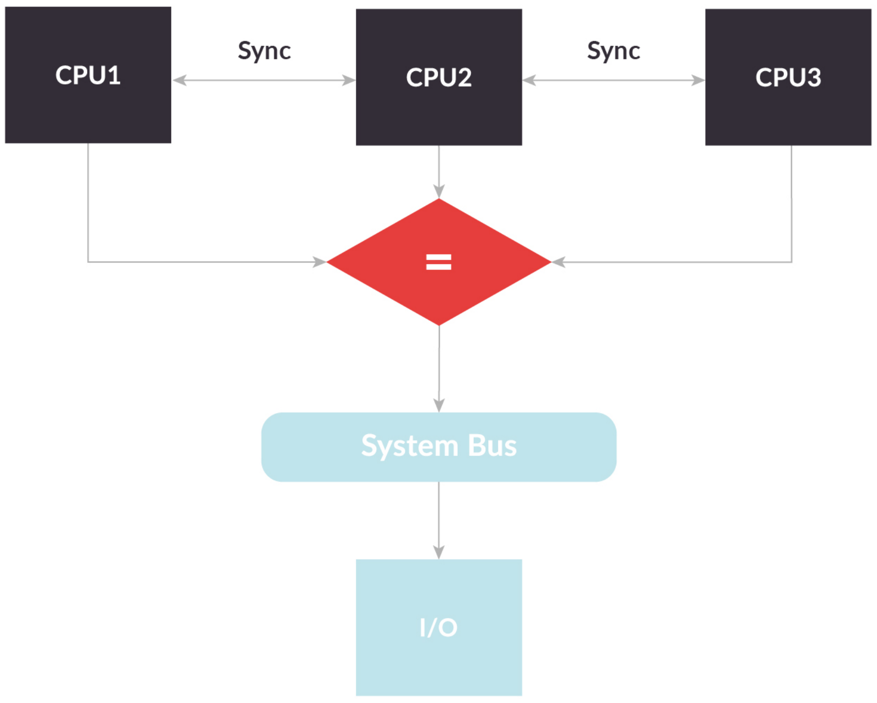

Fig. 3.1: Block diagram of fault detector

3.1 Fault detection

The system bus of all three RISC-V CPUs is routed through the Fault Detector IP core. The IP delays all the bus operations by three clock cycles. This gives enough time to detect a failure and react properly. All the AXI inputs to the Fault Detector IP core are monitored asynchronously.

When any discrepancy is detected, the faulty CPU is immediately detected and the recovery procedure started.

The fault detection is signalized by setting a logical '1' in the 3-bit `invalid` register. Single bits in this register mark CPU pairs which gave different results on the monitored bus. Knowing the non-matching pairs it is an easy task to determine the faulty CPU.

The register is organized in the following way:

- bit 2 - discrepancy detected on CPUs 0 and 1
- bit 1 - discrepancy detected on CPUs 0 and 2
- bit 0 - discrepancy detected on CPUs 1 and 2

In order to get the number of the faulty CPU the `invalid` register has to be negated and translated from one-hot encoding. For example if the CPU0 is the faulty one the discrepancy will be detected on pairs (0, 1) and (0,2) resulting in a `invalid` register seting of `0x6`.

The `invalid` register value triggers the recovery Finite State Machine, starting the recovery process.

3.2 Recovery procedure

Knowing which CPU is the faulty one, the system will attempt the recovery procedure. The procedure can be divided into the following parts (each part is explained in details in the subsections below):

1. Cut off the faulty CPU from the system bus.
2. Ensure the faulty CPU is held in a known state.
3. Ensure the correct CPUs are not in the interrupt state.
4. Stop the correct CPUs.
5. Copy the internal state of the correct CPUs.
6. Reset the bus routing logic.
7. Restore the internal state and start all the CPUs in the same point.

3.2.1 Cutting off the faulty CPU

In order to prevent damaging the correct CPUs by the faulty one, the system cuts off the faulty one from accessing the system bus. All the AXI inputs of the faulty CPU are forced with logic '0'. All the AXI outputs of the faulty CPU are discarded. The system bus signals are generated basing only on the two correct CPUs.

3.2.2 Keeping the faulty CPU in a known state

Since the system cannot predict in which state the CPU was before the failure was detected, we cannot trust that simply cutting it off from the system bus will not disrupt it further.

Further disruptions may put the CPU into a state where recovery will not be possible.

In order to prevent such a situation the faulty CPU is reset, but the reset vector is changed to the hang address within the CPU's boot ROM. This operation ensures that the CPU will not try to access the system bus, and will stay in a known state until the system starts the recovery.

3.2.3 Checking the interrupt state of the correct CPUs

At this stage the recovery system is about to save the internal state of the correct CPUs. The saved state will be used further in the recovery process as a last known-to-be-good state. After the recovery all the CPUs will be restored at this point. The recovery system has to ensure the saved CPUs state was not captured while the CPUs were in the interrupt state. If the CPUs are in the interrupt state, the recovery system waits until they return to normal execution.

3.2.4 Stopping the correct CPUs

Before copying the internal state of the correct CPUs the system must ensure that they are not executing any code. Code execution could change the internal state in the copying moment resulting in situation where part of the copied state is from different moment than the rest. In order to prevent such a situation the system stops the CPUs clocks while copying the internal state.

3.2.5 Copying the internal state

Since the internal state of every RISC-V CPU in the system is routed out, the copying is a simple latching of currently observed signals. The recovery FSM invokes a latch signal causing the internal state to be saved in the flip flops.

3.2.6 Resetting the routing logic

Before restoring the internal state of the CPUs and restarting them, the bus routing logic has to be reset to a state where all the CPUs can access the system bus. Otherwise, a few clock cycles after the restart the Fault Detector IP would again report the CPU as faulty.

3.2.7 Restoring the state of the CPUs

The last step of the recovery process is to restore the internal state of the CPUs and restart them in a last known-to-be-good state. In order to do so, the CPUs are reset, but the reset vector is set to point to a special recovery software. The software is an integral part of each CPU, but for the demonstration purposes it is executed from external FPGA block RAM. Execution of the recovery software does not trigger any transaction on the system bus. The recovery software reads the saved state of the correct CPUs and copies it to a CPU it is running on. The last instruction of the recovery software is setting the program counter (PC) to an address where the CPU state was saved.

CHAPTER

FOUR

SOFTWARE

The software used in the demonstrator can be divided into three parts:

- *Ultrascale+ Processing System software*
- *RISC-V software*
- *FPGA firmware*

These are discussed separately in the respective sections below.

4.1 US+ Processing system software

The software running on the Ultrascale+ Processing System is used mostly for controlling the demonstrator system, and observing the results. The US+ PS quad-core Cortex-A53 CPU runs the Linux operating system. For ease of development the whole device boots from an SD card. The following subsections describe different pieces of the US+ PS software.

4.1.1 Bootloaders

The First Stage Boot Loader (FSBL) and PMU firmware required for creating a Zynq US+ boot image were created from the Vivado project using Xilinx SDK. ARM trusted firmware was compiled from the sources available on Xilinx github.

The second stage bootloader used in the system is the U-Boot bootloader. The base version of the bootloader used in the demonstrator system is a port developed by Antmicro for Enclustra's XU1 SoM. The only modification for the demonstrator system was to limit the RAM memory available for Linux from 2GB to 1.5GB (address range `0x0 - 0x5FFFFFFF`).

4.1.2 Linux kernel

The base version of the Linux kernel used in the demonstrator is a port developed by Antmicro for the Enclustra's XU1 SoM. In order to handle the RISC-V FT system, the `risc-v-ft` driver has been added to the system. The driver handles all the actions required to control the flow of the RISC-V CPU, detect the faults and recoveries, load the firmware and inject faults.

The driver registers itself as a character driver, and exposes a series of IOCTLs to user space. The IOCTLs enables the user space application to control the RISC-V CPUs. The RISC-V firmware can be loaded by simply writing to the `/dev/risc-v-ft` file. The following table presents the available IOCTLs:

IOCTL	Function	Arguments
RISC_V_RESET_CPUS	Assert reset signal for all the RISC-V CPUs and routing logic	None
RISC_V_RELEASE_RESET	Deassert reset signal for all the RISC-V CPUs and routing logic	None
RISC_V_DISABLE_CLKS	Disables clock for all the RISC-V CPUs	None
RISC_V_ENABLE_CLKS	Enables clock for all the RISC-V CPUs	None
RISC_V_BREAK_CPU	Stops and immediately starts the clock for a selected RISC-V CPU	(unsigned int) CPU number
RISC_V_RESET_SINGLE	Asserts and immediately deasserts reset for a selected RISC-V CPU	(unsigned int) CPU number
RISC_V_SET_FIRMWARE_TYPE	Sets the firmware type to be loaded	(unsigned int) Firmware type (0 - normal, 1 - recovery)

4.1.3 Linux user space

The GNU/Linux distribution running on the demonstrator is Arch Linux. A number of additional applications and scripts were developed, or installed in the system:

- `risc-v-ctrl` simple user space application allowing the user to interact with the `risc-v-ft` kernel driver
- `OpenOCD` - open source on chip debugger
- `picocom` - simple serial terminal
- FPGA bitstream loading script

risc-v-ctrl

The basic CLI application allows the user to interact with the kernel driver. The application handles resetting/starting the CPU, loading the firmware and breaking the selected CPU. The application can be controlled with the following switches:

```
-c <val>      - enables/disables the clock for all the CPUs. Possible values 0/1
-r <val>      - asserts/deasserts the reset signal for all the CPUs. Possible values 0/1
-b <cpu>      - breaks the selected CPU
-f <firmware> - loads the main CPU firmware
-u <firmware> - loads the recovery firmware
```

openOCD

The openOCD debugger has been extended with a driver for the SimpleJTAG adapter. Additionally, scripts for connecting to the selected JTAG adapter have been added.

4.2 RISC-V software

Software running on the RISC-V fault tolerant CPU(s) is located in different parts of the memory space and is run depending on the current state of the CPU. In normal operation mode, the CPU runs the Zephyr RTOS. During startup, a bootloader hardcoded in the CPU's boot ROM is executed. While recovering from a fault, special recovery firmware is executed from the recovery memory space.

4.2.1 Zephyr

Zephyr is an open source real-time operating system (RTOS) - for more information about it see *Zephyr*.

In the demonstrator the choice of zephyr was motivated by its complexity - it is much more complicated than a bare metal program, but not as complicated as a fully blown operating system such as Linux.

For proper operation, Zephyr requires periodic interrupts from the system timer.

The RTOS has been ported to the Fault Tolerant demonstrator hardware. The porting required adding a driver for the UART IP core used in the system. The system memory layout had to be adjusted so Zephyr runs fully from the DDR memory.

The application used in the demonstrator simply prints `I'm alive [counter]` line periodically. This is an indicator that the CPU is still operational. Example output of the Zephyr application running on the RISC-V Fault tolerantsystem looks as following:

```
***** BOOTING ZEPHYR OS v1.10.0-rc1 - BUILD: Apr 21 2018 12:43:12 *****
I'm alive! counter = 0
I'm alive! counter = 1
I'm alive! counter = 2
I'm alive! counter = 3
```

4.2.2 BootROM

The zero stage bootloader hardcoded into the CPU's boot ROM used in the demonstrator is the simplest possible bootloader. It contains two sections:

- The first one jumps directly to the DDR base address and executes the code from there.
- The second section, used to hang the CPU in fault mode, keeps the CPU in an infinite loop.

Bellow is the listing of the Bootloader code:

```
.section .text.init
.option norvc
.section .text.start, "ax", @progbits
_start:
csrr a0, mhartid
li t0, DDR_TARGET_ADDR
jr t0

.section .text.hang, "ax", @progbits
.globl _hang
_hang:
```

```
1:
j 1b
```

4.2.3 Recovery

Recovery firmware is executed from an additional recovery memory space, so that the execution of this code does not trigger the system bus transactions. The code simply reads saved registers and restores them on the CPU where it is currently executed. The very last instruction is to jump to the memory address where the CPU had been when the registers were saved.

4.3 FPGA firmware

The FPGA design was developed for Zynq MP Ultrascale+ 6EG using Vivado 2017.4. The design was also evaluated with Vivado 2018.1.

The following IP cores were developed for the system:

- RISC-V 32IA CPU core
- Fault Detector
- System Monitor
- JTAG Router
- Simple JTAG adapter
- Recovery register latch

The description of the IP cores can be found in the *System structure* section.

CHAPTER

FIVE

BUILDING THE SYSTEM

As mentioned in the previous chapter, the system consists of elements that can be categorized into 3 groups: US+ Processing system software, RISC-V software, FPGA firmware. Each of those elements has been developed as a separate subproject and as such has its own git repository. In this chapter we will focus on how to build each part of the entire system.

5.1 US+ Processing system software

5.1.1 Bootloaders

FSBL

In order to build the First Stage Boot Loader (FSBL), the Vivado Software Development Kit is needed. The SDK is provided as part of the Vivado Design Suite. Since building the FSBL requires a hardware description file, which is generated by Vivado during synthesis and implementation of the entire FPGA system, it's necessary to run Vivado first. Once Vivado is done implementing the design we can create the FSBL using the SDK's Hardware Software Interface tool. The steps to generate the FSBL are as follows:

1. Open the demonstrator system in Vivado
2. If you haven't generated the bitstream before, do it now
3. Using Vivado's *export Hardware Desicription* option from the *File* menu save the .hdf file to a known location (this can also be achieved by simply typing `write_hwdef -file demonstrator.hdf` in the tcl console inside Vivado)
4. Open the Hardware Software Interface (hsi) tool in interactive mode and type:

```
generate_app -hw [open_hw_design demonstrator.hdf] -os standalone -proc ps7_
↪cortexa53_0 -app zynqmp_fsbl -compile -sw fsbl -dir fsbl
```

U-Boot

Since we need to cross compile the code for ARM Cortex-A53, we will need a 64-bit ARMv8 Cortex-A cross compiler like Linaro's `aarch64-linux-gnu-gcc-linaro` available on the Linaro project's webpage: https://releases.linaro.org/components/toolchain/gcc-linaro/latest/ Since the base version of the bootloader used in the demonstrator system is a port done for the Enclustra's XU1 SoM, the same configuration can be used. Steps to perform:

1. Change the directory to the `xilinx-uboot` directory
2. Configure the build with:

```
make CROSS_COMPILE="aarch64-none-linux-gnueabi-" ARCH=arm64 enclustra_zynqmp_mercury_
↪xu1_defconfig
```

3. Build U-Boot with:

```
make CROSS_COMPILE="aarch64-none-linux-gnueabi-" ARCH=arm64 -j(nproc)
```

Note: `-j$(nproc)` will parallelize the build using all threads available in the system, use `-jX` to provide a number manually in place of `X`, e.g. `-j8`.

5.1.2 Linux kernel and risc-v-ft module

The steps to build the Linux kernel are very similar to those for U-Boot:

1. Set up the kernel build process with the `xilinx_zynqmp` configuration:

```
make CROSS_COMPILE="aarch64-none-linux-gnueabi-" ARCH=arm64 xilinx_zynqmp_defconfig
```

2. Build the kernel:

```
make CROSS_COMPILE="aarch64-none-linux-gnueabi-" ARCH=arm64 Image -j$(nproc)
```

(see note about the `-j` option in the U-Boot section above)

The `Image` file will be available in `arch/arm64/boot/` upon the completion of the compilation process.

3. The `risc-v-ft` driver module source code is stored in the `drivers/misc/antmicro` directory. In the configuration from the previous step it will be compiled along with all the Linux kernel modules once the following command is used:

```
make CROSS_COMPILE="aarch64-none-linux-gnueabi-" ARCH=arm64 modules -j$(nproc)
```

(see note about the `-j` option in the U-Boot section above)

5.1.3 Linux user space applications

risc-v-ctrl

To cross compile this simple C app a 32-bit ARMv7 Cortex-A Linux targeted toolchain is required, such as Linaro's arm-linux-gnueabihf available at: https://releases.linaro.org/components/toolchain/binaries/latest/arm-linux-gnueabihf/. The steps to compile the application:

1. Change the directory to the one containing the source file - `thales-risc-v-controller`
2. Add the path containing the aforementioned gcc compiler to your system `PATH` variable
3. Execute `make`

OpenOCD

To compile this application we will use the same compiler as in case of the `risc-v-ctrl` application. Follow these steps:

1. Change the directory to `riscv-openocd`
2. Call the bootstrap script: `./bootstrap.sh`
3. Use the following configuration to build for arm with support for the SimpleLink fpga jtag ip-core:

```
./configure \
--disable-verbose-usb-io \
--disable-verbose-usb-comms \
--disable-dummy          \
--disable-libusb         \
--disable-use-libusb0    \
--disable-use-libusb1    \
--disable-ftdi           \
--disable-jlink          \
--disable-stlink         \
--disable-ti-icdi        \
--disable-usb-blaster-2  \
--disable-vsllink        \
--disable-osbdm          \
--disable-ulink          \
--disable-opendous       \
--disable-aice           \
--disable-usbprog        \
--disable-rlink          \
--disable-armjtagew      \
--disable-cmsis-dap      \
--disable-parport        \
--disable-parport-giveio \
--disable-legacy-ft2232_libftdi \
--disable-legacy-ft2232_ftd2xx \
--disable-jtag_vpi      \
--disable-usb_blaster_libftdi \
--disable-usb_blaster_ftd2xx \
--disable-amtjtagaccel   \
--disable-zy1000-master  \
--disable-zy1000         \
--disable-ioutil         \
--disable-ep93xx         \
--disable-at91rm9200     \
--disable-bcm2835gpio    \
--disable-gw16012        \
--disable-presto_libftdi \
--disable-presto_ftd2xx  \
--disable-openjtag_ftd2xx \
--disable-openjtag_ftdi  \
--disable-oocd_trace     \
--disable-buspirate      \
--disable-minidriver-dummy \
--disable-remote-bitbang \
--disable-internal-libjaylink \
--disable-sysfsgpio \
--enable-verbose \
--enable-verbose-jtag-io \
--host=arm-none-linux-gnueabi \
--enable-simplelink
```

4. Execute `make`

The resulting binary is in `/src/openocd`.

5.2 FPGA bitstream and IP cores

5.2.1 Bitstream

To build the bitstream the Vivado Design Suite is required. Version 2017.4 was used for the demonstrator project. In order to generate the bitstream the following steps must be completed:

1. Change the working directory to `thales-risc-v-vivado`:

```
cd thales-risc-v-vivado
```

2. Source the Vivado settings script to have all necessary binaries added to your system path:

```
source <vivado_installation_path>/settings64.sh
```

3. Generate Vivado project build script:

```
./git/tools/generate_project.sh > project.tcl
```

4. Generate the Vivado project file and the bitstream:

```
vivado -mode batch -source project.tcl
```

The generated bitsream will be placed in the following location: `project_1/project_1.runs/impl_1/design_1_wrapper.bit`

Once the bitstream is ready we need to generate an image file with it so that we can load it to the FPGA PL (programmable logic) from Linux running on the US+:

1. Create a temporary directory in the location of your choice and change your working directory to it:

```
mkdir /tmp/risc-v/ && cs /tmp/risc-v/
```

2. Create a new file called `bitstream.bif` with the structure of the bitstream as below inside the new location:

```
echo "all: { fpga.bit }" > bitstream.bif
```

3. Source the vivado settings file as in the bistream generation steps if you haven't done it so far

4. Generate the image file:

```
bootgen -image bitstream.bif -arch zynqmp -o fpga.bit.bin
```

5.2.2 IP cores written in Chisel

Apart from for the RISC-V cores, Chisel was also used to create several other parts of the system. These cores need to be compiled to Verilog before they can be added to the project in Vivado.

The steps to generate Verilog and add them to the Vivado project are very similar. The table below lists the names of the cores written in Chisel. For each of them the procedure is as follows:

1. Change the working directory to the corresponding directory name from the table
2. Invoke `make verilog`
3. Copy the generated verilog file named as in the corresponding verilog file name to the Vivado project into the location provided in the following table:

Name	Directory	Verilog file	Vivado project directory inside thales-risc-v-vivado/git/
Fault detector	thales-risc-v-fault-detector	FaultDetector.v	ip_repo/fault_detector_1.0/src/
JTAG Router	thales-risc-v-jtag-router	JTAGRouter.v	src/hdl/
Registers Router	thales-risc-v-registers-router	RegsRouter.v	ip_repo/regs_router_1.0/src/

CHAPTER

SIX

DEPLOYMENT

In order to run the demonstrator on the Enclustra XU1 module we need to prepare an SD card with all files necessary for booting the Linux system on US+, loading the bitstream to the Programmable Logic (PL) and testing the RISC-V fault tolerant system.

6.1 Preparing the SD card

The SD card will hold two partitions:

1. FAT with the kernel image, device tree, `boot.bin` and U-Boot script used for booting Linux on the US+
2. ext4 with an ARM Arch Linux distribution rootfs with all the demonstrator's binaries and scripts

6.1.1 Formatting the SD card

The command-line based `fdisk` tool can be used to create partitions on the SD card. We are assuming that a new SD card with no partitions will be used:

1. Insert the SD card into the SD card connector
2. Find the device name of the card with `lsblk` and unmount it with:

   ```
   umount /dev/<device_name>
   ```

3. Invoke fdisk:

   ```
   fdisk /dev/<device_name>
   ```

4. Creare a new partition by typing `n`. Make it primary by selecting `p`, use the default partition number and the first sector. Set the size of the partition to 1G for this partition by typing `+1G`
5. Make this partition bootable. Set the bootable flag by typing `a`. Now if you print out the partition table, you should see your new partition with a * under `Boot`.
6. Create the root partition by typing `n`. Select the primary partition, use the proposed first and last sector.
7. If you check your partition table now, you should see the 2 partitions you just created. If the result is as expected, use `w` to write to disk and exit.

8. Format the boot partition to FAT by calling:

```
mkfs.vfat -F 32 -n boot /dev/<device_name><partition_no_1>
```

9. Format the rootfs parition to ext4 by calling:

```
mkfs.ext4 -L root /dev/<device_name><partition_no_2>
```

6.1.2 Boot files

The boot partition needs several files in order for the system to boot. Therefore the following files need to be copied to the boot partition:

- `boot.bin` - bootimage with the first stage bootloader (FSBL), second stage bootloader (U-Boot), ARM trusted firmware and Power Management Unit (PMU)
- `Image` - Linux kernel image
- `devicetree.dtb` - Devicetree of the system
- `uboot.scr` - U-Boot boot script

6.1.3 Rootfs - Arch Linux

The demonstrator runs on the Arch Linux ARM distribution. To copy the rootfs to the SD card with a newly prepared ext4 partition simply extract the provided archive by typing:

```
tar -zxf <name_of_the_archive> -C <path_to_the_rootfs_partition_on_the_sd_card>
```

6.2 Running the demonstrator

Once the SD card is created, Linux and the demonstrator software can be run.

6.2.1 Booting Linux

1. Connect a PC with USB cable to the USB Debug connector on the board
2. Open a terminal and connect to the `/dev/ttyUSB0` serial device using a terminal emulation program like picocom with the baudrate of 115200, e.g. `picocom /dev/ttyUSB0 -b115200`
3. Power on the module

The module boots Linux from the SD card. To login use the following credentials:

```
user: root
password: root
```

Note: It's generally advisable to use ssh instead of the serial interface. The main advantage is that several ssh sessions can be established to the same device at once. This siginificantly improves watching the demonstrator run. To setup a ssh connection just open another terminal and invoke ssh root@<IP_address_of_the_module>. The password is the same as in the Linux credentials.

6.2.2 Loading the driver and bitstream

Before the execution of the firmware on the RISCV cores can be controlled the bitstream with the system needs to be loaded to the PL (Programmable Logic) and the Linux driver to communicate with it has to be loaded:

1. Load the driver module by invoking `insmod risc-v-ft.ko`
2. Upload the bistream with the provided load-fpga.sh script: invoke `./load-fpga.sh`

6.2.3 Uploading zephyr and recovery firmware

Loading the binaries to the memory can be done using the `risc-v-ctrl` application. In order to be able to watch the output of the firmware that is about to be loaded on the terminal, establish an ssh session to the US+ and then use `picocom` to attach to the UART serial device:

1. Establish the ssh connection with:

   ```
   ssh root@<IP_address_of_the_module>
   ```

   and type in the password
2. Use picocom to connect to the device `/dev/ttyUL0` with baudrate of 9600:

   ```
   picocom /dev/ttyUL0 -b9600
   ```

3. Leave the recently created terminal open for monitoring purposes, create a new one and establish another ssh session.
4. Upload the recovery firmware - in the terminal created in the previous step invoke:

   ```
   ./risc-v-ctrl -u recovery.rv32.img -c 1 -r 0
   ```

5. Upload the Zephyr binary by invoking:

   ```
   ./risc-v-ctrl -f zephyr.bin -c 1 -r 0
   ```

In the terminal connected to the UART device on US+ we should already see zephyr running.

6.2.4 Demonstrator in action

At this point the `risc-v-ctrl` application can be used to control the behavior of the system.

Resetting the system

To reset the system use:

```
./risc-v-ctrl -r 1 -c 1
```

You can see how the application stops writing to the terminal.

To deassert the reset to the system run:

```
./risc-v-ctrl -r 0 -c 1
```

The application should now be restarted.

Stopping the system

To stop the clock on the RISCV cores, call:

```
./risc-v-ctrl -r 0 -c 0
```

You can see how the application stops writing to the terminal.

To restart the clock, call:

```
./risc-v-ctrl -r 0 -c 1
```

The application should continue.

Breaking selected cores and recovery

To break a selected RISCV CPU, call:

```
./risc-v-ctrl -b <number_of_the_CPU>
```

On the debug terminal (the one that was used to connect to the serial device) you will be able to observe messages about which CPU was broken and whether it was recovered. If the core is recovered we shouldn't see any interruptions in the terminal showing the output from Zephyr.