RADPC@SCALE: AN APPROACH TO MITIGATE SINGLE EVENT UPSETS IN THE MEMORY OF SPACE COMPUTERS

by

Justin Patrick Williams

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Electrical Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

May 2022

ACKNOWLEDGEMENTS

This work was supported by the NASA Flight Opportunities Program under CAN# 80NSSC20K0107.

TABLE OF CONTENTS

TABLE OF CONTENTS – CONTINUED

LIST OF TABLES

LIST OF FIGURES

Figure Page

LIST OF FIGURES – CONTINUED

Figure Page

ABSTRACT

This thesis presents the flight test results of a single event upset (SEU) mitigation strategy for computer data memory. This memory fault mitigation strategy is part of a larger effort to build a radiation tolerant computing system using commercial-off-the-shelf (COTS) field programmable gate arrays (FPGAs) called RadPC. While previous iterations of RadPC used FPGA block RAM (BRAM) for its data memory, the specific component of RadPC that is presented in this paper is a novel external memory scheme with accompanying systems that can detect, and correct faults that occur in the proposed data memory of the computer while allowing the computer to continue foreground operation. A prototype implementation of this memory protection scheme was flown on a Raven Aerostar Thunderhead high-altitude balloon system in July of 2021. This flight carried the experiment to an altitude of 75,000 feet for 50 hours allowing the memory in the experiment to be bombarded with ionizing radiation without being attenuated by the majority of Earth's atmosphere. This thesis discusses the details of the fault mitigation strategy, the design-of-experiments for the flight demonstration, and the results from the flight data. This thesis may be of interest to engineers that are designing flight computer systems that will be exposed to ionizing radiation and are looking for a lower cost SEU mitigation strategy compared to existing radiation-hardened solutions.

1

INTRODUCTION

The demand for computers that can continue operation in space environments has increased with the rapid rise of space exploration missions. One of the leading causes of harmful effects to computers in space is cosmic radiation. While terrestrial computers are protected from ionizing radiation by the Earth's atmosphere and magnetic field, space computers must be designed to operate in the presence of radiation that can cause material degradation and potential crashes [3].

Taking into consideration the classification of the effect is important when implementing systems targeting resilience to the effects of radiation. The two primary classifications of space radiation are total ionizing does (TID) and single event effects (SEEs) — both of which decrease the computer system's reliability [7]. A TID effect's root cause is from low-level energy particles depositing charge within a device's insulating regions. This leads to a gradual breakdown of the insulating material over time and can lead to unwanted current flow and permanent transistor biasing. This results in excess power draw and overall degradation of the device itself. The complication of the TID is that its effects begin emerging over time and are irreversible [24]. Figure 1.1 outlines how TID and an SEE will affect a complementary metal-oxide-semiconductor (CMOS) device — the most widely used transistor technology in modern computers since the 1980s [6]. The TID causes damage to the material over longer periods of time, where an SEE can trigger variety of faults that are discussed in the following paragraph.

A CMOS device experiences an SEE when high-energy particles strike the device, introducing excess charge in the semiconductor material, and leading to inadvertent logic level shifts. SEEs are divided into three subcategories of faults that occur from these logic-

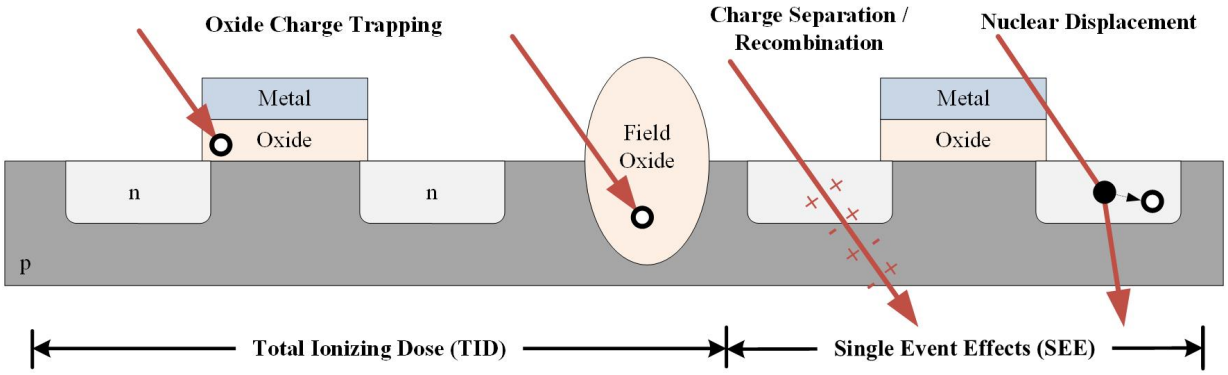


Figure 1.1: CMOS TID Cross Section

level transitions, each of which cause varying levels of harm to the device. The first branch of SEEs can be classified as a single event transient (SET), which occurs when ionized particles deposit charge onto a region of the device and cause unwanted voltage pulses. The second branch of SEEs can be classified as a single event upset (SEU), which is when an SET is captured within a storage device such as a D-flip-flop or memory cell. A single event functional interrupt (SEFI) is when either an SET or SEU occur and introduce a fault that cannot be repaired by conventional recovery strategies. An SEFI could cause a system fault such as a system reset.

Montana State University has spent the past decade developing a reconfigurable computing system, denoted "RadPC," that can detect the effects of SEE-induced faults and respond with a suite of recovery mechanisms [18]. RadPC is a novel SEE mitigation strategy utilizing COTS components and is resilient to SEEs [18, 19]. The need for increased computing in space is becoming largely apparent as sensors and big data become more prevalent in space applications [11].

For the mission discussed in this thesis, the Flight Provider that was provisioned was Raven Aerostar. Raven has locations across the country and are a leading pioneer in high-altitude balloon flights. The flight vehicle that has been provisioned for this experiment is the

Raven Thunderhead, utilizing Iridium satellite communication capabilities to allow customer payloads to send telemetry from their ground-station to their payloads during flight.

In preparation for the lunar mission, the experiment in this paper serves to validate the RadPC@Scale platform with its external memory architecture in harsh environments through the means of a high-altitude balloon flight. This flight targets the system-wide test of the RadPC architecture, error detection, error recovery, error logging, as well as an external memory system, all of which will increase the NASA technology readiness level (TRL) of the RadPC computer architecture. High altitude ballooning serves as a cost-effective way to verify the flight-readiness of a space computer [14].

4

MOTIVATION

The RadPC platform is designed to be a cost-effective space computing solution by implementing its architecture on commercial-off-the-shelf (COTS) components. To achieve resilience to cosmic radiation using COTS grade parts, the design itself integrates radiation mitigation technology directly into the architecture. By using COTS products, the platform avoids the use of expensive components that are manufactured for radiation tolerance that tend to be cost-prohibitive for commercial missions.

A traditional method for mitigating radiation effects on space computing is shielding. This is achieved by designing an enclosure with external metals of varying thicknesses that meet design tradeoffs, including overall system mass versus radiation resilience [29]. A study utilizing numerical analysis on shielding found that as the shielding thickness is increased, the shielding effects are also increased in terms of protection against radiation [4]. The study also came to find that the phenomenon known as scattering also increases.

Other widely used techniques for mitigating radiation effects are implemented at the fabrication level, including radiation hardening by process (RHBP) and radiation hardened by design (RHBD). Both RHBP and RHBD share a common goal of mitigating excess charge in the semiconductor material when a radiation strike occurs. RHBP decreases the probability of a strike depositing charge within the semiconductor gate by altering the underlying semiconductor materials [18, 2]. RHBD achieves a similar end through fabricating components with non-standard layout geometries that attempt to reroute the resulting charge from a radiation strike into the power supply nodes of the circuitry [18, 20]. While both RHBP and RHBD prove to be effective in the mitigation of the harmful effects of TID and SEEs, they are prohibitively expensive for most commercial space missions.

In recent years, the smaller processing nodes of CMOS transistors have resulted in feature sizes (<65nm) that are less susceptible to TID effects due to the reduced probability

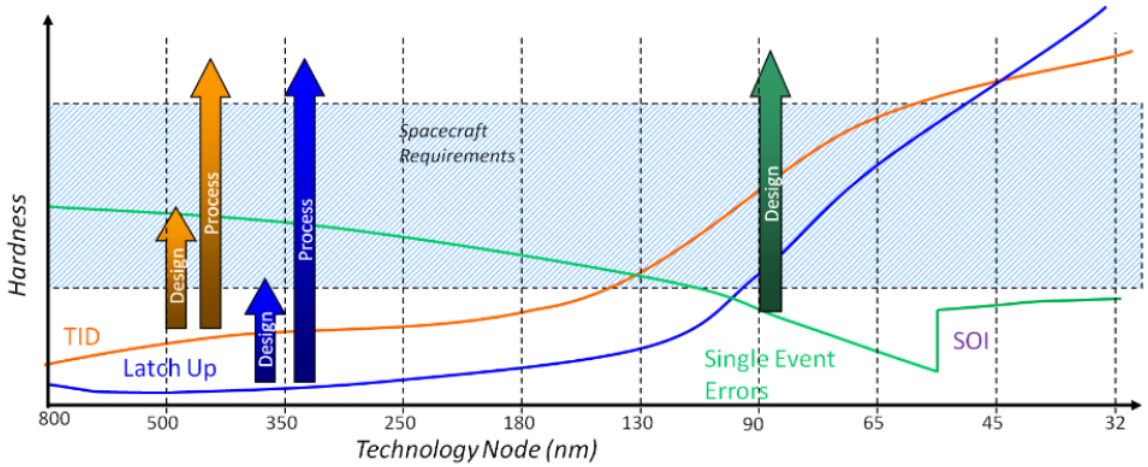


Figure 2.1: Varying level of radiation hardness relative to feature node sizes [1]

of charge getting trapped in the thin insulating regions. In a workshop on radiation hardening by design, a presentation from Boeing provided insight on the relationship between transistor feature sizes and the resilience to radiation effects — Figure 2.1 quantifies the reduction in TID effects on transistors with node sizes less than 65nm [1].

Simultaneously the susceptibility of these smaller transistors to SEEs has increased due to the reduced amount of energy needed to cause a logic level transition. Charles Slayman performed a study on soft error trends and mitigation techniques on CMOS memory devices. Slayman's research targeted mitigation techniques for static random access memory (SRAM) and dynamic random acces memory (DRAM) topologies, both are primarily used in modern computing and CPU architecture. Slayman found that as feature sizes decrease in SRAM, the quantity of SEEs increase; the effects of this are shown in Figure 2.2. The scaling on the y-axis is FIT/chip, where a failure-in-time (FIT) is defined as an event occurring once every 100 centuries. This figure was generated by analyzing the number of soft errors occurring on SRAM in New York City caused by high energy neutron upsets. While Figure 2.2 is recording the number of faults relative to Earth's atmosphere, SEEs are becoming the primary concern for future space missions that use modern semiconductor materials [26].

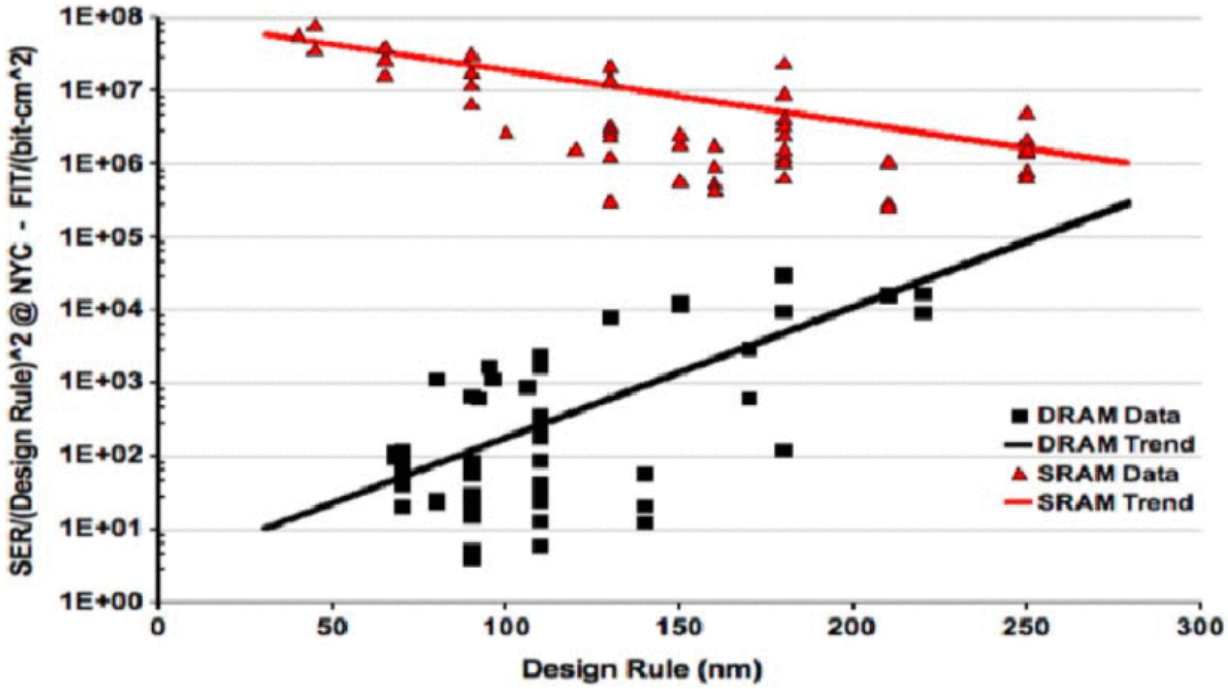


Figure 2.2: Soft Error Trend Per Chip Vs Design Rule [26]

The increased susceptibility of SEE on SRAM devices has led to an increased challenge in the aerospace environment [16]. This model provides insight to how SRAM will behave in a space environment receiving strikes from alpha particles over that of high energy neutron particles in Earth's magnetosphere.

Given the rapid increase of space exploration missions being performed, the need for cost-efficient radiation resilient computing has become a top priority. Montana State University seeks to meet this goal with RadPC@Scale. The fault recovery procedure is part of a larger effort at Montana State University to develop a radiation tolerant computer technology for use in space. The RadPC computer has matured over the past 12 years through a variety of flight demonstrations on various sub-systems. This balloon flight specifically focused on detecting and correcting errors in external data memory for RadPC.

7

RELATED WORK

There are existing FPGA based radiation tolerant computer systems that use similar design methodologies for mitigating cosmic radiation SEEs. This chapter presents a small survey showcasing various FPGA based computers, and if the information is available, the specific FPGA IC's that are being used and missions that the computers are targeting.

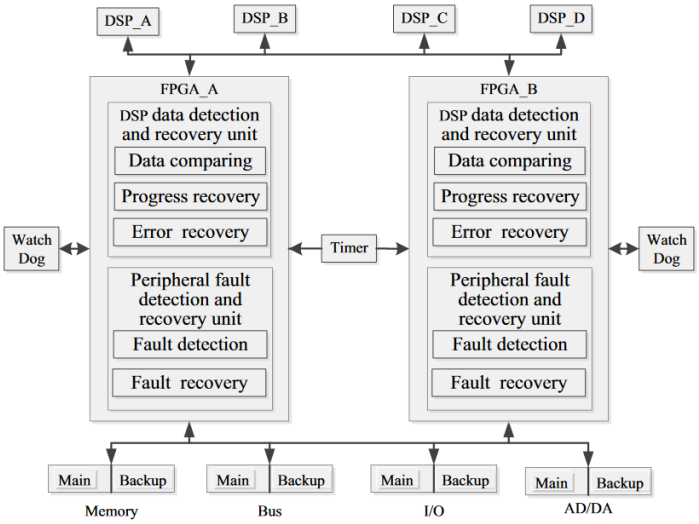
Many proposed FPGA based SBC's targeting space missions are implementing COTS based FPGA chips — specifically due to the fact that COTS FPGAs are cheap and highly reconfigurable. Zhen Dong proposed a radiation tolerant SBC implementing two separate FPGA chips, and 3 DSP chips configured in a 'Parallel + Backup' method [10]. In this architecture, the FPGAs are implementing the system control unit in a dual parallel capacity. The DSPs are configured in a TMR fashion with one additional DSP IC as a backup. If a DSP is faulted, the FPGA-based control unit will force the faulted DSP to restart while pulling from the backup DSP to take its place during operation and the main DSP to restart.

Bhargav Shashidhara published research on their implementation of a COTS FPGA based SBC that incorporates a TMR processor system with voted outputs [25]. The voter is a state machine that monitors the health of the system and identifies a faulted soft-core processor. The system additionally has an interrupt controller that monitors and arbitrates between the outputs and voter signals. The system also incorporates an interrupt controller block that holds all systems until the faulted tile is repaired. The FPGA and toolchain used for this design was a Xilinx Nexys 4 DDR ARM/FPGA SoC with Vivado 2017 Design Suite.

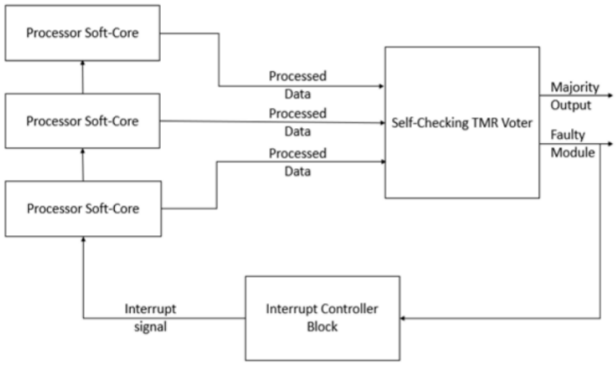
Christian Fuchs published research on their implementation of a COTS FPGA based SBC with a suite of fault recovery mechanisms. The system implements four forms of external memory: FeRAM to store OS code, MRAM to store application code, NAND flash to store telemetry, and DDR for main memory on the computer system [12]. It implements four tiles in an NMR configuration with an ARM Cortex-A53 application processor as its

designated softcore. Radiation recovery mechanisms include a voter subsystem, DRAM memory scrubber, partial reconfiguration and soft-error-mitigation to monitor configuration memory. At the time of this paper, this architecture was implemented as a proof of concept on a Xilinx XCKU5P FPGA.

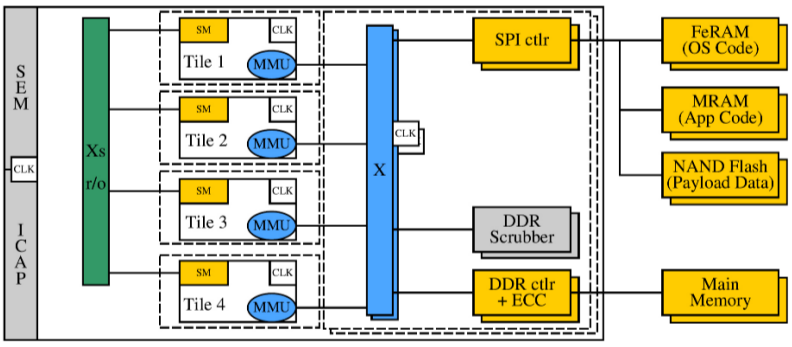
The system architecture for Dong's, Shashidhara's, and Fuchs' FPGA based SBCs is shown in Figure 3.1a, 3.1b, and 3.1c, respectively. This shows a glimpse in what research is actively being performed on implementing FPGA based SBC's that target LEO and deep space missions. There is a major effort in the development of these computers, and RadPC@Scale is a contender within this field. The architecture of the RadPC and RadPC@Scale follows a similar design methodology to that of Fuch's, Shashidhara's, and Dong's implementations and design architectures.



(a) Dong's System Architecture Overview [10]



(b) Shashidhara's System Architecture Overview [25]



(c) Fuchs System Architecture Design Overview [12]

Figure 3.1: ECC Hamming Correction Memory Encoder and Decoder Entity Diagrams

10

DESIGN OF EXPERIMENTS

The design of the experiment must include testing and verification of the external data memory system and RadPC@Scale's recovery procedures. The use of a high altitude balloon experiment allows us to expose the RadPC@Scale computer to a radiation rich environment which will allow natural faults to occur. The system includes a microcontroller for instrumentation such that the computer can accumulate the occurrences of a natural radiation fault as well as whether or not the computer recovered.

Mission Requirements

This section describes the criteria for mission success. The overall mission requirements break down into three subcategories: Raven success requirements, payload requirements, and ground support requirements. The Raven requirements were presented to us by our system integration engineers, detailing what would be considered a successful mission flying MSU's payloads. The payload and ground support requirements were imposed by the team at MSU. Raven's requirements stem from the capabilities of the Thunderhead high-altitude balloon system, as well as its flight-control-unit's interface to the payloads.

The top-level mission requirements is shown in Figure 4.1. The technology and programmatic requirements shown in Figure 4.1 are not focused on this thesis. They are specific to NASA for reporting/rating our mission success.

Raven Requirements

The Raven requirements are seen in the yellow column of Figure 4.1 denoted *Raven Requirements (R)*. Each of these requirements outline how the payloads must interface to the FCU for operation during flight.

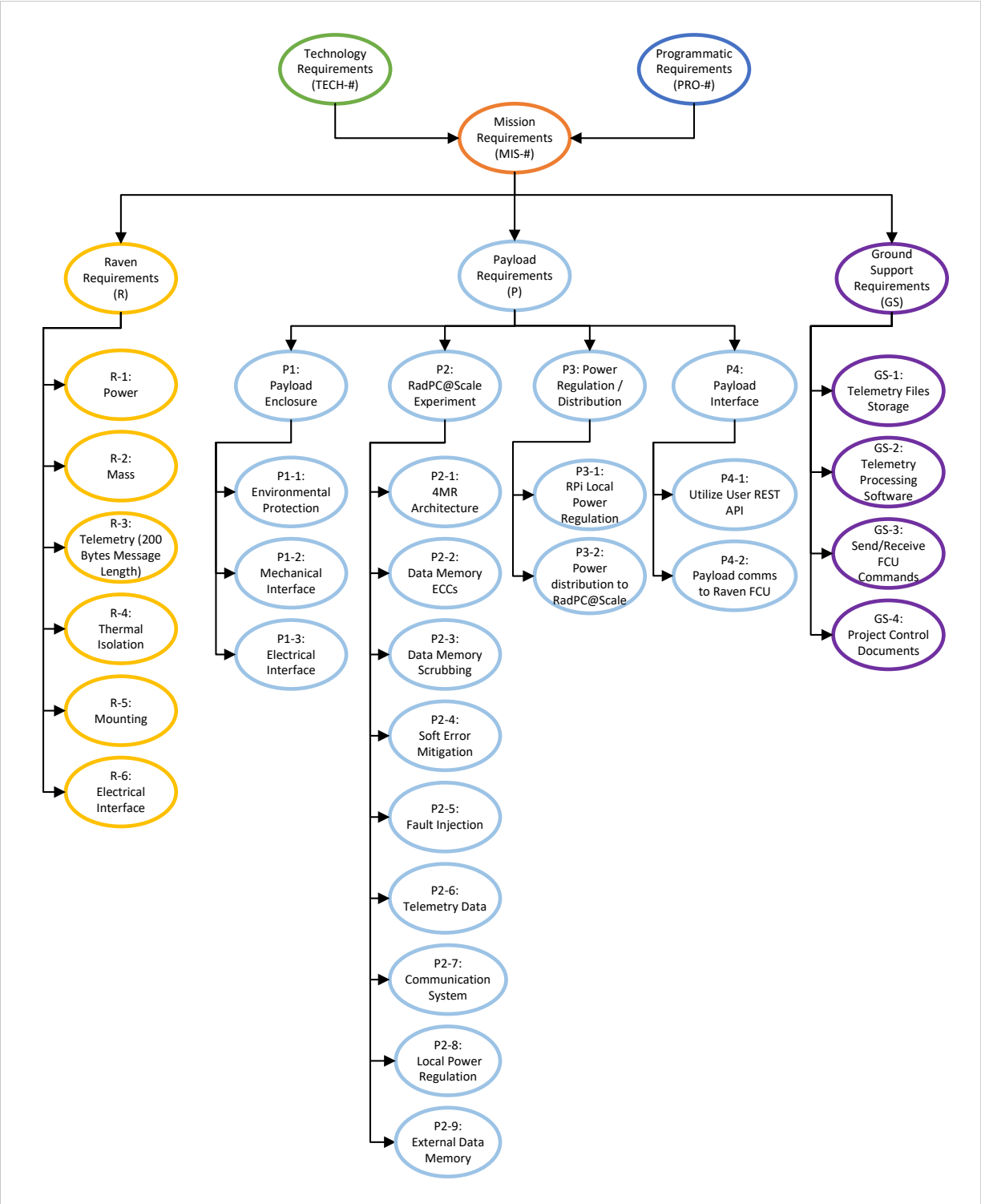


Figure 4.1: Top Level Mission Requirements Flowchart

- R-1: Power — The FCU will provide +28VDC to the payloads. Each payload(s) must not draw more than 100[W] of power.
- R-2: Mass — The payload(s) must not exceed a weight of 25lbs.
- R-3: Telemetry Size — The payload telemetry must not exceed 200 bytes for each message transaction.
- R-4: Thermal Isolation — Operate in extreme hot and cold cases
  - Operate in hot case of up to 80[C].
  - Operate in cold case of down to -83[C].
- R-5: Mounting — The payload(s) must satisfy
  - Withstand forces of up to 6G's in any direction.
  - Include fasteners for redundant safety lanyards
- R-5: Electrical Interface — The FCU must interface to the payload(s) via a an Ethernet TCP socket.

Payload Requirements

The payload must satisfy its pre-existing requirements as well as new requirements that pertain to this project. The payload requirements break down further into four subcategories: Payload Enclosure, RadPC@Scale Experiment, Power Regulation and Distribution, and Payload Interfaces. A detailed breakdown is seen below in a Payload Requirements list.

- P1: Payload Enclosure — The payload enclosure must house the RadPC FPGA experiment and interface circuitry to arbitrate between the Raven FCU and the RadPC FPGA experiment.

  - P1-1: Environmental Protection — The enclosure must protect the RadPC@Scale experiments from hot and cold thermal environments as well as any small particles or incoming debris.
  - P1-2: Mechanical Interface — The payload must mechanically mate with the mounting plates provided by Raven in the MICD. This must ensure that the payload is safely secured to the Raven Thunderhead Gondola.
  - P1-3: Electrical Interface — The payload must operate on a +28VDC power input from the Raven Gondola. The payload must communicate via a TCP ethernet interface for sending and receiving data telemetry.

- P2: RadPC@Scale Experiment — The RadPC@Scale Experiment must implement and/or satisfy:
  - P2-1: 4MR Architecture — The FPGA must implement a 4-Modular-Redundant softcore processing architecture.
  - P2-2: Data Memory ECCs — The data memory between each core must be protected using error correction codes.
  - P2-3: Data Memory Scrubbing — If a fault is detected, the system must scrub its data memory of any faulty values and rewrite the faulted memory addresses with a voted correct data memory value.
  - P2-4: Soft Error Mitigation — The program configuration memory must be monitored with a configuration memory monitor that can detect single or double SEE events and repair them.
    * It must be able to inject single bit SEEs and report its error type and repair this error.
    * Injection and reporting will occur through an external serial interface.

  - P2-5: Fault Injection — The system must implement a means of injecting faults to trigger the recovery mechanisms of the architecture.
  - P2-6: Telemetry Data — The system must log its health over time during the experiment.
  - P2-7: Communication System — The system must export telemetry data via an RS422 serial interface.
  - P2-8: Local Power Regulation — The system must regulate an input voltage of +28VDC down to
    * 5.0V - System voltage rail that supplies each subsequent power rail
    * 3.3V - Voltage level reference for RS422 interface
    * 2.5V - Voltage level for system microcontroller and external IO logic levels.
    * 1.8V - Voltage reference supply for FPGA auxiliary power.
    * 1.0V - Voltage reference for the internal FPGA BRAM, configuration memory, and the core voltage.
  - P2-9: External Data Memory — The system must communicate to a redundant external data memory system.

- P3: Power Regulation / Distribution — The experiments require a system to regulate power and distribute power to all electrical systems.
  - P3-1: RPi Local Power Regulation — The Pi requires a +5VDC voltage for power.
  - P3-2: Power Distribution to RadPC FPGA — The Lunar board must receive a voltage ranging between 24VDC-36VDC for its local power regulation.

- P4: Payload Interface — The payloads must implement:

  - P4-1: Interpret REST User API Calls — The RPi scripts must parse the relevant protocol for commands sent via the FCU during flight.
  - P4-2: Payload Communication Link to FCU — Raven specifies that the payloads communicate to the FCU via a 0MQ TCP socket protocol over an ethernet connection.

Ground Station Requirements

The ground station is the method of creating data requests during flight and housing the data telemetry that is sent back. The ground station must interface to the FCU during flight by implementing Raven's Iridium application programming interface (API). This API provides payloads the capability to create a REST user callback which can be sent from the ground station through Raven's servers to the Iridium network. The telemetry is beamed down to the FCU from the Iridium network. The specific requirements are outlined below:

- GS-1: Telemetry File Storage — The ground station server must house all data telemetry incoming from Raven's API.
- GS-2: Telemetry Processing Software — The ground station must implement packet visualization in both list and graphical format.
- GS-3: Send/Receive FCU Commands and Data — The ground station must be able to create REST User callbacks to Raven's API and receive data responses to its corresponding REST callback.
- GS-4: Project Control Documentation — The ground station must be cleanly documented and exportable to future project use in the RadPC lab.

Requirement Summary

Many of the requirements listed in the Payload Requirements section are inherited from the RadPC Lunar project documentation. As such, the design and experiment documentation focused in this thesis is on the external memory architecture, the interface hardware/software to the Raven FCU, and the ground station implemented for this experiment.

17

RADPC ARCHITECTURE

The existing RadPC architecture employs redundant cores running synchronous to one another. This approach is an extension of the widely adopted triple modular redundant (TMR) approach used in space systems. TMR by itself is not enough to effectively mitigate the effects of cosmic radiation [27]. RadPC extends TMR with an additional core creating a quad modular redundant system (QMR) to provide increased reliability running in conjunction with fault detection and fault mitigation strategies. Additionally, RadPC pairs memory scrubbing with the QMR layout to increase its reliability against SEEs [13]. Each core contains identical program memories that are monitored via external fault detection strategies. The individual cores are referred to as 'Computation Tiles' for the rest of this paper as they represent regions that can be reconfigured individually on the FPGA.

This particular experiment is known as RadPC@Scale extending the existing RadPC architecture by adding additional external memory. Rather than using block RAM instantiated within the FPGA fabric, the computation tiles communicate to external data memories. The tradeoff between using external memory is a slower interface with a significantly larger capacity.

The top-level system architecture is seen in Figure 5.1. The following sections show the abstractions of the RadPC hierarchy. The computation tiles seen in Figure 5.1 are abstracted in the top level architecture. The Data Memory blocks are external integrated circuits (IC) from the FPGA. The Data Memory Scrubber (DMS) is a state-machine within the FPGA. The Checkpoint Bus ensures that the program execution of each computation tile is synchronous to one another. When repairing computation tiles, the checkpoint system allows the repaired core to "catch up" with the other computation tiles before resuming foreground operation. The voter subsystem compares the outputs of the four computation tiles and asserts a data memory scrub process if one computation tile output does not match

the other computation tile outputs. The DMS walks through memory sequentially and compares the outputs of all the computation tiles and overwrites any computation tile with a mismatched memory value to the correct output from the others. Finally, the Configuration Memory Monitor (CMM) is a subsystem that monitors any single bit flip events in each computation tile's configuration memory. Additionally, the CMM can receive commands via serial bus to inject faults in configuration memory and trigger its repair mechanisms. Any time a fault has been repaired, it will send a message over the serial interface to the external micro-controller that it has repaired a single error in configuration memory.

RadPC IC Part Selection

The RadPC platform aims to be a cost-effective, radiation tolerant SBC targeting missions in LEO and beyond. To satisfy these design goals, many parts were picked with care for the system. Table 5.1 identifies the parts and a brief justification on why that particular IC was selected. Much of this design was completed prior to the RadPC@Scale project, so what is listed is what was flown during the high-altitude balloon flights.

RadPC@Scale implements its softcore processors on a Xilinx Artix A7-35T FPGA, where as the computers proposed in Chapter 3 use a Xilinx XCKU5P FPGA and a Xilinx Nexys 4 SoC FPGA, which is a development board that uses a Xilinx Artix A7-100T. Table 5.2 compares the specifications of the FPGAs against one another. The Kintex FPGA blows the Artix FPGAs out of the water in terms of number of CLBs, slices, BRAMs, and logic cells. However, the purpose of the RadPC@Scale system is to provide a radiation tolerant computer using COTS products at a reasonable cost. The Artix 7 family satisfies those parameters while still providing sufficient CLBs, LUTs, and BRAMs.

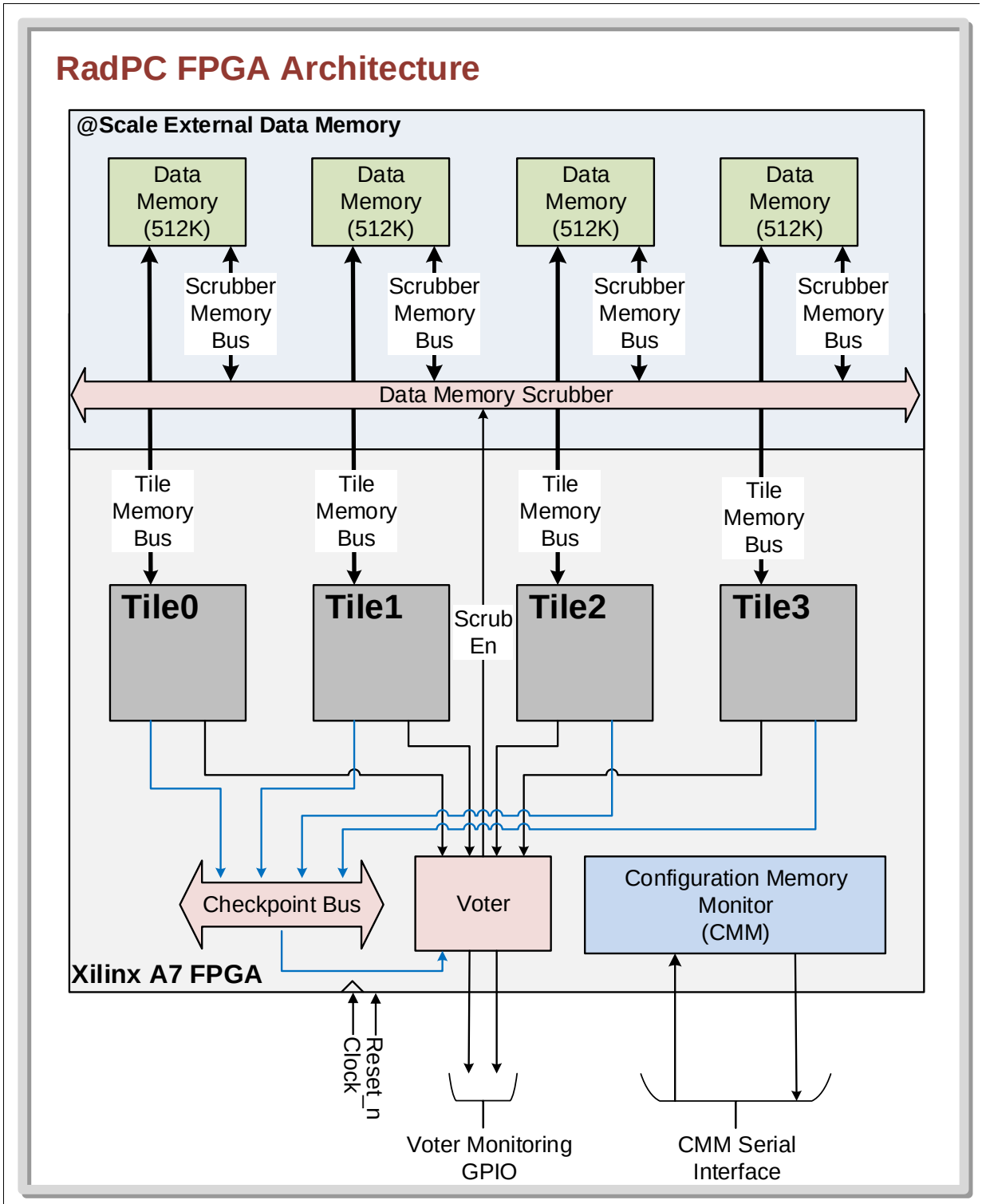


Figure 5.1: RadPC at Scale FPGA Architecture

IC	RadPC Functional Description
TI microcontroller MSP430FR6994	• FPGA Data Monitoring • Power rail monitoring (5V, 3.3V, 2.5V, 1.8V, and 1.0V) • Packet construction in FRAM • FPGA Full Configuration and Partial Reconfiguration (PR)
System FPGA Xilinx Artix XC7A35T-1FTG256I	• NMR Tile Architecture • Full Tile Design (Note: Detailed in Chapter 6)
NAND Flash Memory IS25LP256D-JLLE	Data telemetry storage and FPGA bitstream storage
Current Sensor INA186A2IDCKR	Measure current rails and feed to MSP ADC
RS422 ↔ UART Transceiver MAX14854GWE+	Convert UART to RS422 to satisfy Lunar Payload Comms Requirement

Table 5.1: RadPC SBC Part Selection and Justification

	Xilinx Kintex XCKU15P	Xilinx Artix A7-35T	Xilinx Artix A7-100T
System Logic Cells (K)	1,143,450	33,280	101,440
Block Memory (Kb)	34,600	1,800	4,860
CLBs (Slices)	N/A	5,200	15,850
CLBs (Max Dist. RAM) (Kb)	9,800	400	1,188
DSP (Slices)	1,968	90	240
I/O pins	544	250	300

Table 5.2: FPGA Comparisons

RadPC Mitigation Strategies

For the RadPC Lunar Mission, the team has outlined various potential fault points and how each of the corresponding mitigation strategies within the architecture will mitigate these fault points. The visual representation of this is illustrated in Table 5.3.

Fault Condition	Observed By	Action Taken	Action Owner	Fault Severity	Fault Prob
SEU in Tile foreground circuitry	Voter	PR effected Tile	MCU	MEDIUM	MEDIUM
SEU in foreground circuitry outside of Tiles	Voter	Full FPGA Reconfiguration	MCU	HIGH	LOW
Single Bit Error in off FPGA Data Memory	ECC	Automatic Data Recovery by ECC	Tile Data Memory ECC	LOW	MEDIUM
Multiple Bit Error in off FPGA Data Memory	Voter	Overwrite corrupted with values from other healthy data memories	DMS	MEDIUM	LOW
Single Bit Error in Configuration Memory	CMM	Automatic Correction	MCU	LOW	HIGH
Double Bit Adjacent Errors in Configuration Memory	CMM	Automatic Correction	MCU	LOW	MEDIUM
Double Bit, Non Adjacent Error in Configuration Memory	CMM	Full FPGA Reconfiguration	MCU	HIGH	LOW

Table 5.3: Summary of FPGA Fault Conditions and Mitigation Approach

Faults that are detected via the Voter take longer to repair as it will cause a system-wide pause. Faults being repaired by the ECC or the CMM are much faster to repair unless the CMM detects a double-bit non-adjacent fault.

RadPC Architecture Payload Requirements

This thesis will show the RadPC@Scale architecture at different key hierarchical levels. From the FPGA RadPC hierarchy level shown in this chapter, requirements P2-1 and P2-4. Chapters 6 and 7 will satisfy the remaining requirements that pertain to FPGA design

methodology.

The on-board power regulation scheme and micro-controller satisfy telemetry data generation and storage, communication to external devices, and local power regulation within the payload requirements section. The micro-controller communicates to an RS422 transceiver chip via UART to transmit packets and receive command requests. These features were implemented by the RadPC lab and are not the focus of this research. This research extends the functionality of the RadPC computer and inherits these requirements. To see more, please see additional publications by Brock J Lameres and Chris Major on the current workings of the RadPC project.

RADPC COMPUTATION TILE ARCHITECTURE

Figure 6.1 shows the computation tile architecture targeting the Xilinx Artix 7 FPGA series. Each computation tile implements a Xilinx microblaze with program memory, an AXI peripheral bus controller, and GPIO blocks associated with memory and communication to external devices. The block labelled 'Memory Controller' is abstracted by each computation

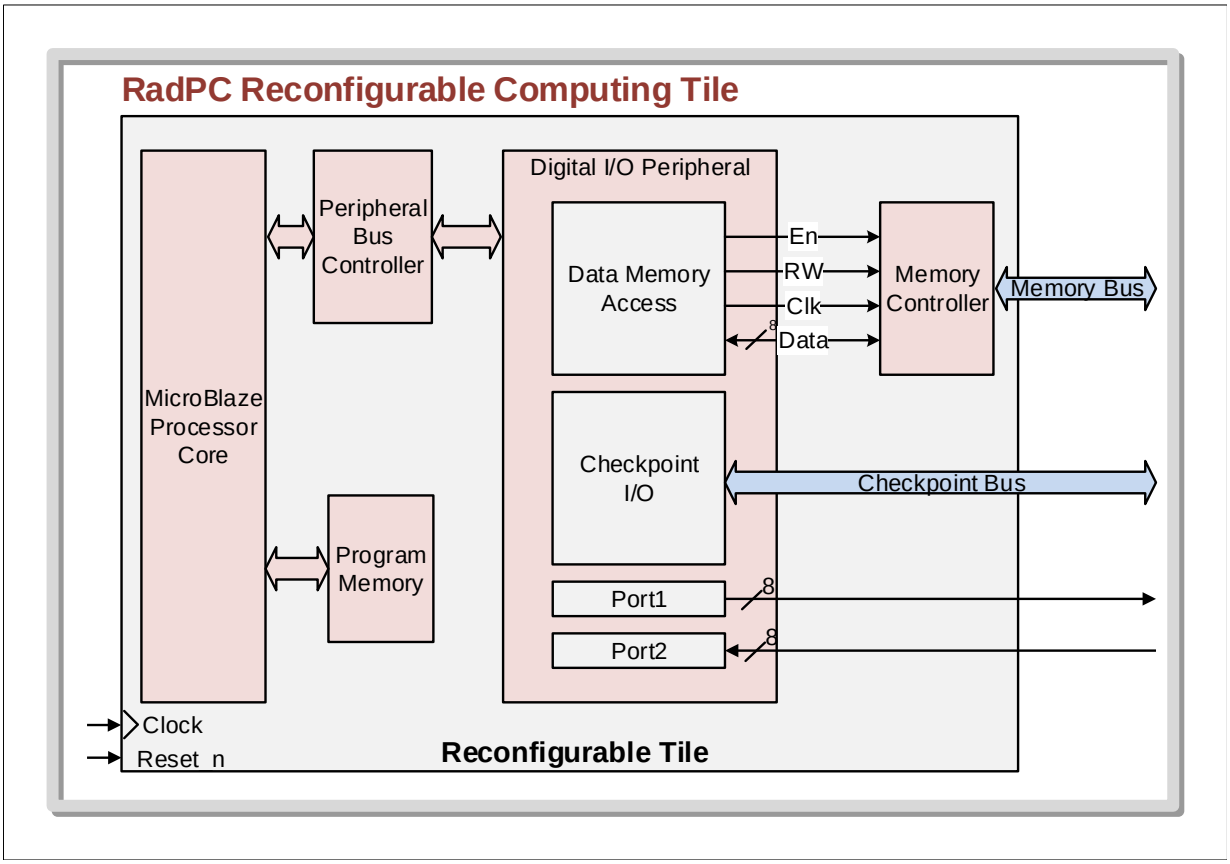


Figure 6.1: RadPC at Scale Tile Architecture

tile and is depicted in the following section. The peripheral bus controller arbitrates the digital input/output (I/O) peripherals to the Microblaze processor core. The microblaze computation tiles communicate to the crossbar which then routes the signal busses to the varying GPIO objects [30]. The Data Memory Access block is what connects to the Memory

Controller blocks that are discussed next chapter. The Checkpoint I/O communicates via the Checkpoint Bus to the external instrumentation micro-controller. The Voter is in charge of reconfiguring these computation tiles when faults arise based on the outputs of each computation tile's checkpoint bus. Each computation tile has 8Kb of program memory for the user program.

25

REDUNDANT EXTERNAL MEMORY ARCHITECTURE

The Redundant External Memory Architecture incorporates several blocks to implement resilience to ionized radiation. The memory architecture is a series of control interfaces that arbitrate between computation tile memory commands and the DMS. This chapter explains how the computation tiles/scrubber communicate to the external SRAM through a serial peripheral interface (SPI), the DMS interface, and the Stormbreaker fault injection blocks.

First, it implements a SPI finite-state-machine (FSM) to control a low-level SPI driver instantiated within the FPGA fabric. The SPI control FSM has enable inputs from both the processing computation tiles as well as the DMS. The DMS has priority access to the data memories over the processing computation tiles. Second, there are error-correction-code encoding and decoding blocks. These blocks detect when a single bit SEE fault has occurred and repair the value if it has been detected. Finally, there is a separate fault injection methodology that will inject faults when the computation tiles try to write to a certain address in memory. These blocks are called "Stormbreakers" whose primary function is to trigger the RadPC@Scale recovery mechanisms by injecting faults directly into external memory, thus bypassing the microblaze computation tiles. When the computation tile reads back a value from external memory that has had a fault injected, it will trigger the full system recovery blocks. Figure 7.1 depicts the computation tile memory architecture where the Memory Controller block is in charge of the arbitration between computation tile memory access and DMS access.

RadPC@Scale SRAM Memory Module

This section will focus on the 23LC1024 SRAM memory module, its specifications, and how to communicate to it as a SPI master device.

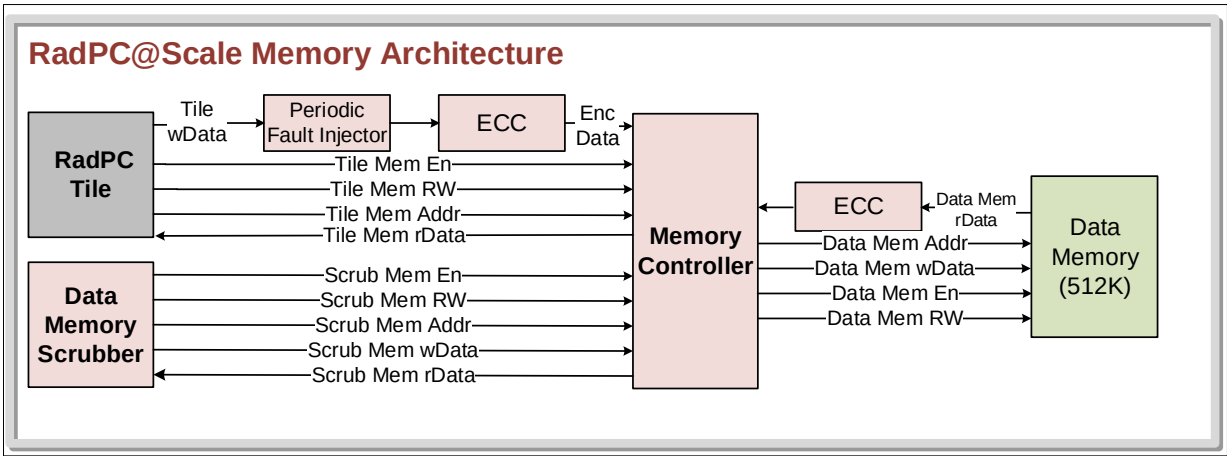


Figure 7.1: Simplified computation tile memory architecture.

23LC1024 SRAM Module

In Chapter 2, research was presented that outlines DRAMs increased resilience to SEEs over SRAM in Figure 2.2. While Figure 2.2 outlines the increased SEE susceptibility of SRAM, the development time to implement a the communication protocol for DRAM with four computational RadPC tiles proved to be not viable to meet the project deadline. However, the mitigation approaches of the NMR+voter, memory scrubber, and hamming ECCs are applicable to DRAM technology. A more in depth analysis outlining the decision between SRAM and DRAM are outlined in the failure mode and effects analysis (FMEA) chart in Figure 7.2. The timing analysis and memory arbitration blocks to allow four separate computation tiles to interface to a single DRAM memory posed too high of a failure probability over that of a slower, lower capacity SRAM.

With the exclusion of DRAM as a factor for the external data memory to the RadPC@Scale computation tiles, the selection of components came down several factors: part availability in 2020 with an ongoing chip shortage, physical package, and memory capacity for simple serial memory devices. This lead to the decision of the 23LC1024, which was in stock and contained the largest memory capacity while not being a BGA or BGA

System	RadPC@Scale	Potential Failure Mode and Effects Analysis (Design FMEA)		FMEA Number	RadPC@Scale 1
Subsystem	Memory Interface Controller			Prepared By	Justin P Williams
Component	External Memory PMOD PCB			FMEA Date	12/22/2021
Design Lead	Justin P Williams	Key Date	12/22/2021	Revision Date	12/22/2021
Core Team	RadPC Team			Page	1 of 1

Item / Function	Potential Failure Mode(s)	Potential Effect(s) of Failure	Sev	Potential Cause(s)/ Mechanism(s) of Failure	Prob	Current Design Controls	Det	RPN	Recommended Action(s)	Responsibility & Target Completion Date	Action Results: Actions Taken	New Sev	New Occ	New Det	New RPN
Microchip 23LC1024 1Mb SRAM SPI Module	SEE sensitive	SEU	3	Cosmic radiation.	3	Voter.	2	**18**	Memory scrubber. Redundant Memory array. ECC.	Justin P Williams, May 20, 2021.					
Micron MT45V64M4 DDR SDRAM	FPGA Timing failure analysis	Won't meet project deadline.	4	16MHz uBlaze clock versus 200MHz SDRAM clock	3	In lab testing.	2	**24**	Pick a different memory IC.	Justin P Williams, May 20, 2021.					
	SEE sensitive	SEU	3	Cosmic radiation.	1	Voter.	2	**6**	Memory scrubber. Voter. ECC.	Justin P Williams, May 20, 2021.					

Figure 7.2: FMEA SRAM vs DRAM — Timing Closure and SEU Resilience

related package type.

23LC1024 Memory Specifications

The 23LC1024 is organized in a 128Kx8 memory array, that is accessible via an input command, an input address, and input data if the device is being written to. The maximum operating frequency of the 23LC1024 is 20MHz, with support for byte, page, and sequential access modes for read and writes [21]. The maximum memory bandwidth for the device is outlined by Equation 7.1.

$$BW = 16,000,000 \left[\frac{clocks}{sec}\right] \cdot \frac{1}{8 \cdot 40} \left[\frac{bytes}{bits \cdot BufferLength}\right]$$
$$BW = 0.40 \left[\frac{MB}{s}\right] \tag{7.1}$$

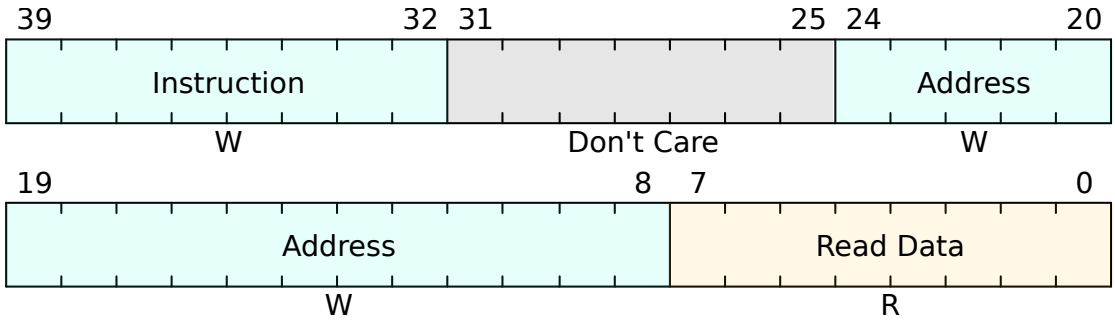
Using the Equation 7.1, Table 7.1 outlines the maximum BW of the 23LC1024 as the clock speed varies.

Clock Rate [MHz]	Memory BW [MB/s]
8	0.20
10	0.25
12	0.30
16	0.40
20	0.50

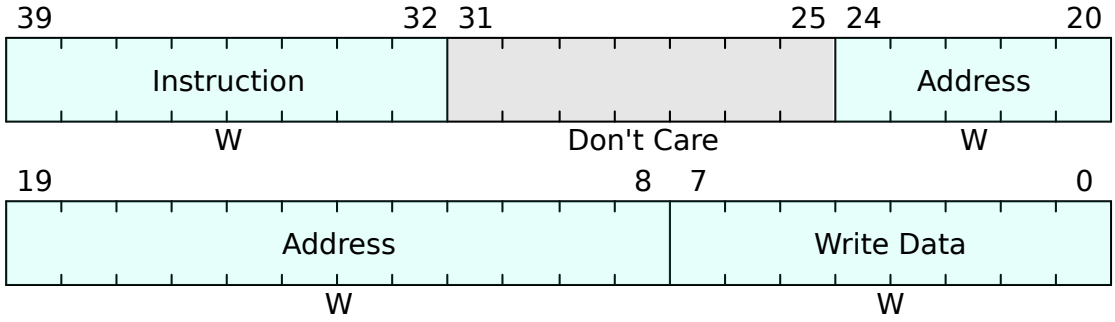
Table 7.1: Memory BW Versus Clock Speed

SPI Memory Transactions

The processing computation tiles write to the external SRAM data memory by sending a write instruction followed by an address, and the data being written. A bitfield of the write transaction can be seen in the Figure 7.3b. The processing computation tiles read from the external SRAM data memory by sending a read instruction followed by an address. A bitfield of the read transaction can be seen in Figure 7.3a. Note that the SRAM module takes seven clock cycles to load in an instruction before the 17-bit address and 8-bit data byte can be sent. The read data is colored differently, as the SRAM is replying with data to the FPGA. The waveforms of both SPI transactions between a single computation tile and its corresponding SRAM module can be seen in Figure 7.4. Figure 7.4 is two stacked waveforms of two groups. The top group is the first half of the SPI transaction, and the bottom group is the second half of the SPI transaction. The top stack of waveforms depicts a SPI Write from either the computation tile or DMM to the SRAM. The bottom stack is the read. During the write process, the Master-In, Slave Out (MISO) line is at a High-Z state. During the read transaction, the MISO line will respond with data at that specific clock cycle to the computation tiles/DMM. There are clock break lines on all transactions to signify that there are more clock cycles for loading in the instructions, address, and write



(a) Requesting data from a SRAM Module



(b) Writing data to a SRAM Module

Figure 7.3: Read/Write Bitfield Instructions

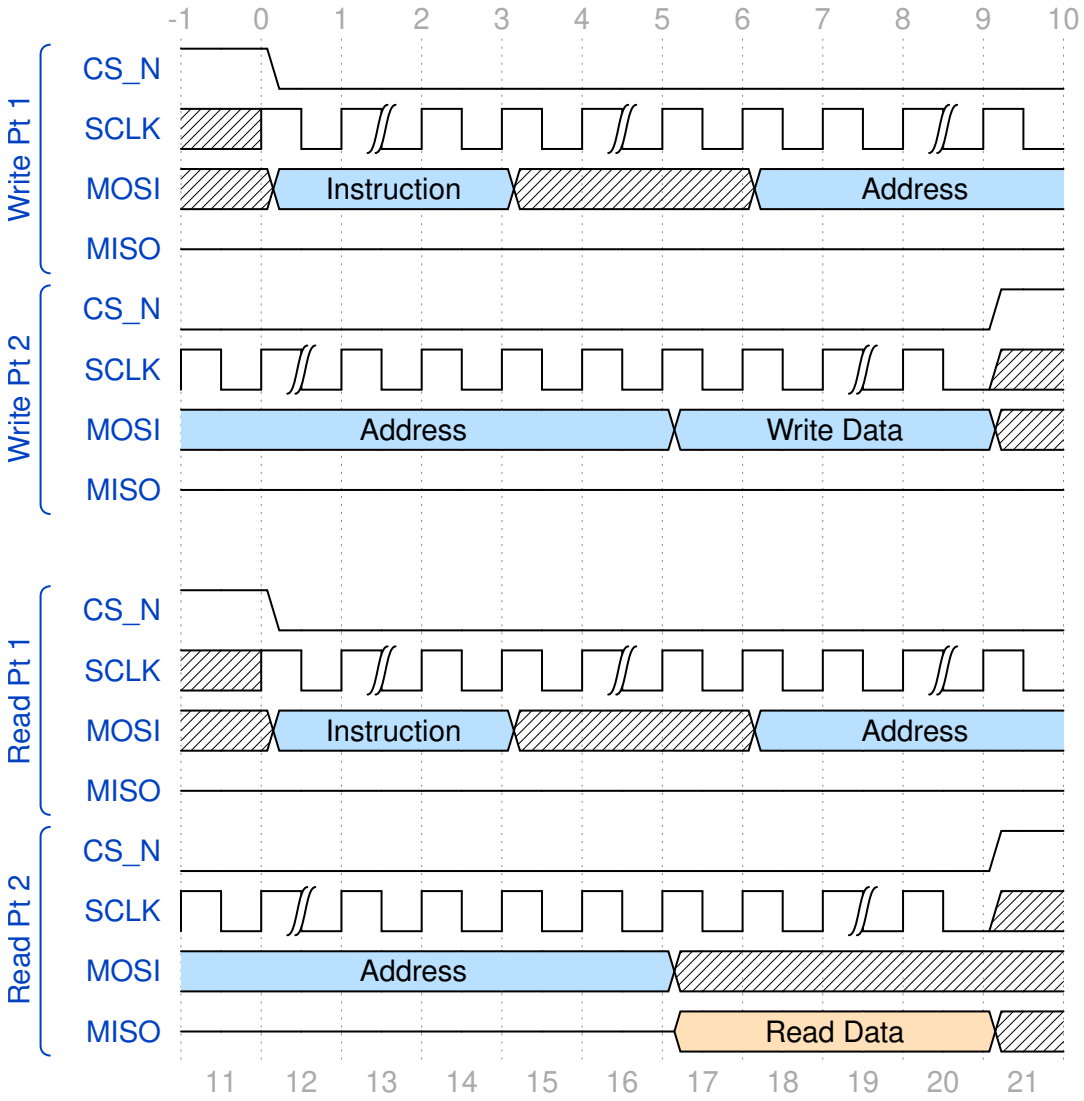


Figure 7.4: SPI Waveform transactions between computation tiles (or Scrubber) and SRAM

data/read data than what is depicted in Figure 7.4.

23LC1024 Die Size and Expected SEUs During Flight

With the 23LC1024 SRAM power supply requirements and date of production, it can be assumed that its process node size is between 150-180nm [22]. Given that assumption, we can refer back to Figure 2.2 and outline where we can expect the system to perform in Figure 7.5. Figure 7.5 outlines the number of upsets we can expect on a 23LC1024 in NYC

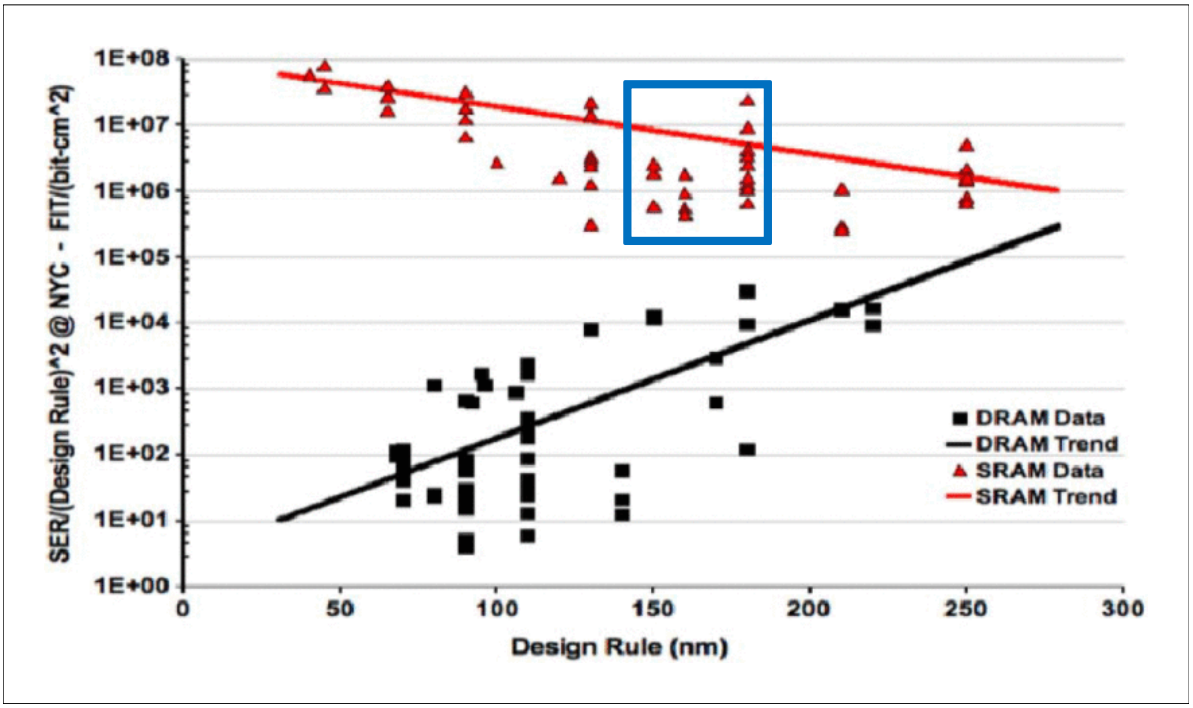


Figure 7.5: Expected soft errors of the 23LC1024 in NYC from High Energy Neutron Particles [26]

caused by high energy neutron particles is around $1E+07$ FIT/chip, according to Slayman's research [26].

Previous high altitude balloon flights on NASA's Rad-X high altitude ballooning program led to some experiments that measured TID and the expected amount of SEEs on SRAM devices. The Rad-X program ran a high altitude balloon for a research team

Altitude [ft]	Altitude [km]	SEUs $Gbit^{-1}Day^{-1}$ R = 0 GV	SEUs $Gbit^{-1}Day^{-1}$ R = 4 GV	SEUs $Gbit^{-1}Day^{-1}$ R = 8 GV
65,616	20	40	22	9
131,233	40	100	24	9
196,850	60	160	24	9
262,467	80	155	23	9
328,083	100	150	22	9

Table 7.2: Expected SEUs for 23LC1024 SRAM modules

led by James Rosenthal which estimated that a payload, at an altitude of around 80,000 feet, should experience TID effects of $3.52 \pm 0.70[\mu Gy/hour]$ [23]. On a separate Rad-X high altitude balloon flight, a team of researchers led by Alex Hands flew 2 separate Hitachi 4 Mbit SRAM memory devices as well as radiation detectors to measure how many SEEs/SEUs the Hitachi SRAM devices experienced as a function of altitude [15]. Hand's research calculated the number of SEUs the SRAM devices experienced given the radiation measurements and varying rigidity cutoffs, and created plots shown in Figure 7.6. Figure 7.6 displays varying levels of resilience to SEUs due to the levels of geomagnetic shielding that's provided to the different SRAM devices. Observe that the primary cause for SEUs in this research is from proton particles. From these plots, Table 7.2 has been generated to show the expected faults caused by proton particles that the 23LC1024 would experience provided these varying levels of geomagnetic rigitidies [15]. Evaluating Table 7.2, and assuming a geomagnetic rigidity of 4 GV, then for the 52 hour RadPC@Scale flight, the number of SEUs per Gigabit per day that the 23LC1024 could expect to experience is between 44-50.

Stormbreaker Fault Injection Blocks

The Stormbreaker blocks inject faults periodically based on predefined user registers. Each time a fault is injected, the predefined fault address is incremented by some user-

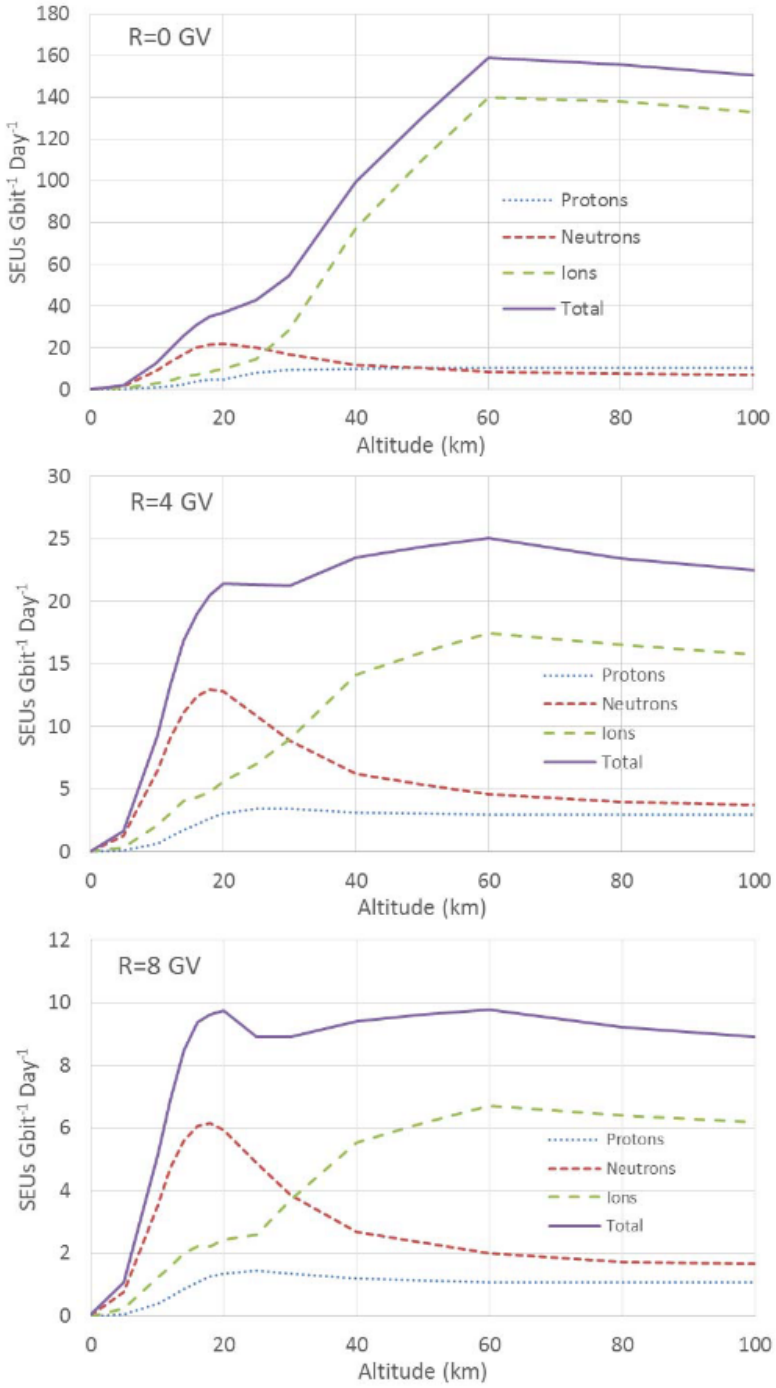


Figure 7.6: Calculations of SEU rates vs. altitude for protons, neutrons and ions at three cut-off rigidities for a Hitachi SRAM [15]

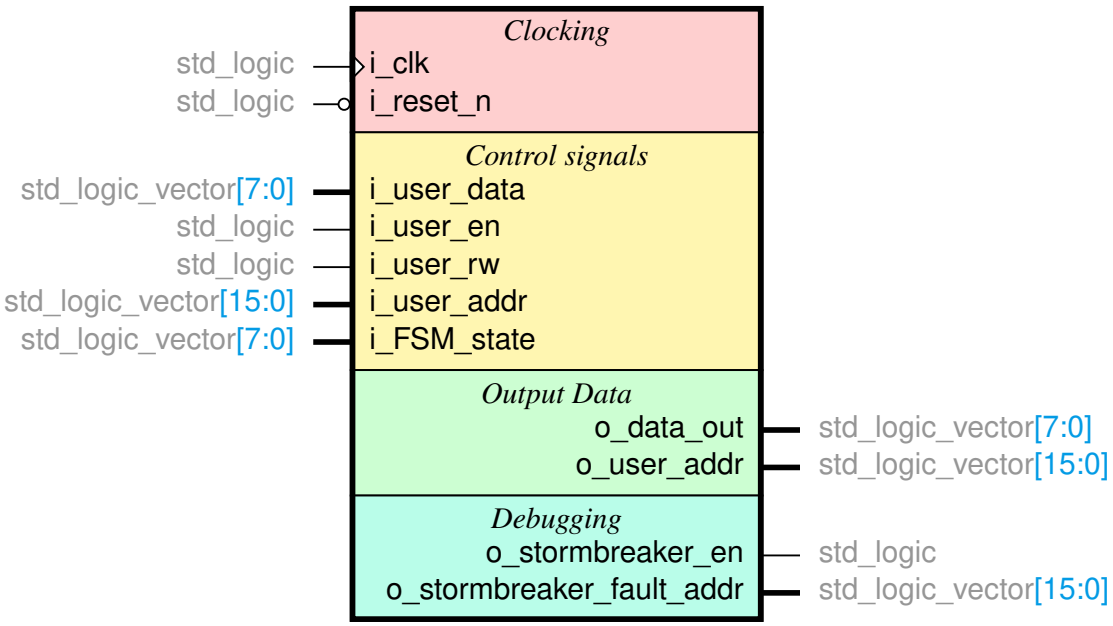
defined fault address offset. This cycle is repeated through memory and repeatedly triggers the recovery systems of the RadPC@Scale architecture. Figure 7.7 depicts the design flow in which Stormbreaker operates. The entity of the stormbreaker component is outlined in Figure 7.7a and the dataflow diagram is outlinend in Figure 7.7b. When the input data stream comes in, it compares the input address that is being written to a registered fault address. If the fault address equals the target input address, it will output faulty data which is then written to that target address. When this block is enabled, the fault address is incremented by a user-defined amount and waits for the input target address to then equal the new fault address. Otherwise, the data is passed through, unaltered.

Hamming Correction Codes

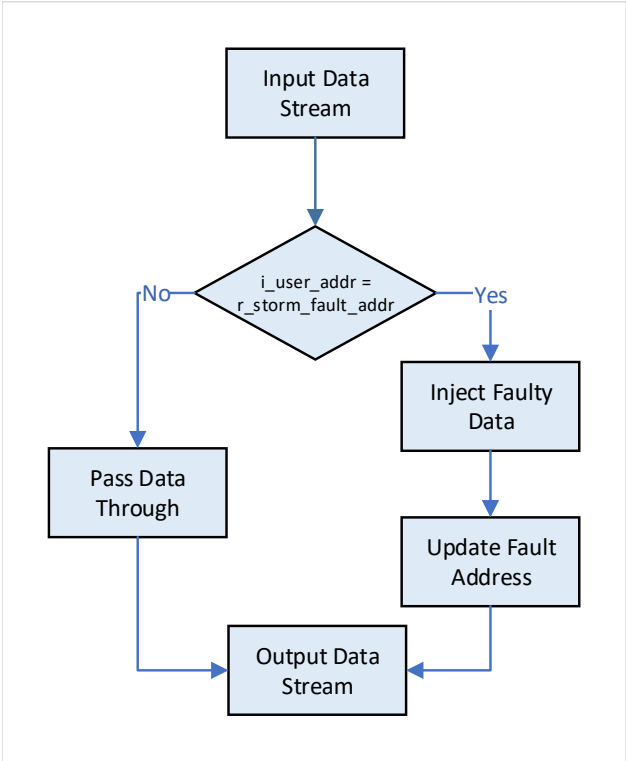
Hamming Code Background

Hamming correction codes were invented by Richard Hamming at Bell Laboratory in the late 1940s. They were invented due to the early Bell Labs computer encountering run-time errors that would require a full restart. Rather than performing the full restart on an error, Richard Hamming decided that there should be a means of automatic error detection and correction during run-time [28].

In the case of this thesis, a hamming code structure of (12,8) was used in the design that was flown on the high altitude balloon flight. This ensured that the memory hamming could detect a 1-bit error and correct the 1-bit error if any occurred. If a 2-bit adjacent error or any other kind occurred, then the hamming code implementation could not have identified and corrected it. The specific implementation of this hamming encoder and decoder within the FPGA was developed by Skylar Tamke, and implemented by Annie Bachman. For further details, please refer to Skylar's MS thesis for a more thorough explanation of hamming codes [28].



(a) Stormbreaker Interface Entity



(b) Stormbreaker Dataflow

Figure 7.7: Stormbreaker Entity and Dataflow

Hamming Encoding Theory

The idea of a Hamming Code is to use logical XOR gates on data bits to find parity bits. When performing error encoding, the designer must calculate the minimum amount of parity bits for the length of data being encoded. This means that the word length is 12 bits long and encodes a input word of 8-bits, with 4-bits of parity. The minimum amount of parity bits required is modeled in Equation 7.2 [28].

$$n \geq 2^m - 1 \tag{7.2}$$

To more easily conceptualize the procedure of encoding data in a (12,8) structure, Figure 7.8 illustrates the process of calculating the parity bits for an input data size of 8. Mathematically, the parity bits being generated are modeled by Equations 7.3, 7.4, 7.5, and 7.6.

$$P1 = D1 \oplus D2 \oplus D4 \oplus D5 \oplus D7 \tag{7.3}$$

$$P2 = D1 \oplus D3 \oplus D4 \oplus D6 \oplus D7 \tag{7.4}$$

$$P3 = D2 \oplus D3 \oplus D4 \oplus D8 \tag{7.5}$$

$$P4 = D5 \oplus D6 \oplus D7 \oplus D8 \tag{7.6}$$

P1, P2, P3, and P4 represent the parity bits for the data range of D1 through D8, where the '$\oplus$' symbol represents the logical XOR operation. The structure of the parity bits takes on the form of Figure 7.9 where the parity bits are interspersed throughout the 12-bit word [28].

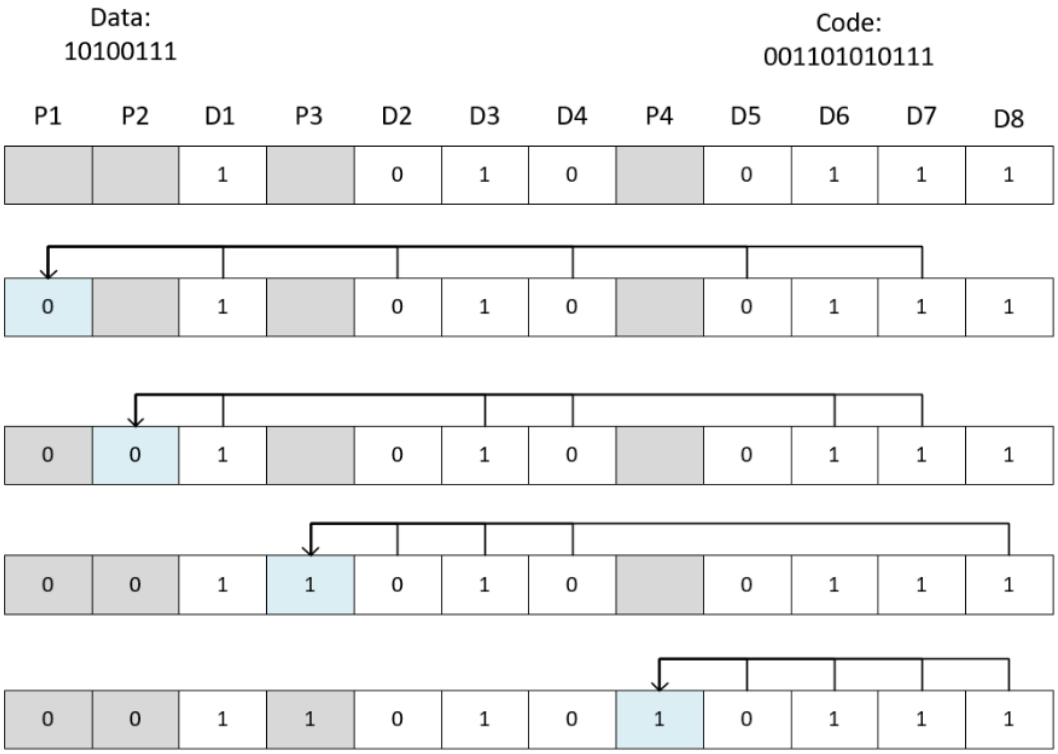


Figure 7.8: Non-systematic approach to generating hamming parity bits on a (12,8) structure [28]

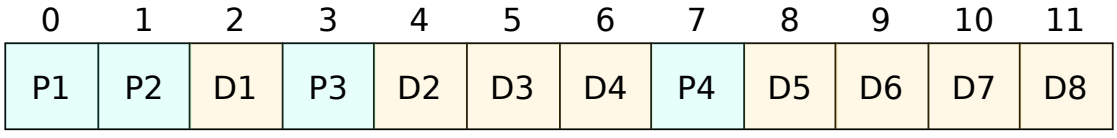


Figure 7.9: (12,8) Hamming Codes Bitfield Structure

Hamming Decoding Theory

The decoding process of hamming codes incorporates generating a 'syndrome vector' that compares against the parity bits. The syndrome vector checks one of two things when decoding a hamming code:

1. A set of parity bits versus parity check bits
2. A set of equations using parity bits

For (12,8) hamming codes, the engineer can calculate the syndrome bits exactly the same as the parity bits and compare the results, shown in Equation 7.7.

$$S1 = D1 \oplus D2 \oplus D4 \oplus D5 \oplus D7 \tag{7.7}$$

If they syndrome bit is different than the parity bit, then an error has occurred. The correction process involves looping through the parity versus syndrome bit comparisons to identify if a fault has occurred. The faulted bit can then be overwritten with it's inverse.

RadPC@Scale Hamming Implementation

RadPC@Scale implements the documented (12,8) structure outlined in the Hamming Encoding and Decoding sections in the FPGA fabric. Because the hamming code circuitry is combinational logic, the outputs of the encoder and decoder occur within the same clock cycle as the inputs that drive them. A VHDL code snippet of the encoding process is shown in Code Snippet 7.1.

```
1  w_parity1 <= ( r_data(0) XOR r_data(1) XOR r_data(3) XOR r_data(4) XOR r_data(6) );
2  w_parity2 <= ( r_data(0) XOR r_data(2) XOR r_data(3) XOR r_data(5) XOR r_data(6) );
3  w_parity3 <= ( r_data(1) XOR r_data(2) XOR r_data(3) XOR r_data(7) );
4  w_parity4 <= ( r_data(4) XOR r_data(5) XOR r_data(6) XOR r_data(7) );
```

Source Code 7.1: Hamming Encoding VHDL

The signals in Code Snippet 7.1 w_parity1 through w_parity4 are the parity bits associated with the encoded data word. To finish the encoding process, the data must be output following the hamming code algorithm order in Code Snippet 7.2.

```
1  o_output_code <= (r_data(7 downto 4) & w_parity4 & r_data(3 downto 1)
2          & w_parity3 & r_data(0) & w_parity2 & w_parity1);
```

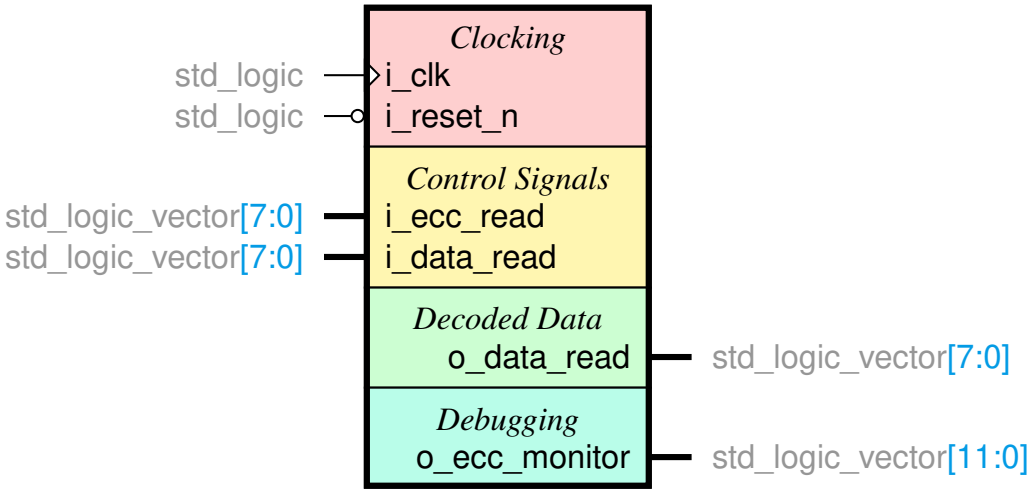
Source Code 7.2: Hamming Output Signal VHDL

The signals in Code Snippet 7.2 r_data is the input byte and o_output_code is the output word that is 12-bits long. The decoding process simply inverts this encoding process and compares the input parity bits. If a single bit error is detected, it can fix this error by comparing the parity bits. The top-level entities for the encoder and decoder blocks can be seen in Figures 7.10a and 7.10b.

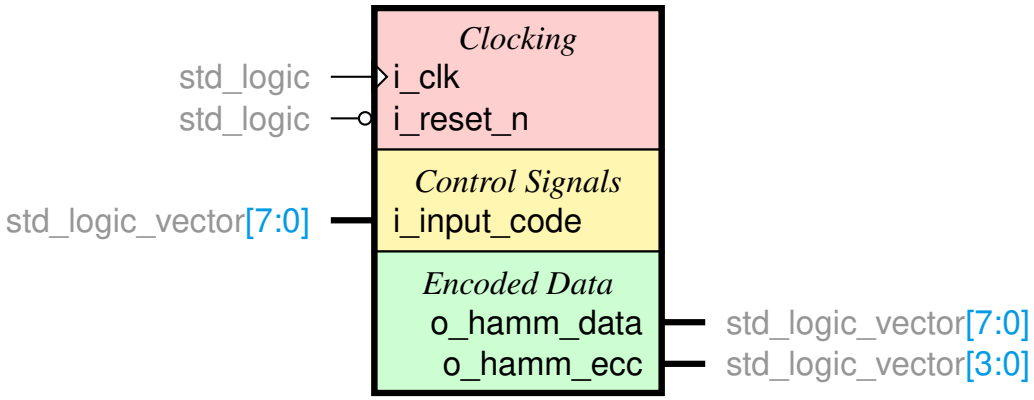
Hamming Code Implementation Tradeoffs

Within the memory system, there are multiple forms of hamming codes that I could have implemented for the system other than the (12,8) structure. Due to the inherit design of hamming encoding, there comes the trade-off of error detection certainty and error resilience. This introduces the concept of Single Error Correct, Double Error Detect (SECDED) codes. The SECDED allows the decoder to detect a single bit error and correct it, as well as detect double bit errors without correction.

However, the designer can also opt to detect triple bit errors. Once the detection of triple bit errors is implemented, the topology is not able to both correct single bit errors and detect triple bit errors reliably, thus enforcing the aforemetioned trade-off between certainty versus reliability.



(a) Memory Decoder Entity



(b) Memory Encoder Entity

Figure 7.10: ECC Hamming Correction Memory Encoder and Decoder Entity Diagrams

RadPC@Scale focuses on the resilience to the harmful effects of radiation. Reliability is key for the RadPC@Scale memory system, and the consistent detection and correction was critical for ensuring that the system maintains operation in harmful space environments. The hamming decoder detects and repairs single bit errors significantly faster than the Voter + DMS recovery mechanisms. From the existing architecture, there isn't a need to report that a double bit error has occurred with the Voter will detect the double bit error and invoke the DMS. Therefore, the implementation of the SECDED hamming codes were deemed unnecessary for this project flight.

RadPC System Memory Controller

The memory controller state machine takes in two pairs of control signals to interface to the external SRAM modules. The memory controller FSM is instantiated once per computation tile. It is in charge of arbitrating between memory access requests from the processing computation tiles and implementing recovery procedures using the DMM if it is called. The top-level interface can be visualized in Figure 7.11. There are two sets of input control signals that drive the memory controller seen in Figure 7.11: The computation tile control signals, and the scrubber control signals. Both sets of control signals can enable memory transactions to the SRAM external memory. This memory controller is quadruplicated and given out to one computation tile each. The DMM is connected to all four memory controllers, seen in Figure 7.1, as it requires reading and writing to each one synchronously while it is operating. The SPI Controller Inputs interface to the SPI driver peripheral. Again, for redundancy, the SPI peripheral is also quadruplicated and given to a computation tile each. The SPI Controller Outputs are fed back into the Memory Controller as outputs from the SPI peripheral. These signals update the memory controller when a transaction is completed.

The data inputs on the computation tile control signals are outputs from the hamming

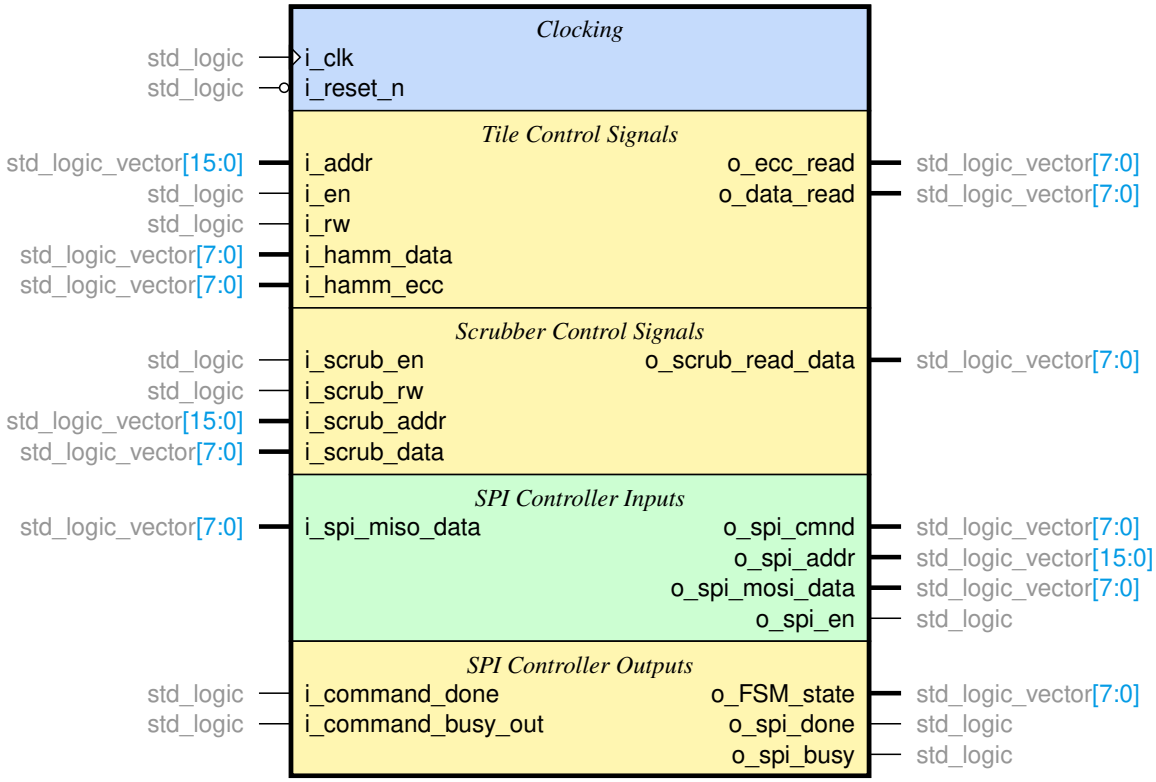


Figure 7.11: RTL Entity of Memory Controller

memory encoder. The SRAM module has a byte address width whereas the data post-encoding is a 12-bit word. The addressing scheme of the memory system modeled by Equations 7.8 and 7.9.

$$\text{Encoded Data Address} \qquad SourceDataAddr = 2 * (InputAddr) \tag{7.8}$$

$$\text{Hamming ECC bits} \qquad SourceDataAddr = 2 * (InputAddr) + 1 \tag{7.9}$$

Reading and writing to the SRAM modules is outlined in Section 7, however, the RadPC@Scale memory controller must communicate to the SRAM modules by sending a 12-bit encoded word to a memory module with an address width of a byte. Recall that a visual representation of the 12-bit word is shown in Figure 7.9. In order to store the entire 12-bit word, the system had to trigger two full memory transactions for each word that is being read from or written to the redundant memory array. Figure 7.12 further illustrates the distinct memory paths between the DMS and the Computation tile. Within each data path, either the DMS or Computation Tile, each must first send the instruction, address, and data if it's a write transaction, to the SRAM module. It will do this twice to fully store the 12-bit word where the first transaction operates on the lower byte of the 12-bit word. The second memory transaction operates on the upper nibble of the 12-bit word.

A visualization of an example 12-bit ECC encoded word being written to physical memory within the 23LC1024 SRAM module is shown in Figure 7.13. The blue block in Figure 7.13 is outlining the usage of the memory encoder and the memory controller block to physically write the 12-bit encoded word to two separate adjacent memory addresses within the 23LC1024 SRAM device.

The system is not utilizing memory to its full capacity as the system is only using 75% of its address space by writing 12 total bits to 2 addresses of 16 bits. Adding more ECC bits would better utilize the memory address space, but Voter already accounts for the detection

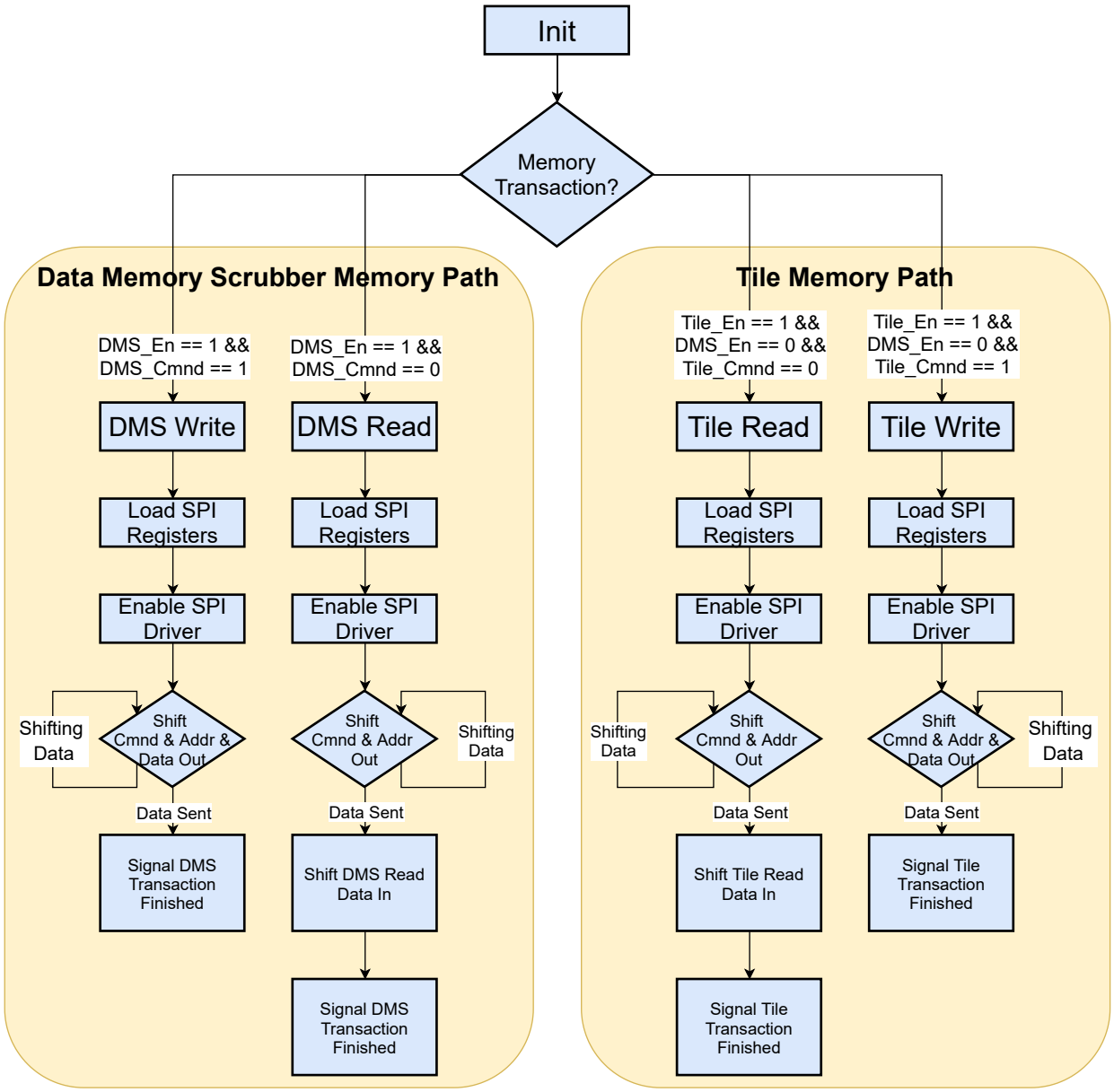


Figure 7.12: System Level Memory Interface Flowchart

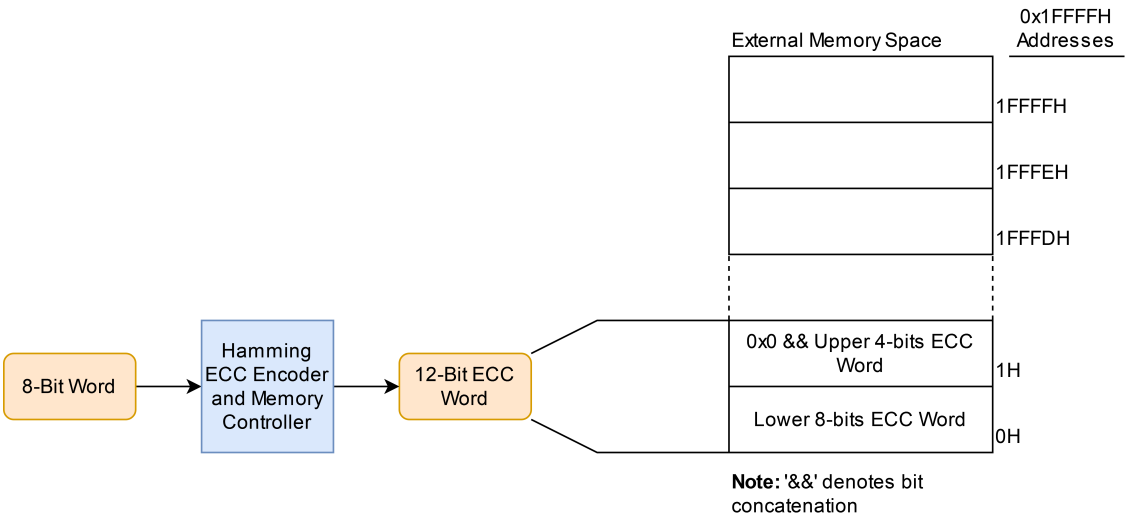


Figure 7.13: Physical memory layout of an individual encoded byte

and recovery of double-bit errors incurred by manual fault injection or cosmic radiation. Therefore, adding more ECC parity bits would not be a more optimal use of the wasted address space. That development time would be better spent at optimizing the addressing scheme of the memory itself, rather than the increase of ECC parity detection.

Redundant Memory Array and Hamming Codes

The use of redundant memory systems (one per computation tile) paired with hamming codes provides an inherent speed bonus versus using one memory system (all computation tiles communicate to one memory system). Using a single DRAM with increased hamming ECC's was considered, but ultimately, it was decided to go with the redundant array system with single bit detection and correction hamming ECC's. The interdependencies between memory transaction requests across all four computation tiles, the chance for memory thrashing, and the memory coherency lead to this decision.

A failure mode and effects analysis (FMEA) was drawn, shown in Figure 7.14, to quantify the differences in expected failure when comparing a redundant memory array of

System	RadPC@Scale	**Potential**	FMEA Number	RadPC@Scale 1	
Subsystem	Memory Interface Controller	**Failure Mode and Effects Analysis**	Prepared By	Justin P Williams	
Component	External Memory PMOD PCB	**(Design FMEA)**	FMEA Date	12/22/2021	
Design Lead	Justin P Williams	Key Date 12/22/2021	Revision Date	12/22/2021	
Core Team	RadPC Team		Page	1 of 1	

Item / Function	Potential Failure Mode(s)	Potential Effect(s) of Failure	Sev	Potential Cause(s)/ Mechanism(s) of Failure	Prob	Current Design Controls	Det	RPN	Recommended Action(s)	Responsibility & Target Completion Date	Action Results				
											Actions Taken	New Sev	New Occ	New Det	New RPN
Microchip 23LC1024 1Mb SRAM SPI Module	Memory scrubber does not revoke access from tiles properly	Memory coherency issues	3	Cosmic radiation.	3	Voter.	2	**18**	Memory scrubber. Redundant Memory array. ECC.	Justin P Williams, May 20, 2021.					
Micron MT45V64M4 DDR SDRAM	Memory coherency / memory thrashing	Tiles accessing wrong regions of memory	4	Human error.	3	In lab testing.	2	**24**	Memory R/W arbitration block.	Justin P Williams, May 20, 2021.					
	Memory scrubber tile region R/W mismatch	Scrubber comparing incorrect memory partitions against one another	5	Human error.	4	In lab testing.	2	**40**	Memory R/W arbitration block.	Justin P Williams, May 20, 2021.					

Figure 7.14: FMEA Redundant Memory Array Versus Single DRAM with Increased ECC Performance

SRAMs against a single large DRAM memory shared between the computation tiles. While the advantages of using a large DRAM is a significant capacity increase on the systems data memory, it comes at the cost of memory access coherency. There must be a system in place to properly arbitrate between memory partitions belonging to their respective computational tiles. This arbitration must perform not only for computation tile R/W access, but the DMS R/W access as well. This arbitration must serially access a memory partition for each computation tile whereas with a redundant array, the memory controller is simply quadruplicated and synchronized such that each computation tile can access its own data memory in parallel to the others. This is a more true-to-form memory architecture to what RadPC@Scale aims to accomplish.

External Memory PCB Design

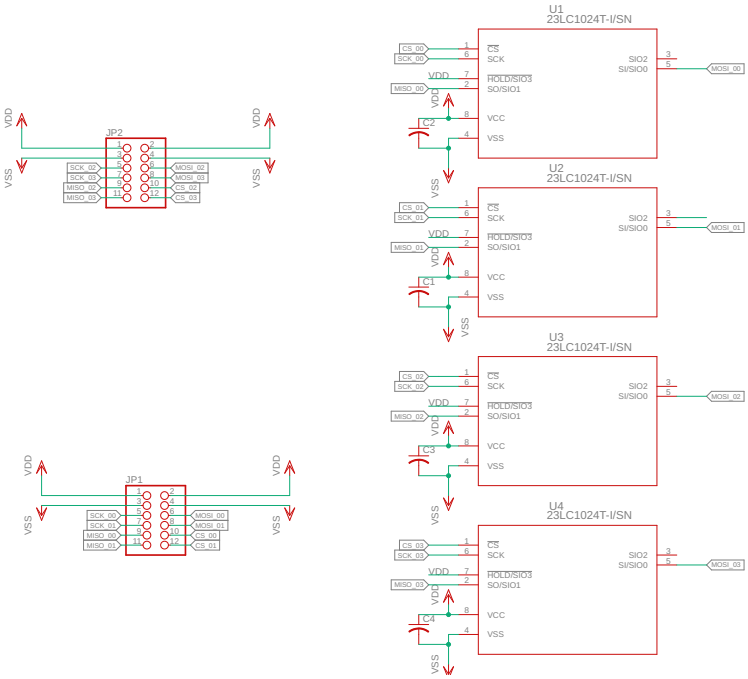
The external memory design for RadPC@Scale uses four Microchip 23LC2024 SRAM modules, one per each computation tile. The design complies with the Digilent PMOD connection standard, connecting to the bottom PC104 header of the RadPC@Scale computer board [9]. The PCB layout breaks the four IC's into two groups, where each group has their associated data lines routed to a PMOD connector. The PCB has two PMOD connectors that insert to the PC104 header mentioned above.

Pin 7 and pin 8 have been connected as per guidelines from the 23LC1024 datasheet [21]. Leaving pin 7 floating causes the IC to remain in a high-Z mode upon a read request. While the IC was in a high-Z mode, the read data during any transaction was faulty. This was discovered after the initial design and the read data from the individual chips often was not the same value across the four computation tiles.

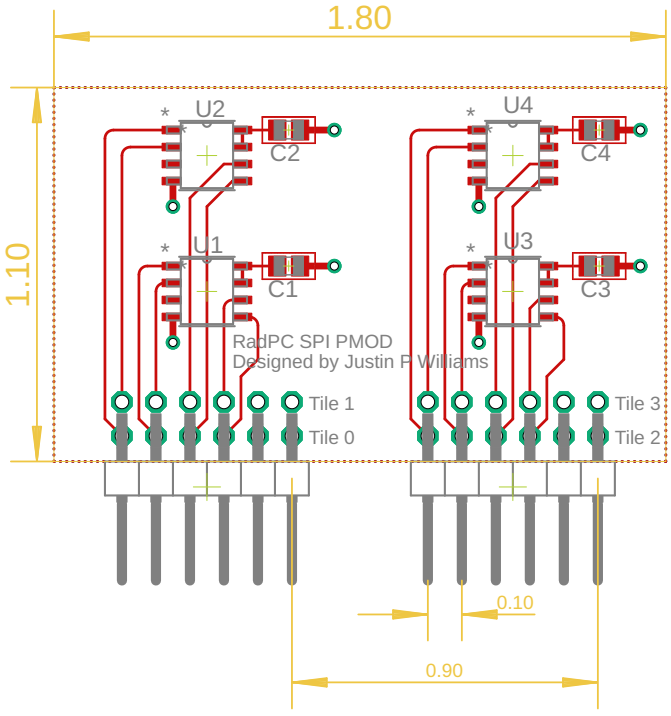
The design guidelines were based on an SRAM board design application note released by Cypress Corporation [5]. Specific guidelines followed include proper decoupling capacitor placement and value selection, routing traces on the top layer of the two layer board, and a dedicated ground plane on the bottom layer.

Payload Computation tile Requirements

This chapter shows the design solutions to satisfy the rest of the payload requirements listed in Chapter 4. The memory system described in this chapter satisfy implementing error-correction-codes (P2-2), data memory scrubbing(P2-3), fault injection through data memory (P2-5), and a suite of external memory for the computation tiles to communicate to (P2-9).



(a) SRAM PMOD Schematic



(b) SRAM PMOD PCB Design

Figure 7.15: External Memory PCB Schematic and Layout

RPI HARDWARE INTERFACE

This chapter introduces the hardware interface design between the RadPC@Scale payload and the Raven Thunderhead FCU. The interface is a Raspberry Pi 4B (RPi) that executes Python scripts implementing 0MQ socket libraries to interface with the FCU. When the FCU transmits a request to the RPi over an ethernet connection, that request is converted to a UART command that is sent to the RadPC@Scale board.

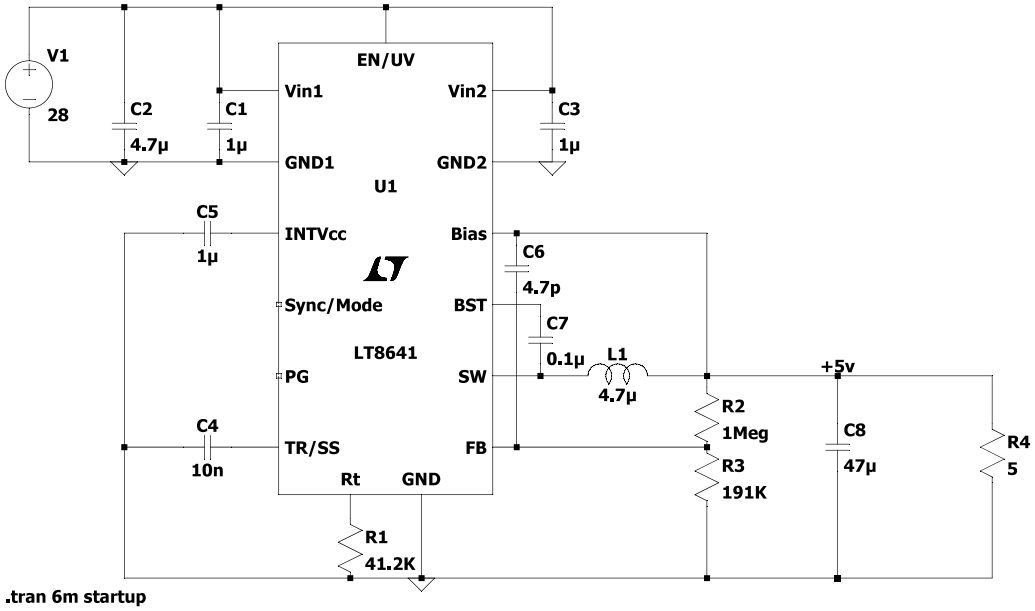
RPi Hat Design

The RPi Hat is a shield that routes UART signals to a RS422 transceiver chip for communication to the RadPC@Scale computer, regulates an input voltage of +28VDC to +5VDC to power the RPi, and distributes the input +28VDC to the RadPC Lunar board.

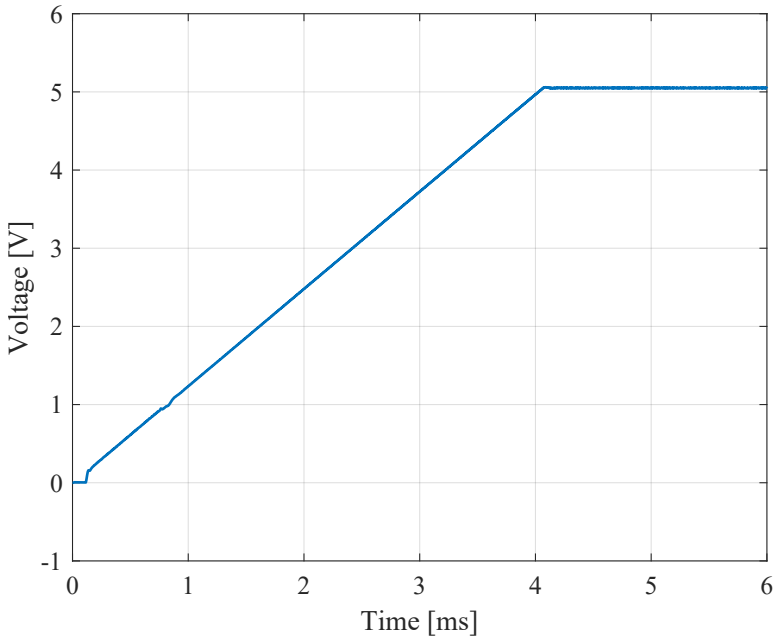
Preliminary Design and Simulation

The power regulation subcircuit uses an LT8641 switching regulator manufactured by Analog Devices. This device was picked to satisfy the power requirements of 5V at 3A outlined by the Raspberry Pi 4B datasheet [17]. The design was simulated using Analog Devices spice simulator, LTSpice, before its implementation on a PCB. The simulation schematic is shown in Figure 8.1a. The outputs of the simulation were run using a 6[ms] simulation time depicted in Figure 8.1b. The design was modeled from the 5V 3.5A typical application design from the LT8641 datasheet on page 22 [8].

After the simulations verified a desired output, the design was then translated to an Eagle PCB design. The first subcircuit of the design was the simulated power block. The schematic view of this circuit is show in in Figure 8.2. Key design notes are listed in the upper right of Figure 8.2 that came from the recommendations of the datasheet [8].



(a) LTSpice 5V Regulator Simulation



(b) LT8641 5V Output Simulation Profile

Figure 8.1: LT8641 Schematic and Simulation Outputs

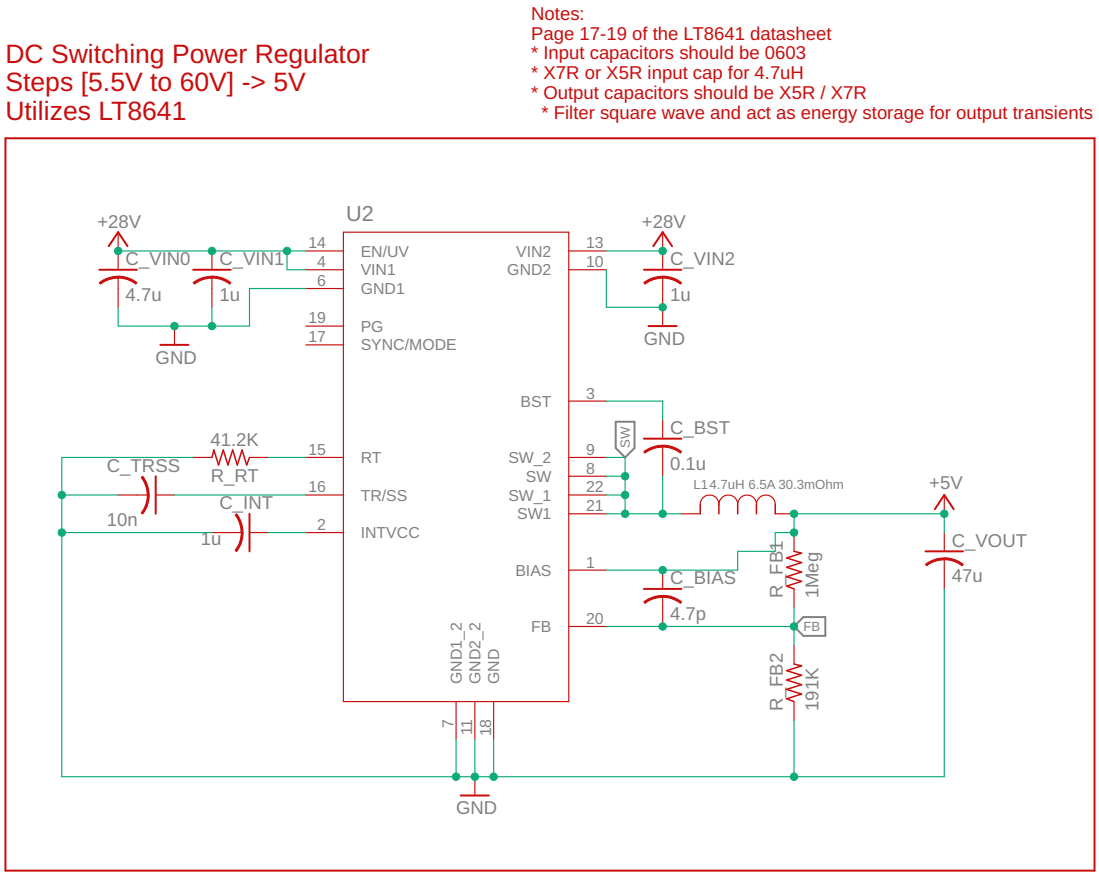


Figure 8.2: LT8641 Switching Regulator Eagle Schematic Design

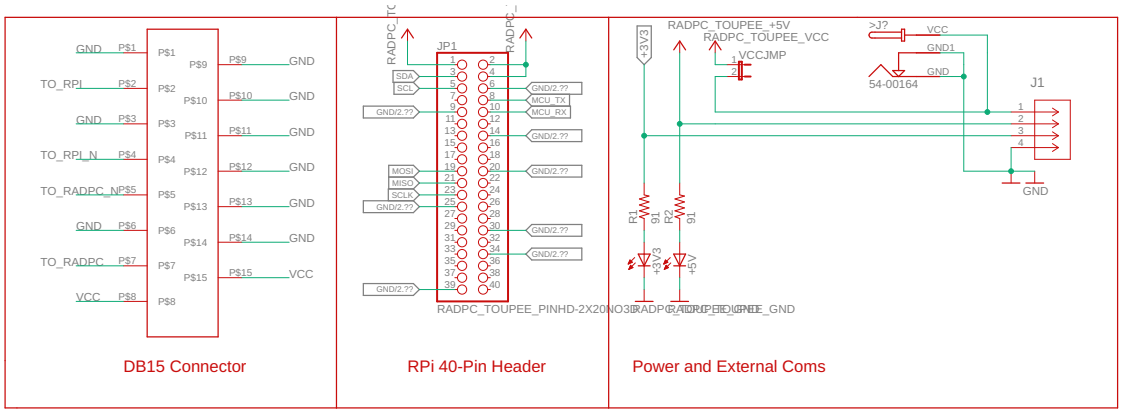


Figure 8.3: RPi Hat Data Line Routing

The RPi hat design shown in Figure 8.3 takes on the same form factor as the RPi where the 20-pin header on the hat design can mount to the 20-pin header of the RPi. This header routes the UART TX/RX pair as well as the 5V and GND power traces to the RS422 transceiver subcircuit, respectively [17]. This design was translated to an Eagle PCB design. The +28VDC power input power, a common ground, and the RS422 differential pair signals are routed to a DSUB15 (DB15) connector. The RPi circuit is connected to the RadPC computer through the DB15 connector seen on the RPi hat. A physical wiring harness was created to route the RS422 twisted pairs and the +28VDC power and ground lines.

The RS422 subcircuit is illustrated in Figure 8.4. This circuit is directly matching what is on the RadPC@Scale computer board. Therefore, from both the instrumentation microcontroller onboard RadPC@Scale and the RPi, each of them communicate to one another via UART protocol.

Implementation and Measurements

The successful implementation of the power regulation circuitry shown via the Analog Discovery 2 oscilloscope measurements depicted in Figure 8.5. The vertical scaling of the digital scope is set to $1[\frac{\text{mV}}{\text{div}}]$. It can be seen that on this measurement window that the

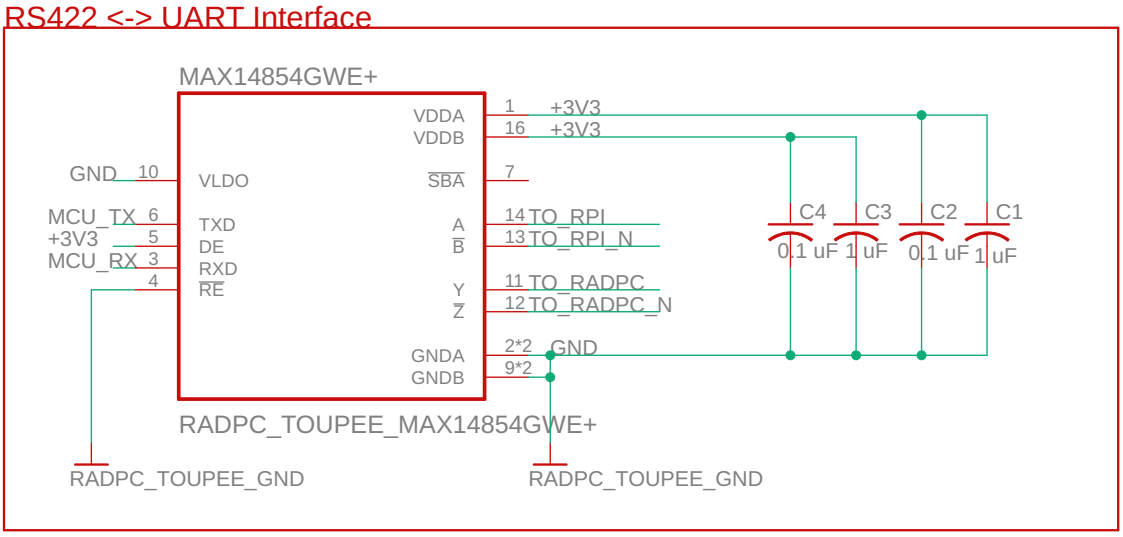


Figure 8.4: RPi Hat RS422 Subcircuit

output voltage has a voltage ripple of approximately $1[\frac{\text{mV}}{\text{div}}]$ centered at around 5.059[V].

RPi SW Design

The software designed implements a communication bridge that transcodes instructions that have been transmitted from the ground station to the FCU into a UART command that's then sent to the RadPC computer. After it has transmitted the instruction received from the FCU to the RadPC platform, it awaits telemetry packets to come back over the UART line and stores them into a buffer which is then sent from the RPi to the FCU.

Figure 8.6 depicts the high level flow of when a request is sent from the ground station through Raven's Iridium API, and how it is pipelined throughout the payload data structure. The top-level diagram of Figure 8.6 depicts the general communication data path between the ground server and the payload. The bottom left diagram of Figure 8.6 characterizes the data flow of when the ground station makes a request. The bottom right diagram characterizes the data flow of when the RadPC computer is creating a response.

The full source code for the 0MQ bridge is listed in appendix A. This code is

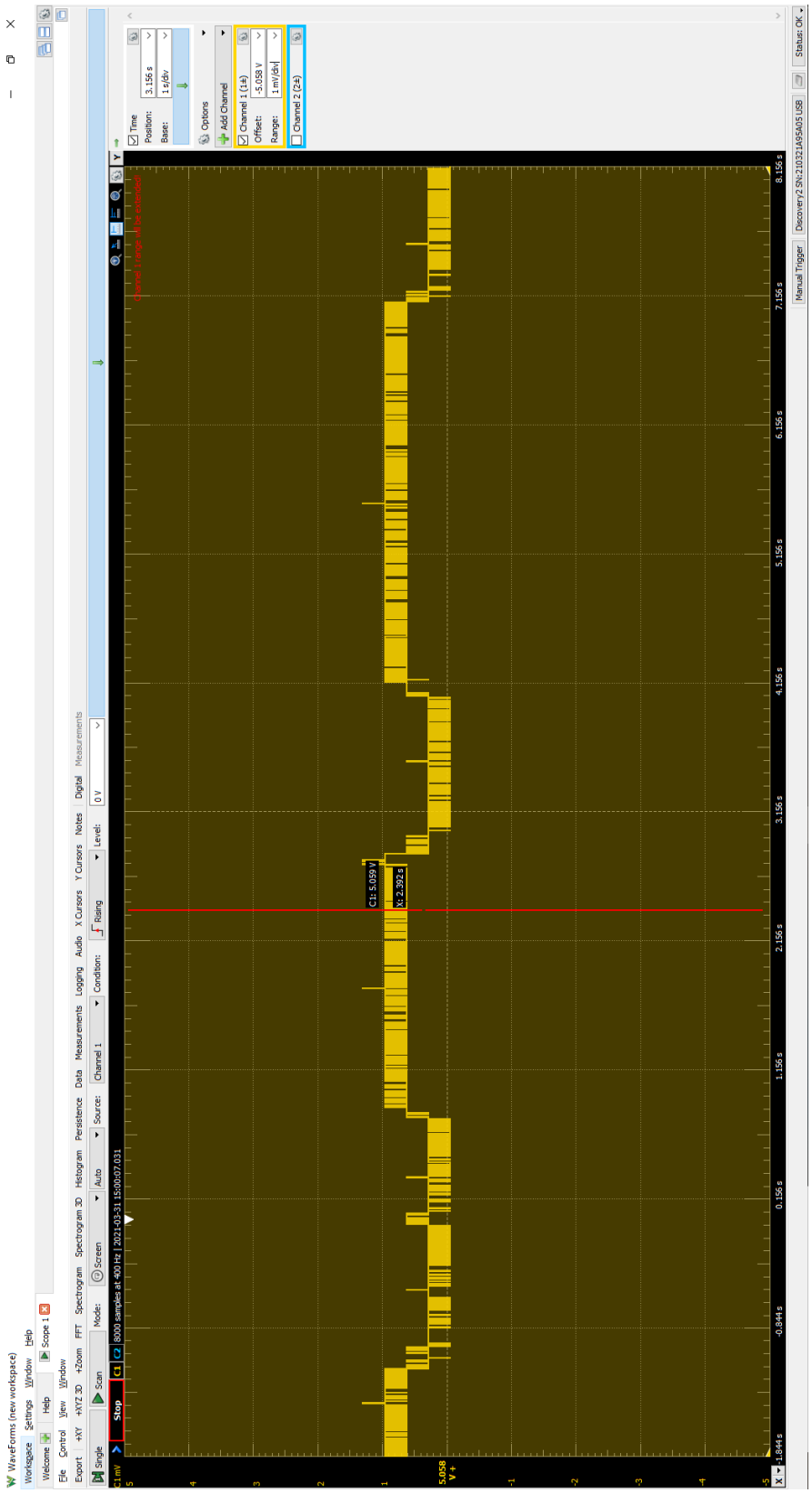


Figure 8.5: Oscilloscope 5V Regulator Measurements

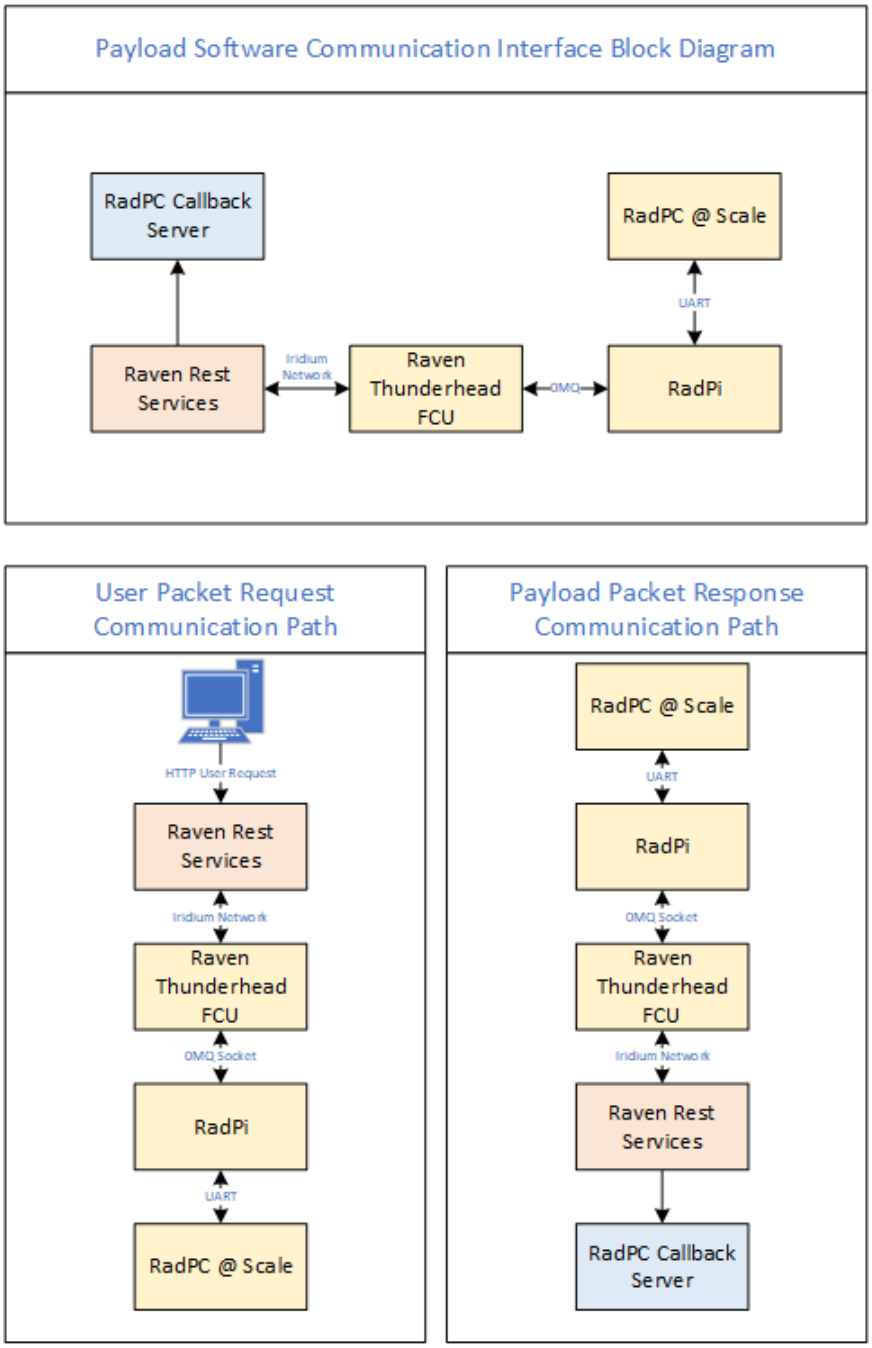


Figure 8.6: Software Interface Request / Response Flow

implemented to begin execution on power-up of the device.

Ground Station

This section depicts the front-end of the server that was used to create packet requests through Raven's REST Callback API, the organization and layout of where packets were stored, and the user interface. This code and server was developed by a fellow colleague, Zachary Becker, with given requirements/specifications that was provisioned. Together, we integrated the hardware packet telemetry to the RPi code such that it would properly store the data.

Packet Filter Menu

The packet filter menu allows the user to filter through date ranges to validate that the data is coming back over the Iridium network. This menu was used especially during flight to monitor that the packets were coming in at the constant five minute cadence that they were being requested at. Figure 8.7 shows the available date ranges to filter packet data from, the additional payload filter dropdown, and each packet entry as a result of the filter inputs. Figure 8.8 depicts a packet during the second flight from Payload 2. In each packet entry of the packet menu, the user can see packet data that is parsed into relevant sections on the right-hand side. It begins with packet header, and goes down through the relative FPGA faults detected and recovered. On the left side, there is data that is parsed from the telemetry header provided by Raven in their packet telemetry. This includes payload number, latitude and longitude, an image of the location on a Google Map, altitude, and the message that is encoded from the payload.

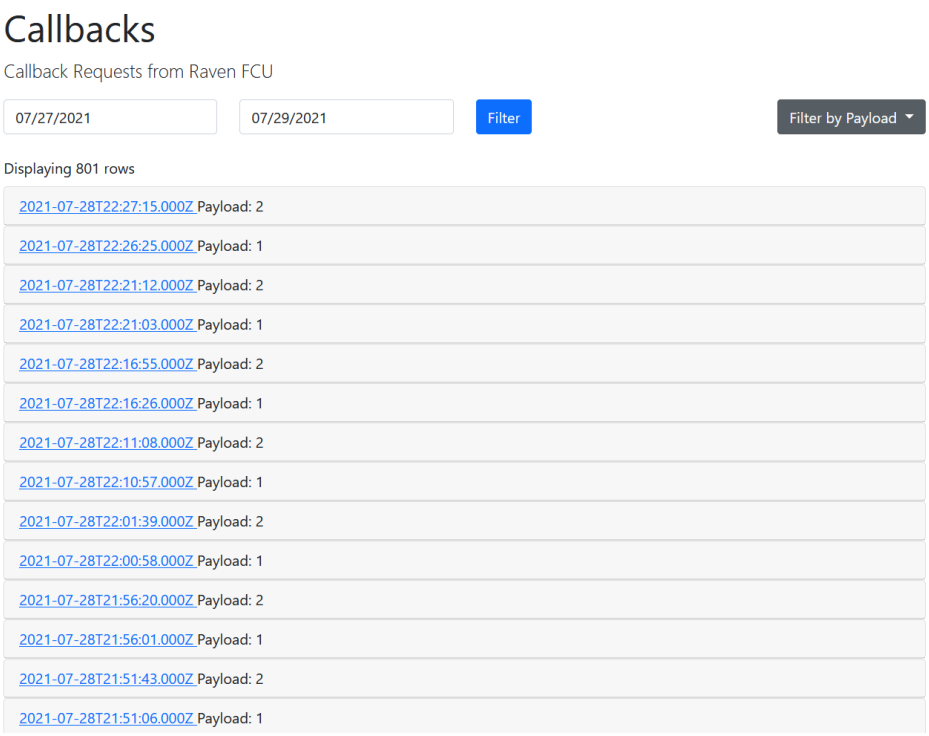
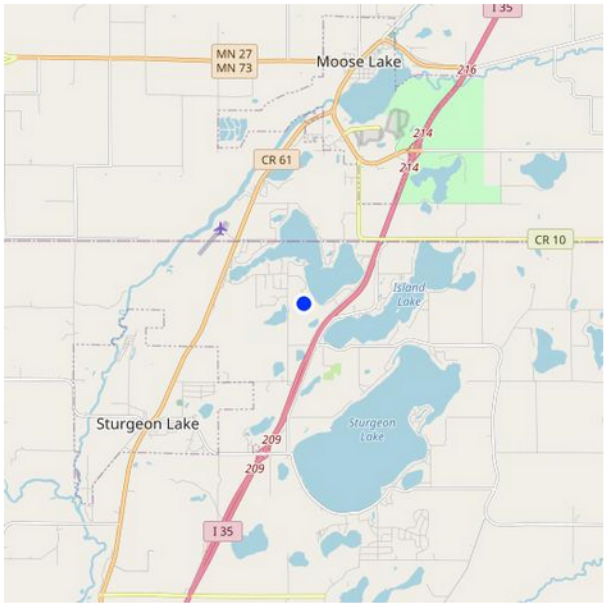


Figure 8.7: Packet Filter Menu - Top Level

Displaying 1110 rows

2021-09-24T16:25:33.000Z Payload: 2

2021-09-24T16:19:38.000Z Payload: 1



paynum
1

messageReceivedTime
2021-09-24T16:19:38.000Z

flightState
Operable

latitude
46.4054554

longitude
-92.7781383

altitudeM
22871.211

message (base64)
TW9udGFuYSBTdGF0ZSBSYWRQQyBGUEdBAEo8BmIJPAtjDH8H/w
RECp8A3AgPDckLIQKxBP4DQ+7urgACwwLDAsMCwgBmAAAAAA
AAAAAAZgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABmAAA
KcgsLAAAAAAAEIgAAAAAAACvzAAA=

message (hex)
Montana State RadPC
FPGA004a3c0662093c0b630c7f07ff04440a9f00dc080f0dc90b2102

header : Montana State RadPC FPGA
1VV : 0.997314453125
1VC : 0.05771484375
1.8VV : 1.7791748046875
1.8VC : 0.07810058593750001
2.5VV : 2.498779296875
2.5VC : 0.02666015625
3.3VV : 3.319091796875
3.3VC : 0.00537109375
5VV : 5.03662109375
5VC : 0.17231445312500002
28VV : 27.822265625
28VC : 0.0168212890625
mcu_temp : 32.01171875
fpga_temp : 65.446650390625
tile_00_cnt : 14
tile_01_cnt : 14
tile_02_cnt : 14
tile_03_cnt : 14
fpga_port_out_vote_cnt : 174
fpga_port_in_vote_cnt : 0
tile_00_fault_cnt : 707
tile_01_fault_cnt : 707
tile_02_fault_cnt : 707
tile_03_fault_cnt : 706
sem_fault_cnt : 102
tile_00_inj_cnt : 0
tile_01_inj_cnt : 0
tile_02_inj_cnt : 0
tile_03_inj_cnt : 0
sem_inj_cnt : 102
tile_00_chkpt_lag : 0
tile_01_chkpt_lag : 0
tile_02_chkpt_lag : 0
tile_03_chkpt_lag : 0

Figure 8.8: Packet Filter Menu - Sample Packet

Packet Visualization

The ground station server also provides packet visualization for the user. The visualization tool allows a user to select an input data range, the payload number, and a field for which they'd like to see displayed. Figure 8.9 depicts the measured 5V rail from the RadPC payload instrumentation over a date range of 09/22/2021-09/29/2021. The selected value is from the Chart Select drop down and has the designated 5VV rail. Note that the "VV" characters represent the 5V rails corresponding voltage measurement. Were it to say 5VC, it would be the 5V rails corresponding current measurement.

Requirement Fulfillment

The RPi Hardware interface design satisfies requirement P4-1 via the implementation of our ground station which creates REST API callbacks. The RPi hardware satisfies requirement P4-2 via the implementation of the 0MQ python scripts seen in Appendix A. The online ground station server satisfies requirements GS-1 through GS-3. The last requirement is being assembled as documentation on the implementation and usage of the scripts provided in Appendix A as well as the back end of the ground station server shown in the earlier sections.

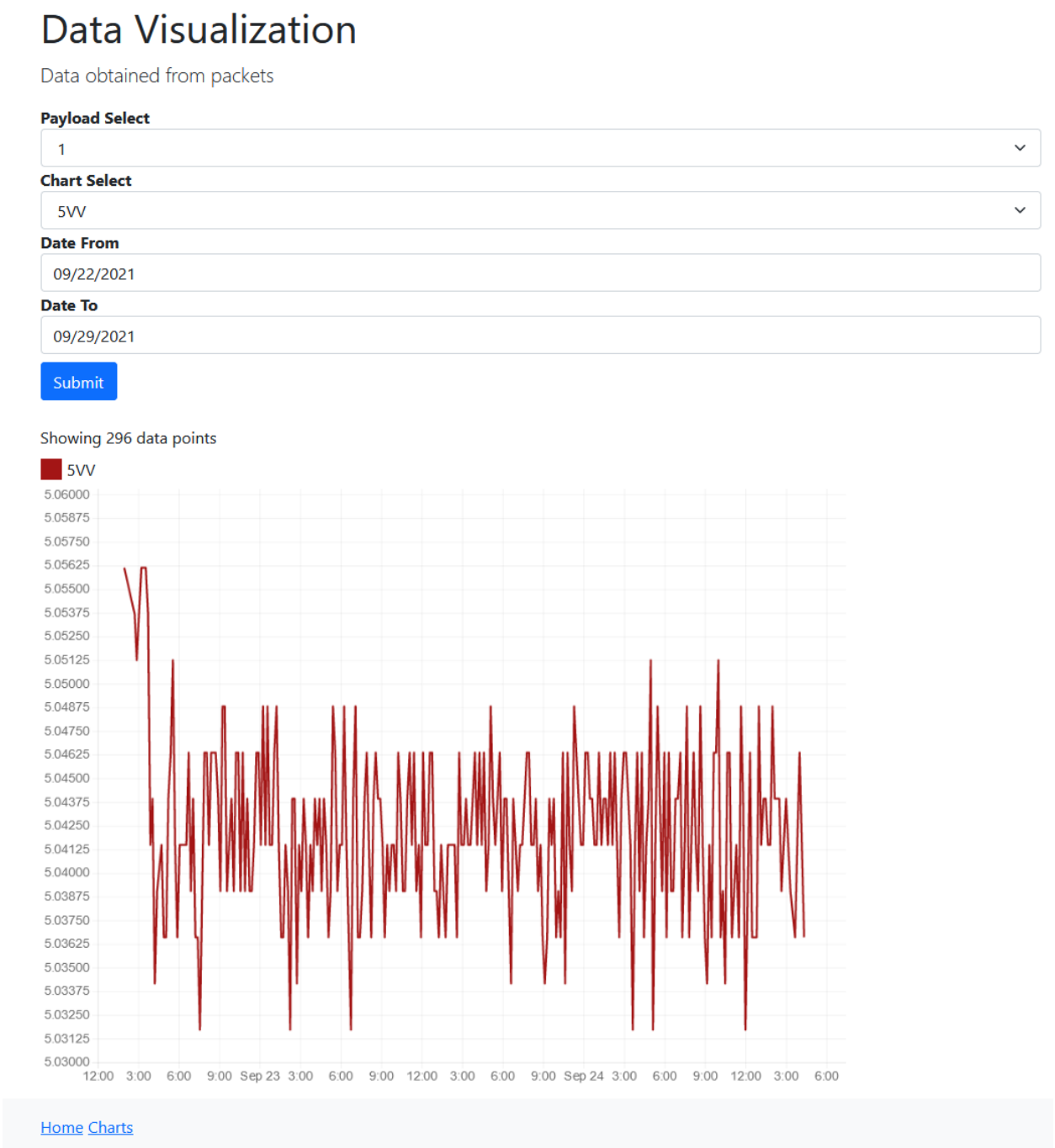


Figure 8.9: Packet Chart Visualizer - Payload 1 5V Rail

RESULTS

The RadPC computer assembles data packets on a 10-second cadence. The ground station server requested packets on a 5-minute cadence, with each payload offset from one another. The data plotted in the following figures is the whole data set that was post-processed after payload retrieval.

Packet Structure

Every packet has a 128-byte structure. Beginning with a packet header, then moves into power data for each voltage rail. After the power data, it logs pertinent FPGA data. Table 9.1 shows the packet data descriptions, each field's length, and its location.

The primary data fields this paper will discuss from Table 9.1 are the power monitors, temperatures, Computation Tile Fault Counts, CMM Fault Counts, CMM Injection Counts, CMM Correctable Faults, MEM Scrubber Outputs, and Mission MCU Reset Counts.

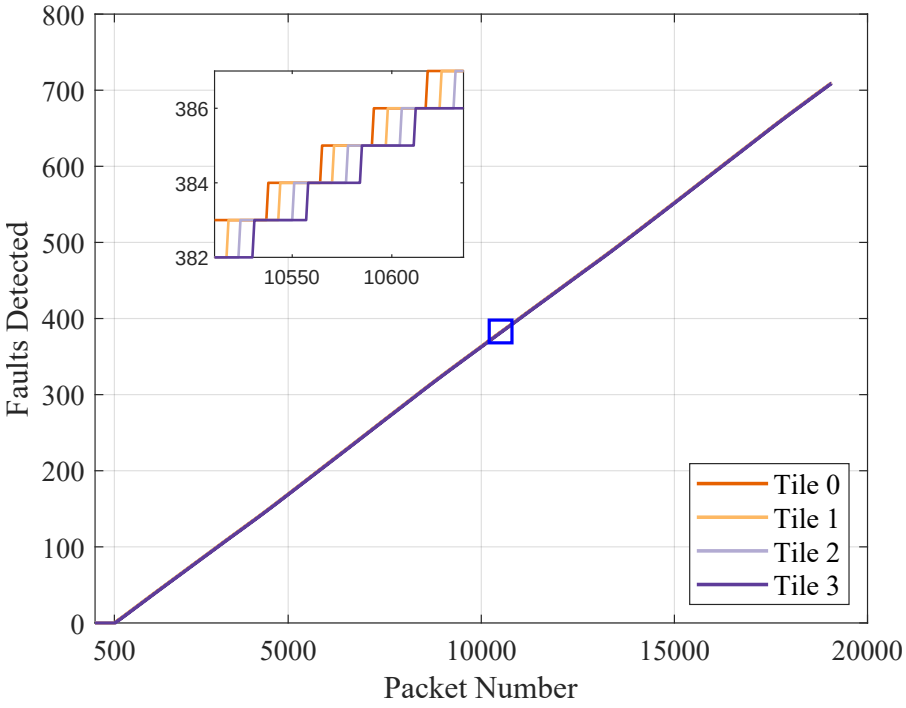
FPGA Data Plots

This section depicts all data fields related to the FPGA computer during flight. It is divided out into subsections relevant to different data fields.

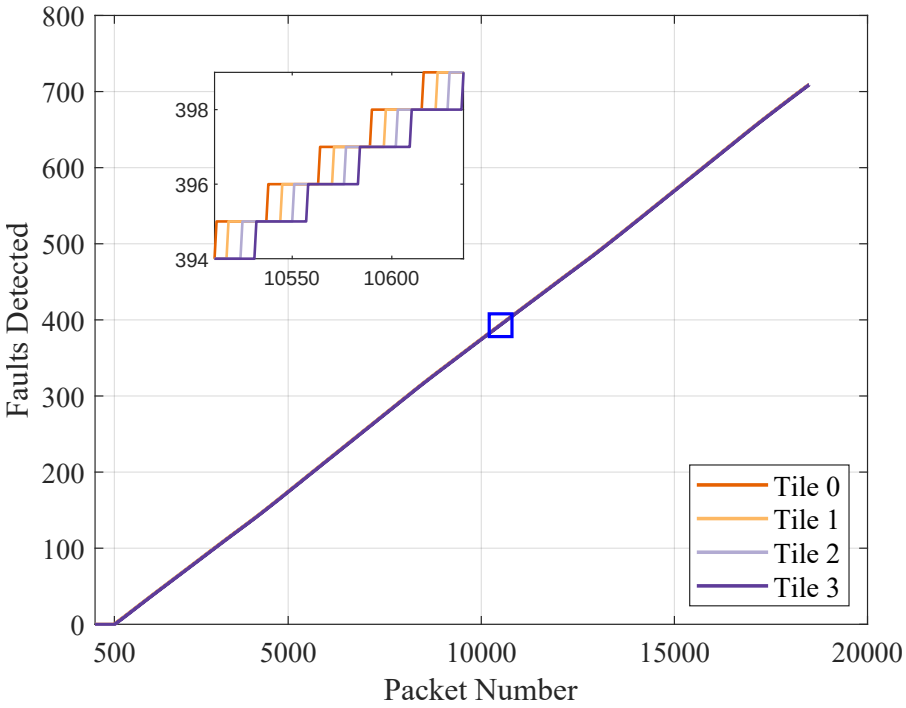
It can be seen in the exploded view highlighted by the blue box in Figure 9.1a that the fault counts increment in a round robin fashion. This is implemented by design and begins with a fault injection on Computation Tile 0, then Computation Tile 1, and so on and so forth. The Stormbreaker block stands between the computation tile memory control signals and the memory controller, illustrated in Figure 7.1. Due to the inherent design of the fault injection, the system did not detect any external radiation faults. If it did, there would have been additional offsets between the number of faults detected between computation tiles. Faults injected on the second payload behave similarly to the first, shown in Figure 9.1b.

Field Length (bytes)	Field Description	Packet Location
24	Packet Header	0-23
3	Packet Number	24-26
4	1.0V Power Rail	27-30
4	1.8V Power Rail	31-34
4	2.5V Power Rail	35-38
4	3.3V Power Rail	39-42
4	5.0V Power Rail	43-46
4	28.0V Power Rail	47-50
2	MCU Temperature	51-52
2	FPGA Temperature	53-54
2	FPGA Computation Tile Count	55-56
1	FPGA Voter Outport	57
1	FPGA Port In (Unused)	58
8	Computation Tile Fault Count	59-66
2	CMM Fault Count	67-68
8	Computation Tile Injection Count	69-76
2	CMM Injection Count	77-78
8	Computation Tile Checkpoint Lags	79-86
8	Computation Tile correctable Faults	87-94
8	Computation Tile Uncorrectable Faults	95-102
2	CMM Correctable Faults	103-104
2	CMM Uncorrectable Faults	104-105
2	MEM Error Correction Codes	106-107
6	MEM Scrubber Outputs	108-114
2	Mission MCU Reset Count	115-116
1	Last Communication Request	117
6	Reserved	118-123
2	CRC Codes	124-125
2	End Packet Symbol	126-127

Table 9.1: RadPC Telemetry Packet Structure



(a) Payload 1 - FPGA Computation Tile Fault Counts



(b) Payload 2 - FPGA Computation Tile Fault Counts

Figure 9.1: Packet Faults Detected on both Payloads

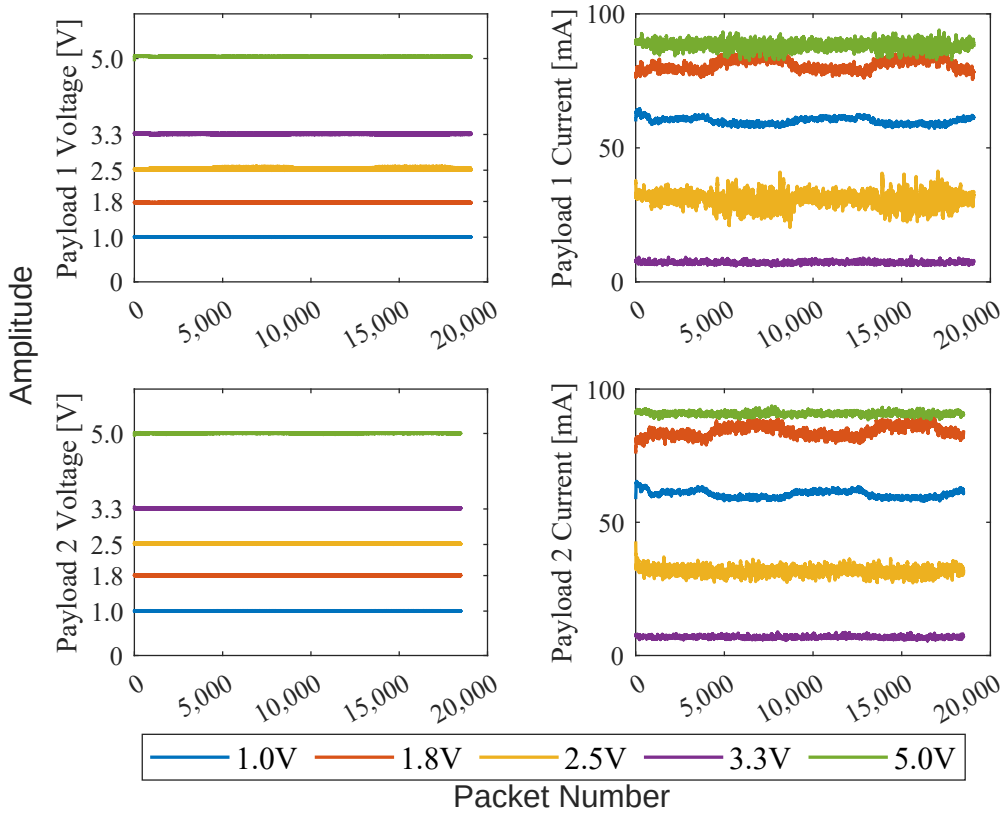


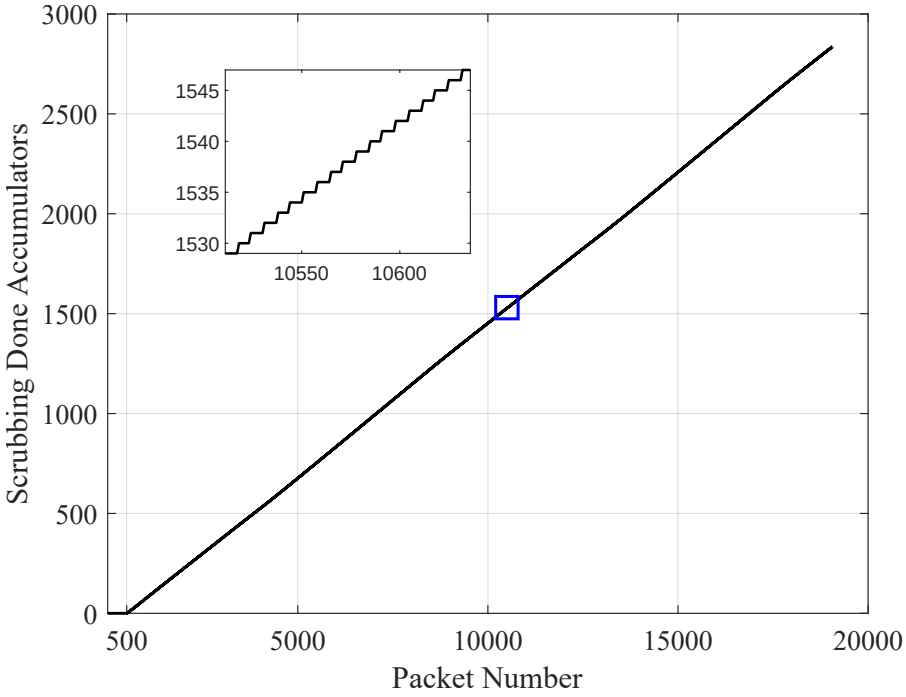
Figure 9.2: Payload Power Plots

Power Monitoring

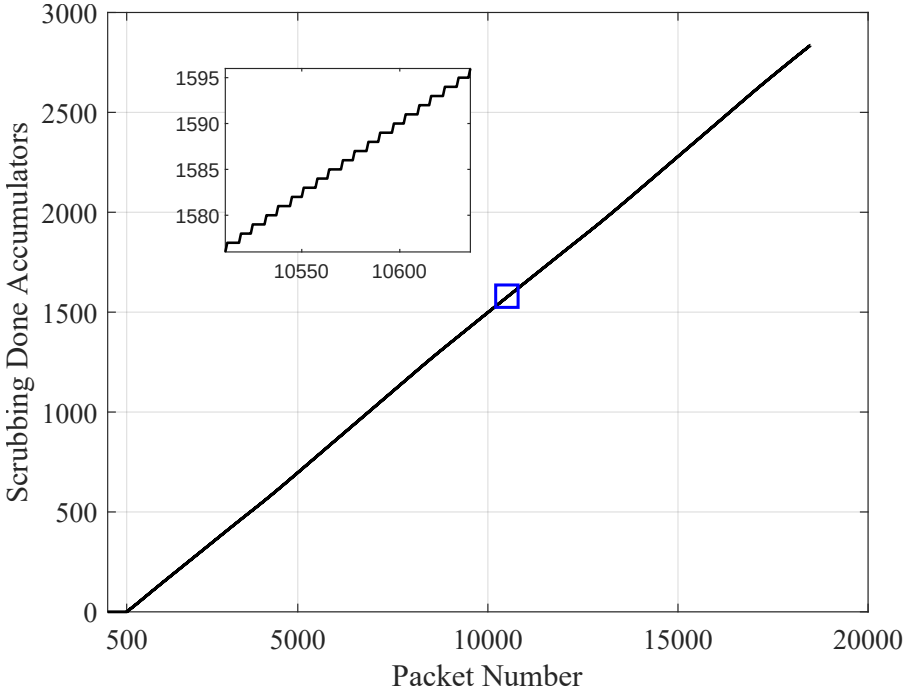
The RadPC Computer implements an ADC measurement scheme for monitoring the voltage supplies. The computer has 5 power supplies that feed the downstream FPGA and MCU. Notice in Figure 9.2 that the voltage rails follow the desired voltage levels. The depicted current draw is a moving average, with a window width of 32, as the current requirements of the system varied depending on what was occurring at the system level. Similar to the power plots from Payload 1, Payload 2 behaves in a similar manner. The power monitoring shows that the payloads were in good health during flight.

Data Memory Scrubber

This section depicts the DMS values throughout the flight duration. The data scrubber



(a) Payload 1 - Data Memory Scrubber Data Logs



(b) Payload 2 - Data Memory Scrubber Data Logs

Figure 9.3: Data Memory Scrubber Triggers

data depicted in Figure 9.3a and Figure 9.3b are the accumulations of each time the voter system triggered a data memory scrub. The number of scrubs is modeled by equation 9.

$$S = F_1 + F_2 + F_3 + F_4 \tag{9.1}$$

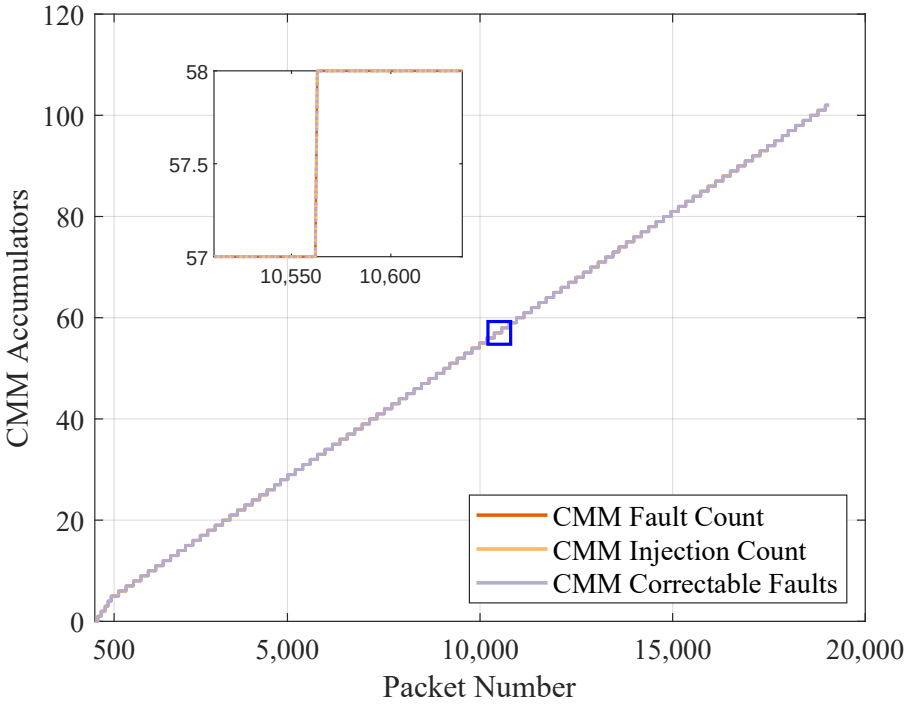
Where S is the number of scrubs that has occurred, and F_x is the accumulated faults on any given computation tile x. As shown in Figures 9.1a, 9.1b, and 9.3a, the faults did not begin to be introduced until around data packet 500. This was by design as Raven flight operations crew incorporate tests for the balloon launch that include repeated power cycling of the payloads. To avoid data offsets that needed data post-processing due to the flight operations power cycles, a delay of approximately 20 minutes was introduced prior to injecting faults.

FPGA CMM

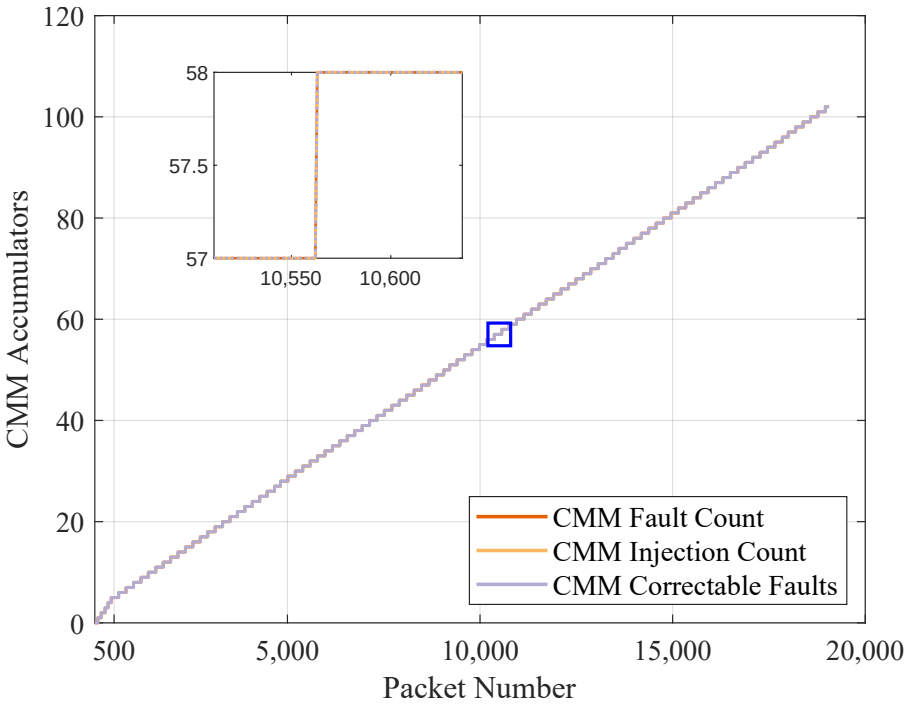
Recall the CMM IP can receive commands via a serial interface to inject errors. The foundation of this IP is to both manually inject faults and detect faults caused by external radiation and then recover from these faults. As denoted on Figures 9.4a and 9.4b, the injections, faults detected, and faults recovered were all detected by the monitoring MCU at the same time and are all overlaid onto one another. Due to the faults detected being the same as the faults injected, it can be seen that there were no faults caused by cosmic radiation.

Post Flight Processing

This section discusses the verification of the ECC codes implemented in each packet, as well as the CRC check on the transmission of each packet. There are scripts within Appendix B that depict code that will verify, over the full range of packet data, the voted



(a) Payload 1 - CMM Data



(b) Payload 2 - CMM Data

Figure 9.4: Configuration Memory Monitor Logs

data output against the corresponding ECC code. These scripts will also verify that there were no transmission faults that occurred during packet transmission. Faults could occur due to noise transmission lines on a PCB level, or through the Iridium network.

Recall in Table 9.1 that each voted value on field 57 has a corresponding ECC encoded value on Field 106-107. However, due to the restricted number of digital I/O and the physical PCB layout of the RadPC computer, only computation tile 2's ECC values were routed to the packet telemetry. In other words, the number of faults detected on computation tile 2 will equal the number of times that there is a mismatch between the voted output and the ECC value within the packet telemetry. The scripts in Appendix B counted 709 mismatches. Again, the number of times that the voted output in packet telemetry did not encode to what was listed in the ECC field is equal to the number of faults that were detected on computation tile 2, which is 709. The CRC codes did not detect any transmission errors for packets.

CONCLUSION

The experiment carried out with the payloads presented in this paper, named RadPC@Scale, has demonstrated the performance of a cost-effective, COTS, radiation tolerant space computing system. Through its suite of fault detection and mitigation strategies, it has shown that it will perform under harsh space environments by recovering from faults, injected or induced by radiation. While this flight has shown that there were no faults induced by radiation onto the payloads, the RadPC@Scale architecture still maintained healthy operation as depicted by the data packets shown above. Over the course of the 50-hour flight, the memory experiment injected 2836 faults and successfully recovered from each one.

While the system maintained ideal operation during its flight, there are aspects of the experiment that could be improved. Namely, the rate at which faults were being injected into the system. The fault rate was straining the system such that if a fault was induced via cosmic radiation onto any of the SRAM memory chips but a manually injected fault was detected, the system would scrub and therefore erase the record of the fault induced by radiation. Therefore, reducing the rate at which faults are injected should be a change for the next iteration of this high-altitude ballooning experiment.

Another aspect that was not performing at ideal conditions was some of the RadPC@Scale instrumentation circuitry. Particularly, the hamming code monitoring in the data packets. Due to physical I/O constraints on the instrumentation microcontroller, there is not enough GPIO lines to route all four computation tiles' hamming codes to monitor in parallel. This lead to the packet fault trend where every time computation tile 2 had a detected fault, only then would the packet's hamming codes reflect a fault that could be correlated back to the voted output. On a future iteration of this experiment, it would be prudent to route certain (or all) of the instrumentation data to a serial interface to the microcontroller rather than

over a parallel interface.

The limitations of the instrumentation used in the RadPC@Scale experiment made it difficult to identify whether or not we can confidently say that we were or weren't struck by ionized radiation. This is in part due to the slow sample rate at which packets are assembled (10 second cadence), the aforementioned shortage of parallel monitoring, and the rate at which faults were injected caused the system to repair out any induced faults before the system could detect them. The Voter would not detect any radiation faults in data memory that it is not actively reading from. While this is not an issue during runtime, it is an issue when we are attempting to instrument and verify the systems health and recovery process.

However, this experiment also exhibited many strengths of the RadPC@Scale architecture and the FCU interface. The FCU interface is a suite of hardware and software that can be used for many missions to come with Raven Aerostar. The work put into this interface has allowed it to be adaptable for many more high-altitude ballooning missions for more technical stress tests of the RadPC@Scale hardware. The ability to send flight data from the payloads during flight allowed for real-time health monitoring on the ground. If the packets indicated that the system was not operating at full capacity, a command would have been sent over the interface to reset the system to maintain operation. The mechanical interface is also in specification to Raven for future flights. The combination of a full experiment interface, both electrical and mechanical, to the Raven FCU balloon makes for future RadPC experiments very easy to do with minimal interface engineering.

The workflow that I've implemented for future FPGA development will be instrumental to accelerated RadPC IP design and implementation. Vivado and MATLAB have streamlined interfacing for optimal HDL development, allowing an engineer to design IP in MATLAB Simulink, and then directly export it to a library that Vivado actively uses in its own project flow. This allowed me to create modifications in MATLAB Simulink, recompile projects in Vivado, and then test the system without having to handwrite any

HDL. Incorporating this design flow in the future will accelerate any new grad students or engineers comprehension of the RadPC architecture.

FUTURE WORK

The next major step forward for the RadPC@Scale platform is to begin integration of DRAM into the system. Preliminary steps towards this include partitioning the memory, creating an arbitration block, routing the existing Hamming ECC encoder/decoder, and testing/verification. DRAM allows the system to include a much higher capacity external data memory source.

The SPI interface that I implemented for the SPI SRAM 23LC1024 modules was a big step forward for serial I/o. However, these SPI interfaces are not interfacing to external circuitry that is not tile-specific. That is, each SPI interface exclusively talks to a device that is part of the internal architecture. A preliminary step towards incorporating more Serial I/O would be to create a supervisor digital circuit that compare the data being sent from the RadPC@Scale computer to an external device, as well as fanout any incoming data from external devices to the RadPC@Scale platform. The existing SPI interface would be a fantastic block to develop this supervisor block as it has already been verified and tested for interfacing to external devices.

As of now, the voter exists as a Finite State Machine. This is a sequential logic circuit within the fabric of the FPGA, and is thus susceptible to SEEs. Creating a combinational logic based voter system would assist in the prevention of any future logical faults that could be incurred upon the system while increasing performance. This proposed circuit should be able to implement the existing functionality of the FSM-based voter. The major hiccup with this is that the Xilinx Microblaze IP softcore processors are blackbox designs. As such, we cannot easily implement the voter to checkpoint our processors without direct access to key registers such as the control register, status register, or the program counter.

To build on the idea of implementing a combinational logic based voter, incorporating a RISC-V glass-box processor would go hand-in-hand with the voter. The benefits of the

glass-box softcore processor include having absolute control over the entire system. Right now, the FPGA recovery mechanisms serve as external reactionary blocks to the Microblaze softcore. A glass-box softcore processor allows us to take a proactive approach to recovery, and make a more streamlined recovery procedure.

Finally, we could take the time to implement the existing SRAM modules onto the physical RadPC@Scale PCB. This is would require more hardware level testing over firmware/software testing within the FPGA and MSP430.

REFERENCES CITED

REFERENCES CITED

[1] Amort. Radiation-hardening by design phase 3, December 2012.

[2] G. Anelli, M. Campbell, M. Delmastro, F. Faccio, S. Floria, A. Giraldo, E. Heijne, P. Jarron, K. Kloukinas, A. Marchioro, P. Moreira, and W. Snoeys. Radiation tolerant vlsi circuits in standard deep submicron cmos technologies for the lhc experiments: practical design aspects. *IEEE Transactions on Nuclear Science*, 46(6):1690–1696, 1999.

[3] J.L. Barth, C.S. Dyer, and E.G. Stassinopoulos. Space, atmospheric, and terrestrial radiation environments. *IEEE Transactions on Nuclear Science*, 50(3):466–482, 2003.

[4] S. Caorsi and G. Cevini. Tdfem analysis of the scattering properties of shielding structures. In *2003 IEEE International Symposium on Electromagnetic Compatibility, 2003. EMC '03.*, volume 1, pages 288–291 Vol.1, 2003.

[5] Chakrapani. Sram board design guidelines, April 2016.

[6] Sah Chih-Tang. Evolution of the mos transistor-from conception to vlsi. *Proceedings of the IEEE*, 76(10):1280–1326, 1988.

[7] C. CLAEYS. *Radiation effects in advanced semiconductor materials and devices.* SPRINGER, 2010.

[8] Analog Devices. 65v, 3.5a synchronous step-down silent switcher with 2.5ua quiescent current, 2018.

[9] Digilent. Digilent pmod™ interface specification 1.1.0, 2017.

[10] Zhen Dong, Yanning Guo, Youmin Gong, and Chuanjiang Li. A high reliability radiation hardened on-board computer system for space application. In *2016 Sixth International Conference on Instrumentation Measurement, Computer, Communication and Control (IMCCC)*, pages 671–674, 2016.

[11] Ahmed O. El-Rayis, Tughrul Arslan, and Ahmet T. Erdogan. Addressing future space challenges using reconfigurable instruction cell based architectures. In *2008 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 199–203, 2008.

[12] Christian M. Fuchs, Nadia M. Murillo, Aske Plaat, Erik van der Kouwe, Daniel Harsono, and Todor P. Stefanov. Fault-tolerant nanosatellite computing on a budget. In *2018 18th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, pages 1–8, 2018.

[13] M. Garvie and A. Thompson. Scrubbing away transients and jiggling around the permanent: long survival of fpga systems through evolutionary self-repair. In *Proceedings. 10th IEEE International On-Line Testing Symposium*, pages 155–160, 2004.

[14] Hunter Hall, Matthew Chamieh, Jonathan Chu, Rohan Daruwala, Van Duong, Samuel Holt, Daniel Jeong, Michael Lally, Luan Nguyen, Ramin Rafizadeh, Peter Soon Fah, Kyle Weng, Christine Yuan, Chrishma Derewa, and Adrian Stoica. Utilizing high altitude balloons as a low-cost cubesat test platform. In *2020 IEEE Aerospace Conference*, pages 1–11, 2020.

[15] Alex Hands, Fan Lei, Keith Ryden, Clive Dyer, Craig Underwood, and Chris Mertens. New data and modelling for single event effects in the stratospheric radiation environment. *IEEE Transactions on Nuclear Science*, 64(1):587–595, 2017.

[16] Justin A. Hogan, Raymond J. Weber, and Brock J. LaMeres. A network-on-chip for radiation tolerant, multi-core fpga systems. In *2014 IEEE Aerospace Conference*, pages 1–7, 2014.

[17] Raspberry Pi (Trading) Ltd. Raspberry pi 4 model b, 2019.

[18] Chris Major, Brock LaMeres, David Klumpar, Larry Springer, John Sample, Skylar Tamke, Rubin Meuchel, Colter Barney, Annie Bachman, and Jake Davis. Overview of the upcoming radpc-lunar mission. In *2021 IEEE Aerospace Conference (50100)*, pages 1–7, 2021.

[19] Christopher M. Major, Annie Bachman, Colter Barney, Skylar Tamke, and Brock LaMeres. Radpc: A novel single-event upset mitigation strategy for field programmable gate array–based space computing. *Journal of Aerospace Information Systems*, April 2021.

[20] A. Makihara, M. Midorikawa, T. Yamaguchi, Y. Iide, T. Yokose, Y. Tsuchiya, T. Arimitsu, H. Asai, H. Shindou, S. Kuboyama, and S. Matsuda. Hardness-by-design approach for 0.15 /spl mu/m fully depleted cmos/soi digital logic devices with enhanced seu/set immunity. *IEEE Transactions on Nuclear Science*, 52(6):2524–2530, 2005.

[21] Microchip. 1mbit spi serial sram with sdi and sqi interface, 2012.

[22] Sanjeeb Mishra, Neeraj Kumar Singh, and Vijayakrishnan Rousseau. Chapter 9 - power interfaces. In Sanjeeb Mishra, Neeraj Kumar Singh, and Vijayakrishnan Rousseau, editors, *System on Chip Interfaces for Low Power Design*, pages 319–330. Morgan Kaufmann, 2016.

[23] James Rosenthal, Bryan Hayes, and Christopher Mertens. A silicon micro dosimeter for high-altitude measurements of cosmic radiation. In *2018 IEEE Aerospace Conference*, pages 1–7, 2018.

[24] M.J. Rycroft. Handbook of radiation effects: Holmes-siedle a. and adams l., 1994 479 pp., oxford science publications, £45 hbk, isbn 0-19-856347-7. *Journal of Atmospheric and Terrestrial Physics*, 57(13):1672–1673, 1995.

[25] Bhargav Shashidhara, Shrikant Jadhav, and Young Soo Kim. Reconfigurable fault tolerant processor on a sram based fpga. In *2020 IEEE International Conference on Electro Information Technology (EIT)*, pages 151–154, 2020.

[26] Charles Slayman. Soft error trends and mitigation techniques in memory devices. In *2011 Proceedings - Annual Reliability and Maintainability Symposium*, pages 1–5, 2011.

[27] L. Sterpone and M. Violante. Analysis of the robustness of the tmr architecture in sram-based fpgas. *IEEE Transactions on Nuclear Science*, 52(5):1545–1549, 2005.

[28] Skylar A Tamke. Optimization of error correcting codes in fpga fabric onboard cube satellites. In *MSU Master's Thesis*, 2020.

[29] Ramazan Uzel and Alime Özyildirim. A study on the local shielding protection of electronic components in space radiation environment. In *2017 8th International Conference on Recent Advances in Space Technologies (RAST)*, pages 295–299, 2017.

[30] Xilinx. Axi reference guide, July 2017.

APPENDIX: A

APPENDIX: A

This code is implementing the 0MQ socketing library that bridges the data interface between the RadPC@Scale computer and the Raven FCU. When the FCU receives a command from the groundstation, it passes that command through a 0MQ socekt to the RPi. The RPi then converts this command to a UART transmission to the RadPC@Scale computer. The RPi then listens for a response from the RadPC@Scale computer, and upon its response, pipes this data back over the 0MQ socket bridge to the Raven FCU.

Within the code itself, it is seen that there are multiple functions related to the description above. On line 27, the "uart_client" function is declared. This function handles the transmission of the FCU command and the reception of the RadPC@Scale response. The function "respond_to_fcusim" handles the transmission of the packet data received via the UART protocol back to the FCU.

```
1  #!/home/pi/envs/radpc_scale/bin/python
2
3  import zmq
4  import serial
5  import time
6  import argparse
7  import logging
8  from  multiprocessing import Process, Queue, Lock
9
10 import FCU_SWICD_PayloadMessage_pb2 as PayloadMessage
11 from protobuf_lib import getProtoBufMessage
12
13 logging.basicConfig(level=logging.DEBUG,
14     format='%(asctime)s [%(levelname)s] %(message)s',
15     filename='/home/pi/radpc_scale/sample.log',
16     datefmt='%Y-%m-%d %H:%M:%S')
17
18 DEBUG=False
19
20 if DEBUG:
21     SERIAL_PORT = '/dev/ttyACM0'
22     BAUD_RATE = 9600
23 else:
24     SERIAL_PORT = '/dev/ttyAMA0'
25     BAUD_RATE = 115200
26
```

```
27  def uart_client(rq, sq):
28      while True:
29          if not(rq.empty()):
30              radpc_msg = None
31              msg = rq.get()
32
33              try:
34                  #TODO: request different lengths of bytes depending on the command
                    ↪  request
35                  if msg == b'$':
36                      logging.info("CASE 1")
37                      ser.write(b'\x24')
38                      radpc_msg = ser.read(11)
39                  elif msg == b'"':
40                      logging.info("CASE 2")
41                      ser.write(b'\x22')
42                      radpc_msg = ser.read(128)
43                  else:
44                      logging.info("CASE 3: Unsupported Command")
45                      logging.info("Requesting Heartbeat as default")
46                      ser.write(b'\x24')
47                      radpc_msg = ser.read(11)
48
49                  if radpc_msg is not None:
50                      logging.info("Message from Queue: {}".format(msg))
51                      logging.info('UART RESPONSE:')
52                      logging.info(radpc_msg)
53
54                      # put RadPC message into send_queue for transmission back
                        ↪  through FCU
55                      sq.put(radpc_msg)
56
57              except Exception as e:
58                  logging.warning('error')
59                  logging.warning(e)
60
61  def listen_for_command(rq, sq, socket):
62      ''' RECEIVE FROM FCU '''
63      # receive data from zmq socket
64      msg = socket.recv()
65
66      # instantiate Protobuf object
```

```
67          FCUMessage = PayloadMessage.FCU_SWICD_PayloadMessage()
68          # load data from socket into Protobuf object
69          FCUMessage.ParseFromString(msg)
70
71          # Enter msg into Queue (To send to RadPC)
72          rq.put(FCUMessage.data)
73
74          # messages to STDOUT
75          logging.info("## Command Received from FCU")
76          logging.info("Google Protobuf Message: {}".format(msg))
77          logging.info("messageSentTS: {}".format(FCUMessage.messageSentTS))
78          logging.info("data: {}".format(FCUMessage.data))
79
80      def respond_to_fcusim(sq, socket ):
81          try:
82              #if not(sq.empty()):
83              radpc_msg = sq.get()
84
85              logging.info('respond_to_fcusim')
86              logging.info(radpc_msg)
87
88              logging.info('*** sending response')
89              protoBufMsg = getProtoBufMessage(radpc_msg)
90              protoBufMsg.SerializeToString()
91              socket.send(protoBufMsg.SerializeToString(), zmq.NOBLOCK)
92          except Exception as e:
93              logging.warning(e)
94
95      def get_cl_args():
96              parser = argparse.ArgumentParser(description='RadPC communication interlink
                ↪  (for use with Raven Thunderhead FCU and RadPC Lunar)')
97              parser.add_argument('fcu_ip', nargs='?', default="5561", help="FCU IP
                ↪  address")
98              parser.add_argument('fcu_port', nargs='?', default="10.1.3.7", help="FCU
                ↪  port")
99
100             return parser.parse_args()
101
102     if __name__ == "__main__":
103         # Parse command line arguments
104         args = get_cl_args()
105
```

```
106     # Serial
107     ser = serial.Serial(SERIAL_PORT, BAUD_RATE, timeout=1)
108     ser.reset_input_buffer()
109 
110     # OMQ
111     #source_endpoint = ":{}".format()
112     #fcu_address= "tcp://{}:{}".format(args.fcu_port, args.fcu_ip)
113 
114     port = "5562"
115     source_endpoint = "10.1.3.8:{}".format(port)
116     dest_endpoint = "10.1.7.95:{}".format(port)
117     fcu_address= "tcp://{};{}".format(source_endpoint, dest_endpoint)
118 
119     logging.info(fcu_address)
120 
121     context = zmq.Context()
122     socket = context.socket(zmq.PAIR)
123     socket.setsockopt(zmq.SNDHWM, 0)
124     socket.setsockopt(zmq.RCVHWM, 0)
125 
126     socket.connect(fcu_address)
127     # Thread-safe Queues
128     rq = Queue()
129     sq = Queue()
130 
131     Process(target=uart_client, args=(rq, sq, )).start()
132     #Process(target=listen_for_command, args=(receive_q, send_q, socket,)).start()
133     #Process(target=respond_to_fcusim, args=(send_q, socket, )).start()
134 
135     while True:
136         listen_for_command(rq, sq, socket)
137         respond_to_fcusim(sq, socket)
```

APPENDIX: B

APPENDIX: B

This section includes scripts that handle post processing of the ECC and cyclic redundancy check (CRC) codes from the flight data. Each script was run separately on the data packets to verify that a) the number of faults detected from the ECC codes matches the number of faults detected on that tile, and b) that there were no transmission errors caught from the CRC codes.

ECC Comparison Code

```
"""
Engineer:               Justin P Williams
                        Colter Barney
Date:                   10/25/2021
Description:            Compares input data and its encoded outputs from flight
↪  packets
                        to verified encoded outputs.
Company:                Montana State University
"""

import csv
import tkinter as tk
from tkinter import filedialog

def main():

    # Create file I/O dialog to select encoded map file path
    root = tk.Tk()
    root.withdraw()
    map_path = filedialog.askopenfilenames()
    p1_input_path = "p1_ecc.csv"
    p2_input_path = "p2_ecc.csv"
    root.quit()
    root.destroy()

    # Open files for encoded_map and output ECC data from payloads
    map_file = open(map_path[0],"r")

    # CSV reader to capture map path codes into a dictionary type
```

```
    map = {}        # initialize dictionary
    for line in csv.DictReader(map_file):
        map[int(line["INPUT_VALUE"], 16)] = int(line["OUTPUT_CODE"], 16)

    p1_cnt = ecc_flts(map, p1_input_path)      # Total faults (payload 1)
    p2_cnt = ecc_flts(map, p2_input_path)      # Total faults (payload 2)

    print("Total ECC mismatch (P1):\t" + str(p1_cnt) +
        "\nTotal ECC mismatch (P2):\t" + str(p2_cnt))

def ecc_flts(map, input_path):
    input_file = open(input_path, "r")      # open input file path
    eccs = csv.DictReader(input_file)       # Read in file to CSV dict
    cnt = 0                                 # Initialize fault mismatches

    prev_cnt = ""
    for ecc in eccs:
        if (prev_cnt != ecc["VOTED_OUT"]):
            if (int(ecc["VOTED_OUT_ECC"]) != map[int(ecc["VOTED_OUT"])]):
                cnt += 1

        prev_cnt = ecc["VOTED_OUT"]
    return cnt

if __name__ == "__main__":
    main()
```

CRC Check Code

```
__author__ = "Colter Barney, Justin Williams"

import crcmod as crc
import binascii
import tkinter as tk
from tkinter import filedialog

def main():

    root = tk.Tk()
    root.withdraw()
```

```
12
13      input_paths = filedialog.askopenfilenames()
14      root.quit()
15      root.destroy()
16
17      for input_path in input_paths:
18          input_file = open(input_path,"r")
19          output_file = open(input_path[:-4]+"_corrected.txt","w")
20          for line in input_file:
21              crc_fun = crc.predefined.mkPredefinedCrcFun("crc-16-mcrf4xx")
22              temp = ((binascii.hexlify(line[2:26].encode('utf-8')) +
23                  line[27:-5].encode('utf-8')).decode('utf-8'))
24              result = crc_fun(bytes.fromhex(temp))
25              if(result !=0):
26                  print("Packet: ", line, end="")
27                  print("CRC result: ", hex(result),"\n")
28              else:
29                  print(line, end="",file=output_file)
30
31  if __name__ == '__main__':
32      main()
```