Embedded Debugger

Project Testing and Acceptance Plan

Schweitzer Engineering Laboratories

Subham Behera, Daniel Ochoa, Howard Potter

October 24, 2023

TABLE OF CONTENTS

I. Introduction

I.1. Project Overview

This embedded debugger's goal is to be able to take snapshots of a system, as well as loading these snapshots onto a system from storage. Due to the nature of this task, it is crucial to test the functionality of this program, as errors could prevent snapshots from being loaded onto the system, or maintaining data integrity when creating snapshots. As this project's architecture separates system tasks, it is imperative to make sure these interactions are tested as well. Due to this nature, when different aspects of the architecture are implemented, CI will be observed - when code updates are committed, it will be tested with the rest of the system as is to ensure they interact accordingly. This observation may only begin once a more integrated system is complete, due to the unique integration constraints of this project. Additionally, with the scope of this project, CD will be observed - due to the targeted environment and development environment of this project, a deployable version of the software for its production environment will remain continuous as updates are completed, similarly with the caveat that an integrated deliverable is viable.

I.2. Test Objectives and Schedule

The objective of these tests will generally be to ensure that data integrity is kept intact during reading, loading and saving data. From the initial stages of being able to read a set amount of data from the system's stack to a full dump of the data in memory, integrity is critical for the success of this project. Automated and manual tests can ensure this functionality is kept. In regards to integration, full integration may not be possible until most implementations for each architecture are completed as integration testing will be done in an emulated environment instead of the targeted environment.

As the vitis IDE that is used as the development environment for this project, hardware emulation will be used for much of the development testing. This will also be used to debug the hardware when testing on the targeted environment, in addition to a serial connection to the targeted hardware. Unit testing will be done to test every component in an isolated environment to ensure its own functionality, using testing frameworks within vitis. Manual testing may be necessary, however, for some functionality, such as reading information from the stack of the system. These manual testing procedures will have to be documented.

The testing procedures will generate these major deliverables:
Firstly will be a set of unit tests that cover the Exception Detector, Snapshot Capture and Data Storage functionalities. Secondly will be a document covering our manual tests, the conduct for these tests, their justifications and expected results. The last deliverable will be through GitHub Actions for workflow.

I.3. Scope

These sections will cover our plans on how we intend to create and conduct tests for the debugger. It will not cover all test frames and inputs, as they cannot be determined at this point in the project's life. However it will cover the general planned testing process to ensure functionality of this project.

II. Testing Strategy

Project testing will compose of both automated unit testing, as well as manual testing done during development. While components of this project can be tested using unit tests, such as the Exception Detector, there is an anticipation for manual testing for other components, such as reading data from the CPU registers. Due to this, not every test can be completed for every component, however this should still allow for testing through a CI pipeline. Any component that requires manual testing can be well tested before integrating, and all others can be automatically tested by CI. Because of this, CI may not be observed at the beginning aspects of the project due to these unique constraints, however this will not remain true for the project's lifetime. Generally speaking, the testing process will be:

1. **Write code**: A feature for a component will be developed, and will continue development until it is believed to fulfill requirements of the feature.
2. **Determine test case**: For every functionality, it will be determined if this function will require manual testing, or automated testing.
    a. If manual testing: The developer will generate and justify a list of testing requirements for the function. This list will then be documented.
    b. If automated testing: The developer will generate test cases to test both expected values and edge cases, or other exceptional behavior.
3. **Run tests**: The developer will run the tests for that function within the testing environment, whether they be automated or manual tests.
4. **Fix issues**: If any test fails, the developer should address these failures. Repeat until all tests pass.
5. **Push code to remote**: The developer will push their code to a remote branch.
6. **Pull request to main**: The developer will make a pull request against the main branch. At this point another developer will perform a code review and approve of the commit.
7. **Merge branch:** The branch will be merged into main.

III. Test Plans

III.1. Unit Testing

For unit testing in the Exception Handler project, we will adopt a careful approach focused on isolating individual components such as the Exception Detector, Snapshot Capturer, and Data Storage modules. We will use a combination of manual testing and automated test frameworks tailored for embedded systems. The tests will cover core functionalities crucial for system stability, including real-time exception detection, snapshot capturing accuracy, and seamless data storage. Unit testing will be extensive, aiming to validate each component's behavior in isolation. We will employ mock programs to simulate hardware interactions, ensuring that our tests remain isolated and focused on specific functionalities. We will consider the unit testing phase complete only when all core functionalities related to exception handling have been thoroughly tested and validated.

III.2. Integration Testing

Our integration testing strategy will build upon the foundation laid during unit testing. Given the embedded nature of our system, integration testing will be approached iteratively, beginning with the integration of fundamental components like the Exception Detector and Snapshot Capturer. The testing process will include validating interactions between these components and ensuring that exceptions detected by the detector are accurately captured by the capturer. Each integration step will involve running existing unit tests and creating new ones to validate the integrated functionalities. Due to the unique constraints of embedded systems, full integration might not be possible until the system is deployed on the target hardware. Therefore, integration testing will involve closely simulating hardware interactions and evaluating the system's behavior in an environment that mirrors the actual deployment setting. Developer discretion will be crucial, allowing flexibility in integrating less amenable script structures while prioritizing the robustness and reliability of the overall system. Continuous validation and adjustments will be made throughout the integration process to guarantee seamless interactions between components.

III.3. System Testing

III.3.1. Functional testing:

The functional requirement most relevant to the user is the textual/graphical user interface. The best way to test this is to act like someone who has no idea about this project so the majority of cases and edge cases can be tested. This is as simple as when we are done with development, one of us will try to intentionally break the program by perhaps providing bad input. It depends on what type of interface will be used because a graphical one can have more bugs, but this will all be caught with the testing method mentioned.

III.3.2. Performance testing:

As this project is intended to be used in a specific environment, on the KV260 SOM, it is the only environment that will be tested on. On this hardware, it must function to its requirements, as well as have an acceptable operating time for loading snapshots to minimize system downtime - this can be measured with time, and be compared to an ideal time.

III.3.3. User Acceptance Testing:

For user acceptance, this project must at minimum fulfill all of the requirements, and be able to fulfill these requirements on the specific environment setup. These requirements include functionality to take a snapshot of system information, as well as functionality to load this snapshot to restore the core registers and associated memory to restore a system state. Once these requirements are demonstrated to be functionally fulfilled in the intended environment, this will pass user acceptance testing.

IV. Environment Requirements

For a successful test, it is necessary to have the debugger on and running the software. The software should be able to boot up a textual or graphical user interface, whichever we decide on later, to accept some user input.

V. Glossary

Error Handling: The process of identifying, reporting, and managing errors or exceptions that may occur during the execution of a program. Error handling is primarily concerned with handling unexpected events, exceptions, or errors that arise as a result of incorrect or unexpected input, software bugs, or other exceptional conditions.
Fault Handling: The process of detecting, isolating, and responding to faults or failures that occur within a system or application. Fault handling deals with managing faults or failures that can occur due to a variety of reasons, including hardware failures, software bugs, environmental factors, or unexpected system behaviors.
GUI: Graphical User Interface. The application in which the user interacts with graphical components.
System Snapshot: A capture of the system's current state at a specific point in time. It represents an image of the system, including the state of memory, CPU core registers, and other relevant system components.
TCL: Tool Command Language. It is a scripting language that is commonly used for various purposes, including automation, scripting, and embedded systems.
ISR: Interrupt Service Routine. It is a function in embedded systems and operating systems that is invoked when a specific interrupt condition occurs. Interrupts are signals sent by hardware devices or software processes to the CPU, indicating that an event has occurred that needs immediate attention. When an interrupt is triggered, the CPU interrupts its current execution, saves its state, and transfers control to the appropriate ISR to handle the interrupt event.
System On a Chip (SOC): An integrated circuit that incorporates different components onto one chip, as an integrated system.
System-On-Module (SOM): An integrated circuit board that is modular by design and can accept many different components to suit a need, including SOCs [1]

VI. References

[1] ““System-on-Modules (SOMs): How and Why to Use Them”, AMD XILINX, 2023, available at https://www.xilinx.com/products/som/what-is-a-som.html, accessed (accessed Sep 28, 2023)