Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

DEPARTMENT OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

<Specify Semester>

Enhancing CROC SoC Reliability with a Hardware Watchdog

<Specify Report Type>

Titlepage
Logo
Placeholder

<Specify Author>
<Specify E-Mail>

<Specify Date>

Professor: **<Specify Professor>**

Abstract

This report summarizes the work done in the VLSI 2 course project. The project consisted of extending the Croc SoC with a Watchdog Timer.

Contents

Chapter 1

Introduction

1.1 Overview of the Croc SoC

CROC SoC is a lightweight, open-source system-on-chip designed for education and rapid prototyping. It provides a complete RTL-to-GDSII flow and basic processor and peripheral infrastructure, and even includes an explicit user_domain where students can add and test their own modifications. As a relatively new project, CROC isn't yet fully optimized, leaving ample room to enhance its reliability, performance, and feature set.

1.2 Motivation for Watchdog Integration

TODO: ADD SOURCE NASA
In May 1994, the NASA Clementine lunar probe suffered a critical onboard computer failure shortly after departing lunar orbit. A malfunction caused one of its thrusters to fire uncontrollably, spinning the spacecraft at 80 rpm until it exhausted its remaining fuel. NASA was able to bring Clementine back online with a manual reset command, but only after the mission's continuation toward its asteroid target had become impossible. As it later emerged, just a few lines of watchdog-timer code—hard-wired into the flight computer—would have detected the hang, issued an automatic reset immediately, and prevented the thruster runaway that doomed the asteroid rendezvous.
Likewise, in the CROC SoC environment—where students routinely extend the design with custom peripherals and firmware—undetected hangs or deadlocks can leave the chip in an unusable state. Embedding a hardware watchdog ensures that any such fault triggers an automatic reset to a known-good state, dramatically improving system robustness and streamlining recovery during both development and tape-out testing.

1.3 Project Objectives

This project aims to design and integrate a simple yet effective hardware watchdog timer into the CROC SoC, with the following objectives:

- Reliable Fault Detection: Detect processor stalls or bus deadlocks within a bounded timeout period.

- Automatic Recovery: Trigger a chip-wide reset when the watchdog expires, returning the system to a known-good state.
- Minimal Overhead: Add the watchdog with low area, timing, and power impact, preserving CROC's lightweight footprint.

With these goals, the watchdog integration will bolster CROC's robustness without compromising its simplicity or educational value.

Chapter 2

Related Works

2.1 Overview of existing watchdog timer implementation

- State several possible watchdog implementations, with sources.
- Explain that it is possible to keep it really simple or to complicate it by adding some features, like having a log to know why the watchdog triggered.

2.2 Justification for our approach

Now, explain our choice to keep it small, simple, energy efficient easy to understand and easy to "take over" if it is needed for something else. Explain that the trade off gains in robustness of croc vs complication of watch dog is the best by keeping the watchdog simple.

Chapter 3

Methodology

3.1 Design choices for WDT (e.g., countdown vs. count-up, reset duration)

TODO: INCLUDE IMAGE AND REF TO IMAGE IN USER DOM
To equip CROC with automatic fault recovery, we designed a hardware watchdog and an OBI-protocol wrapper in the user_domain. Our key design decisions were:

- Count-up timer: We adopted the classic count-up approach (versus down-counter) because it aligns with industry practice and synthesizes to minimal logic. On each clock cycle that the watchdog is enabled and not refreshed, a small counter increments. Reaching its terminal value triggers a timeout event.
- One-cycle reset pulse: Upon timeout, the watchdog emits a one-cycle reset pulse. One cycle is sufficient to assert the global reset input across all domains and restart the entire SoC from its defined reset vector, without introducing undue latency or complicating the reset tree.
- Hardware implementation: A hardware watchdog can detect and recover from processor hangs or bus deadlocks even when the CPU itself has stopped executing instructions. A purely software watchdog—implemented as a loop in firmware—cannot fire if the CPU is already stalled, so it cannot guarantee recovery in all fault scenarios.
- OBI-protocol wrapper: The wrapper translates OBI reads and writes into the internal control signals of the counter, keeping the watchdog core itself free of bus-interface complexity.
- Placement in user_domain: By locating the watchdog in the user_domain, we avoid touching the croc_domain's stable, taped-out infrastructure.

3.2 Testing strategy (testbenches, simulation tools)

3.3 Debugging and iteration process

By separating the work in easy, well separated steps it was easy to debbug.

Chapter 4

Hardware Architecture

This chapter describes the hardware architecture of the Watchdog Timer that we implemented in the Croc SoC. In the different sections we talk about different implementations with different features and complexities.

4.1 Simple Watchdog Timer

4.1.1 Hardware implementation

The first implementation of the Watchdog Timer is the simplest one we could imagine. As shown in Figure 4.1, the WDT is composed of very few components. The main component is the counter, which is incremented every clock cycle and implemented as a flip-flop. The counter is connected to a comparator that checks if the counter has reached a certain value. If the counter reaches the value, the comparator will trigger a reset signal that will reset the system. Otherwise, we keep incrementing the counter. As soon as we get a kick signal, the counter is reset to zero.

4.1.2 Tests and Results

Explain how we did the tests (python files to generate stimuli + expected then compare), show figure. Explain that the results were always good at the end.

4.1.3 Conclusion

Explain that thanks to our approach we found that the watchdog works fine, so if a future problem would come it would come from the wrapper.

4.2 Watchdog Wrapper

4.2.1 Hardware implementation

Add a graph of the watchdog wrapper, and maybe a figure of how it connects to the OBI protocol.

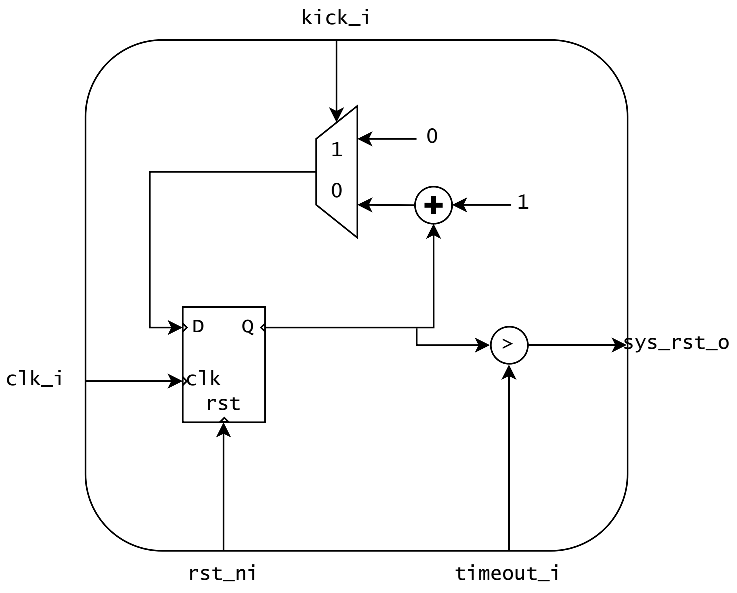

Figure 4.1: Simple Watchdog Timer Architecture

4.2.2 Tests and Results

We checked the functionality of our wrapper by running a helloword.c wich sets the timeout value and sends periodic kicks befor getting stuck in an impossible task. Our watchdog was successfully set thanks to the correctness of the wrapper and it allowed us to reset the program. We also checked the VCD to see if we could find any anormalities, but our wrapper respects OBI protocol :) .

Chapter 5

Evaluation

5.1 Simulation results

Our watchdog passed all our simulations: from the chip ggetting stuck in an infinit loop to TODO FIND OTHERS SIMULATION TESTS

5.2 Performance comparison (goal achieved?)

Our goal was awchieved: we cna successfully say that we managed to improve the reliability and robustness of croc by adding a small area and power. We also managed to store and print our names in a ROM stored in useer domain.

5.3 Possible improvements

A lot of improvements can be done, but all depend of the possible application and of the willingness to trade off simlicity, area, power for something else like debugging ability. TODO: explain more

Chapter 6

Conclusion

6.1 Summary of key findings

6.2 Future work and next steps