Diversity-Driven Hardware Task:
a Microcode Approach

Hugo Daniel Vieira de Carvalho

UMinho | 2023

Universidade do Minho
Escola de Engenharia

Hugo Daniel Vieira de Carvalho

Diversity-Driven Hardware Task: a Microcode Approach

janeiro de 2023

Universidade do Minho
Escola de Engenharia

Hugo Daniel Vieira de Carvalho

Diversity-Driven Hardware Task: a Microcode Approach

Dissertação de Mestrado
Mestrado em Engenharia Eletrónica Industrial e Computadores
Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação de
Professor Doutor Adriano José Conceição Tavares
Professor Doutor Vitor Alberto Teixeira da Silva

janeiro de 2023

Agradecimentos

Primeiramente, gostaria de expressar os meus agradecimentos ao professor, doutor Adriano Tavares, pelo conhecimento transmitido ao longo do mestrado, e pela disponibilidade e interesse demonstrados durante o desenvolvimento da dissertação. O domínio das várias áreas do conhecimento incutido foi sem dúvida importante para concretizar as metas impostas durante este ano de trabalho. Ao professor, doutor Vitor Silva, um sincero obrigado, por todo o conhecimento passado, apoio, e críticas construtivas. A sua experiência foi essencial para o desenvolvimento da dissertação e para o meu crescimento enquanto engenheiro em várias vertentes.

Ao meu colega e amigo José Mendes, agradeço pelo companheirismo e apoio durante estes anos e também durante os desafios e frustrações da dissertação. Não podia deixar de agradecer também aos meus colegas e amigos com quem durante estes anos partilhei o laboratório e experiências, e um especial obrigado aos meus amigos, Nuno Rodrigues e Francisco Marques pelas conversas e motivação para perseguir os meus objetivos.

A toda a minha família, quero agradecer profundamente pela paciência e pelo suporte que me deram e que tornou isto possível. Por todo o apoio, um obrigado do fundo do coração para o meu pai, para a minha mãe, para a minha irmã, Joana, avós, tios e tias. Que esta seja apenas a linha de partida para um futuro preenchido de sucesso e aprendizagem.

Um obrigado final a todas as pessoas que direta ou indiretamente me influenciaram a escolher um caminho onde me sinto concretizado.

Hugo de Carvalho, Guimarães, 2022.

STATEMENT OF INTEGRITY

Resumo

Tarefa de Hardware Orientada para a Diversidade: uma Abordagem em Microcódigo

Atualmente, a tecnologia escala a um ritmo muito elevado, proporcionando um crescimento de várias ordens de grandeza no desempenho de sistemas *VLSI* em áreas cada vez mais pequenas. Contudo, o aumento de desempenho eventualmente atingiu o limite, e *multicores* heterogéneos foram introduzidos, dificultando o *co-design* por falta de mecanismos de adaptação e semelhanças entre, por exemplo, *CPUs* e *FPGAs*. A direção principal da investigação começou então a mudar para uma que fundamentalmente permitia: (i) sistemas fiáveis e seguros, com preocupações relacionadas com adaptação baseada em reconfiguração, (ii) sistemas de baixa latência e determinísticos, baseados na aceleração em *hardware*, e (iii) sistemas eficientes energeticamente, focados na centralidade de dados. Com isto em mente, começaram a surgir modelos de *co-design* que visam abranger estas considerações como, por exemplo, o modelo de acelerador *HAL-ASOS*. Apesar de todos os modelos de programação híbridos para *CPU-FPGA* disponíveis na literatura, tanto quanto se sabe, exclusivamente o *HAL-ASOS* aplica microcódigo para alcançar um sistema com elasticidade *by design* e regularidade. Para expandir a investigação de *HAL-ASOS* a cerca de microcódigo e explorar as suas limitações, a dissertação propõe uma nova e específica arquitetura de acelerador baseada em dois esquemas de microinstrução, para fornecer abstrações ao nível do *kernel* e do utilizador. A arquitetura descreve um *microkernel* baseado nas premissas *HAL-ASOS*, com serviços de gestão de eventos, gestão de memória e memória temporária, gestão de recursos (tanto no kernel como na tarefa), bem como mecanismos de sincronização, e uma tarefa de hardware capaz de realizar *syscalls* estendidas, comunicando com o sistema operativo do *host* através de memória partilhada. Para fazer face à adaptabilidade do acelerador, a arquitetura foi migrada para permitir atualizações de microcódigo, e testada com um esquema de injeção de falhas. As restantes funcionalidades do modelo do acelerador foram verificadas sob um ambiente de simulação utilizando ferramentas Xilinx, e também sob um ambiente de simulação completa com ferramentas *HAL-ASOS*. A implementação do modelo de acelerador num *SoC* Zynq-7010 *ARM/FPGA* foi avaliada. Os resultados mostraram que, no que respeita à escalabilidade, os procedimentos *kernel-bounded* na tarefa de hardware, conseguem sempre reduzir um tipo de célula lógica combinacional da plataforma ZYBO Z7-10 (até 73%), considerando um ligeiro aumento na utilização de *LUT* (até 32%), ambas dependentes da versão. Além disso, os resultados também demonstraram que, por consequência, a execução *kernel-bounded* conseguiu reduzir a dispersão de células lógicas até 29%, e o número de *slices* incompletas da *FPGA* .

Palavras-chave: aceleração em *hardware*, elasticidade evolutiva, microcódigo, tarefa de *hardware*

Abstract

Diversity-Driven Hardware Task: a Microcode Approach

Nowadays, technology is scaling at a very high rate, coming along with a growth of several orders of magnitude in VLSI performance within increasingly smaller areas. However, performance increase eventually hit a pitfall, and heterogeneous multi-cores were introduced, making the co-design harder due to the lack of adapting mechanisms and commonalities regarding, for example, CPUs and FPGAs. The key direction of research then started to shift into a direction that fundamentally allowed: (i) reliable and secure systems, with concerns of adaptation through reconfiguration, (ii) low-latency and deterministic systems, relying on hardware acceleration, and (iii) energy-efficient systems, focused on being data-centric. With this in mind, co-design models that encompassed these considerations started appearing, e.g., the HAL-ASOS accelerator model. Despite all the hybrid programming models for CPU-FPGA available in the literature, to the best of one's knowledge, only HAL-ASOS employs microcode to achieve system elasticity by design and regularity. To expand upon HAL-ASOS research on microcode and explore its limitations, the dissertation proposes a new and specific accelerator architecture based on two microinstruction schemes, to provide kernel- and user-level abstractions. The architecture describes a microkernel based on HAL-ASOS' premises, with services for event management, memory and buffer management, resource management (in both kernel and task), as well as synchronization mechanisms, and a hardware task capable of performing extended syscalls, communicating with the host's operating system through shared memory. To tackle the adaptability of the accelerator, the architecture was migrated to allow for microcode updates, and tested with a fault injection scheme. All the remaining functionalities of the accelerator model were verified under a simulation environment using Xilinx's tools, and also under a full simulation environment with HAL-ASOS tools. The deployment of the accelerator model on a Zynq-7010 ARM/FPGA SoC was evaluated. The results showed that, regarding scalability, kernel-bounded procedures in the hardware task, could always reduce one type of combinational logic cell of the ZYBO Z7-10 platform (up to 73%), considering a slight increase in LUT usage (up to 32%), both version-dependent. Moreover, the results also exhibited that, as a by product, kernel-bounded execution was able to reduce the dispersion of logic cells up to 29%, and the number of incomplete FPGA slices.

Keywords: evolutive elasticity, hardware acceleration, hardware task, microcode

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Acronyms

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | Z

A

AES	Advanced Encryption Standard.
ALU	Arithmetic Logic Unit.
AMBA	Advanced Microcontroller Bus Architecture.
AMD	Advanced Micro Devices, Inc..
API	Application Programming Interface.
ASIC	Application Specific Integrated Circuits.
AXI	Advanced eXtensible Interface.

B

BFM	Bus Functional Model.
BOF	BORPH Object File.
BORPH	Berkeley Operating System for Reprogrammable Hardware.
BRAM	Block Random-Access Memory.

C

CHE	Capability Hardware-Enhanced.
CISC	Complex Instruction Set Computer.
CMF	Common-Mode Failures.
COTS	Commercial Off-The-Shelf.
CP	Check-Pointing.
CPU	Central Processing Unit.

D

DCLS	Dual-Core Lockstep.
DMA	Direct Memory Access.
DMR	Dual Modular Redundancy.

DPR Dynamic Partial Reconfiguration.

DSE Design Space Exploration.

E

eCos Embedded Configurable Operating System.

EVM Event Manager.

F

FAST Features from Accelerated Segment Test.

FFT Fast Fourier Transform.

FIFO First In First Out.

FISH FPGA-Initiated Software-Handled.

FOS FPGA Operating System.

FPGA Field-Programmable Gate Array.

FSL Fast Simplex Link.

FSM Finite State Machine.

FUSE Front-End User Framework.

G

GCC GNU Compiler Collection.

GPU Graphics Processing Unit.

H

HAL Hardware Abstraction Layer.

HAL-ASOS Hardware Assisted Linux for Application Specific Operating Systems.

HDL Hardware Description Language.

HIF Hardware Intermediate Form.

HLS High Level Synthesis.

HTI Hardware Thread Interface.

HTT HybridThreads Transpiler.

HWR Hardware Region.

HWSC Hardware System Calls.

I

I/O Input/Output.

IBM International Business Machines Corporation.

IC Integrated Circuit.

IDU Instruction Decode Unit.

ILP Instruction Level Parallelism.

IP Intellectual Property.

IPC Inter Process Communication.

IR Intermediate Representation.

IRQ Interrupt Request.

ISA Instruction Set Architecture.

J

JIT Just-in-Time.

K

KFIFO Kernel FIFO.

KIVIO Kernel Interface for Virtualized I/O.

L

LBUS Local Bus.

LC Loosely-Coupled.

LINTC Local Interrupt Controller.

LKM Loadable Kernel Module.

LMUTEX Local Mutex.

LRAM Local Random-Access Memory.

LST Lockstep Shadow Threads.

LUT Look-Up Table.

M

MBUS Master Bus.

MEMIF Memory Interface.

MMU Memory Management Unit.

MSB Most Significant Bit.

N

NRE Non-Recurring Engineering.

NSF Next Step When False.

O

OS Operating System.

OSFSM Operating System Finite State Machine.

OSIF Operating System Interface.

P

PL Programmable Logic.

PLB Processor Local Bus.

PoC Proof of Concept.

POSIX Portable Operating System Interface.

PR Partial Reconfiguration.

PS Processing System.

Q

QEMU Quick Emulator.

R

R3TOS Reliable Reconfigurable Real-Time Operating System.

RACOS Reconfigurable Accelerator Operating System.

RAM Random-Access Memory.

RB Rollback.

RE2DA Reliable and Reconfigurable Dynamic Architecture.

RES Resource-Elastic Scheduling.

RIFFA Reusable Integration Framework for FPGA Accelerators.

RISC Reduced Instruction Set Computer.

ROM Read-Only Memory.

ROP Return-Oriented Programming.

RTL Register-Transfer Level.

RTOS Real-Time Operating System.

S

SEOS Simple and Effective Real-Time Operating System.

SEU Single Event Upset.

SGX Software Guard Extensions.

SLD System-Level Datapath.

SoC System-on-Chip.

SPREAD Streaming-Based Partially Reconfigurable Architecture and Programming Model.

SRAM Static Random Access Memory.

ST Shadow Thread.

SYSMUTEX System Mutex.

SYSRAM System Random-Access Memory.

T

TC Tightly-Coupled.

TCLS Triple Core Lockstep.

TDP True Dual-Port.

TMR Triple Modular Redundancy.

TRI Task Resource Isolation.

TS Thread Shadowing.

TUO Thread-Under-Observation.

U

UML Unified Modelling Language.

V

VHDL Very High-Speed Integrated Circuit Hardware Description Language.

VIP Verification IP.

VLSI Very-Large-Scale Integration.

Z

ZCU ZeroCopy Unit.

1. Introduction

In the era of smart-everything it is easy to forget or disregard the technology improvement of the years past, but the fact is that the last few decades allowed for significant performance increases of about five orders of magnitude in Very-Large-Scale Integration (VLSI) [8], e.g., Cerebras' Wafer Scale Engine accommodates one point two trillion transistors within around forty-six cubic millimeters, and Nvidia's Titan V GPU accommodates around twenty-one billion transistors within eight hundred and fifteen cubic millimeters [44]. This happened whilst transistor sizes kept decreasing up to a certain period. This traces back to the slowing of *Moore's Law*, with the number of transistors per chip not doubling every year, and the termination of *Dennard's Scaling*, with the current and voltage of a certain silicon area not being able to drop below threshold levels whilst proving functionality, *c.f* [48]. With this, the increase of clock rates no longer provided steady performance increases, and the migration from single-core to multicore CPUs occurred [10].

Subsequently, due to the multiprocessing limitations imposed by *Amdahl's Law*, which states that performance can not continue to grow indefinitely by adding additional cores to the system, the only route left for performance increase was the one that considered specialized heterogenous multicore architectures, e.g., ARM's big.LITTLE [62] or Intel's Alder Lake [22]. The first shift in notion was the aforementioned heterogeneous multi-cores, but after the paradigm changed with the introduction of heterogeneous architectures based on different processing elements [68], e.g., CPU+FPGA or CPU+GPU. Alongside, these FPGA-based architectures started to dethrone CPU-only architectures. Reliable systems want to use devices like FPGAs to increase the flexibility and computation power in critical scenarios where failures are not acceptable, e.g., self-driving automotive industry, clusters, cryptography, among others. Although, FPGAs are also developing at a high rate, becoming more complex and dense devices [38]. This factor is a concern because the programming of CPU+FPGA systems gets harder, due to the lack of adaptation mechanisms and consistent and designer-friendly programming models for these hybrid systems.

Considering this, research started to shift into a direction that allowed: (i) fundamentally reliable and secure systems that resolved the issues regarding new points of failure with novel adapting strategies, (ii) fundamentally low-latency and predictable systems that surpassed traditional hardware acceleration, and provided tools for a seamless design of systems based on CPU+FPGA, whilst accompanying the evolution of FPGA devices, and (iii) fundamentally energy-efficient systems based on data-centricity to achieve long-lasting operation. Hybrid programming models based on RTOS [4, 3, 49, 33, 34, 32, 46, 24] and Linux [32, 1, 9, 66, 72, 23, 45, 25, 69, 70, 63] then started to appear to yield these considerations and allow for "world crossing", i.e., optimization across the software-hardware hierarchy. One of those models is the HAL-ASOS accelerator model [63, 62].

HAL-ASOS distinguishes itself from the literature mainly because of its microcode mechanisms. Originally, microcode acted as an abstraction layer between the CPU's hardware and the ISA, referred in some instances plainly as *firmware* [67]. HAL-ASOS [62] transposed these concepts into the programmable hardware, and established a microcode-based and CPU-like architecture with the capability for providing kernel- and user-level abstractions like a typical OS. Microprogrammed control units are different from FSM-based control units in various aspects. Firstly, microprogrammed control units have a memory where they store the microinstructions, and that, as a whole, forms the microcode. Additionally, associated with this memory they typically have logic for the decoding of microinstructions into control signals, and sequencing logic that determines the next microcode memory address. The sequencer usually has signals as inputs, and bases the decision of the next address on a logic expression of those signals. Altogether, these components form a controlling entity which is responsible for sequencing and executing microinstructions, dictating the execution flow. In terms of taxonomy the instructions present in the microcode memory can be classified as horizontal or vertical depending on the encoding utilized and the control granularity. Minimally encoded microinstructions fall into the horizontal spectrum, which means that a certain microinstruction subfield might activate a control signal directly, while microinstructions that are more encoded fall onto the vertical spectrum, specifying, for example, a function code to perform a series of datapath operations.

The fundamental research on microcode and the establishment of the HAL-ASOS architecture and its associated tools, e.g., full simulation, facilitate posterior research that aims to explore and improve upon its limitations in an attempt to leverage diversity.

1.1 Motivation

FPGAs are becoming the go-to devices for implementing the majority of critical embedded systems in many applications areas, e.g., aerospace [6, 7, 57, 52], supercomputing and clusters [71], edge computing [75], cryptography [2, 63, 72], high-throughput video processing [55, 56], and power electronics [61]. Moreover, there is no sign that this popularity growth will stop any time soon, so committing to research in this area seems like a good investment for the future. The fact that FPGAs are connected to so many emerging subjects and the fact that the knowledge gained also applies to ASIC design (with some slight changes) is what also propelled the motivation to embrace this research challenge.

On another note, the implementation of hybrid systems presupposes the migration of certain parts of the system to programmable logic, and thus, there are also new threats to the system caused by the static and error-prone nature of the hardware. The application of microcode mechanisms for reconfigurability in hybrid OSes is still in its early stages in terms of the literature. After an in-depth analysis of the employment of these techniques in the context of hybrid OSes, one can identify that only a few focused on reconfiguration premises and that only HAL-ASOS used microcode to tackle the problem. Concerning these factors, one identifies that this is in fact a topic worth exploring, opening up the possibility to research hybrid architectures in what relates to reconfiguration, adaptability, security, and scalability.

1.2 Research Goals

The dissertation work aims to improve upon some HAL-ASOS limitations and openly stated issues. To do so, a sandbox-like kernel-task environment needs to be developed in hardware in order to replicate some of the issues, and only then one can start to address them. Consider that in the sense of research and development, this is meant to be done with in an exploratory manner, so the imposition of initial restrictions to the project would be a lackluster. Even though, after a broad literature review, *c.f* chapter 2, it is still possible to define the main goals as following:

1. **Ad hoc microkernel to support the adaptable architecture:** it is necessary to build a fundamental hardware kernel and hardware task somewhat similar to HAL-ASOS to understand the architecture and the identified limitations;

2. **Diversity-driven hardware task architecture extensions**: upon being able to replicate some of the issues, start improving the existing architecture from the ground up by adding functionality

and/or resolving the issues;

1.3 Dissertation Structure

The dissertation is structured according to the layout of fig. 1.3.1. This section presents the structure of the dissertation. For now, seven chapters constitute the dissertation: chapter 1 (the present one) introduces the dissertation's thematic, the motivation to embrace the research project, and defines its envisioned goals. Chapter 2 approaches three different sections based around the dissertation's title that introduce some background concepts, contextualizes them and provides an in-depth analysis of related works. Chapter 2 starts with some reconfigurable computing considerations, followed by hybrid programming models/operating systems, and finishes with some fundamental aspects about microprogramming/microcode. Chapter 3 focuses on describing the groundwork and parallel work of the dissertation by presenting HAL-ASOS and KIVIO, respectively. At the end of the HAL-ASOS contextualization one identifies its limitations, mainly based on openly-stated issues. Some of these limitations have the intention of being replicated and ultimately approached in chapter 5. Moreover, chapter 4 describes the first goal of the dissertation, which is making a functional microkernel based on HAL-ASOS to know the system and its limitations from the roots. Concerning this, the chapter approaches the microkernel's architecture, microcode, system calls implemented, available services, the hardware task development, resource management considerations, available resources regarding interrupt management, synchronization mechanisms, memory management, interface timing, and functional registers. After, the external interfaces are approached, ending with the microkernel's register mapping onto those interfaces. Chapter 5 addresses some of the limitations of HAL-ASOS introduced in chapter 3 by proposing solutions and also presents solutions for adding additional features regarding adaptation. Chapter 6 talks about the tests performed over the dissertation's work to prove functionality and identify the weak and strong points of the developed architecture. Finally, chapter 7 gives a brief summary on the work's contextualization, describes what was implemented and achieved, and ends with the future works suggestions.

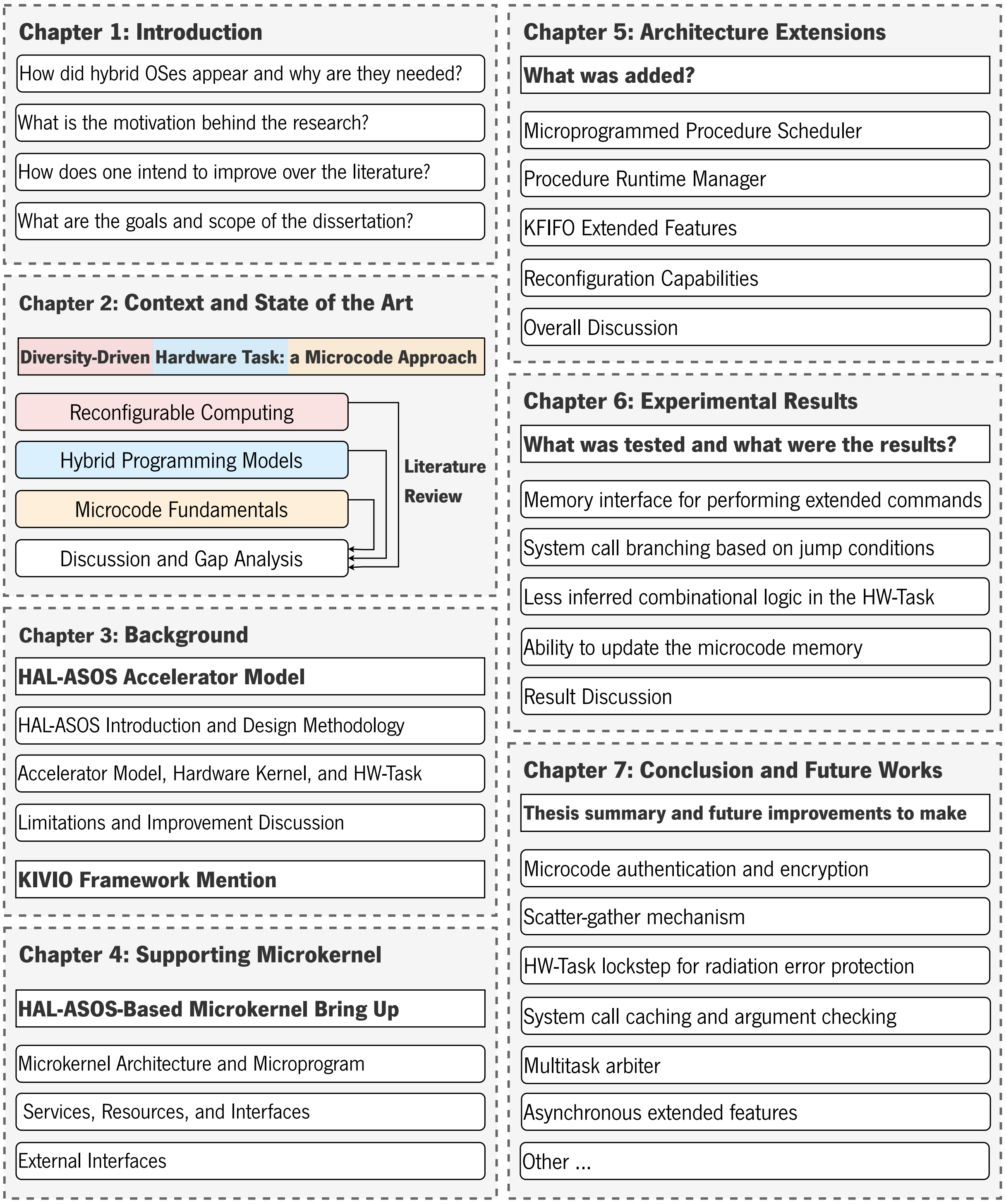
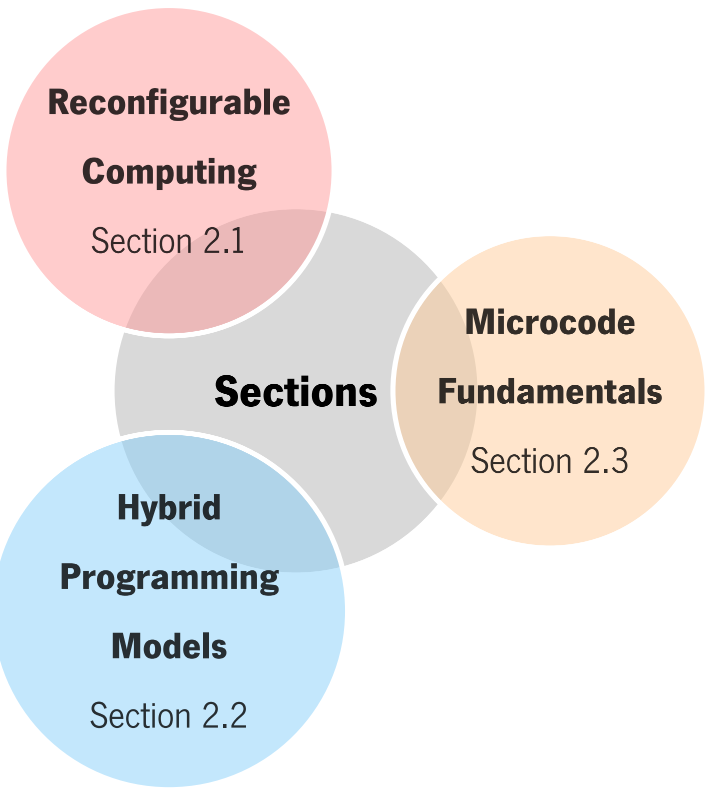

Chapter 1: Introduction
How did hybrid OSes appear and why are they needed?
What is the motivation behind the research?
How does one intend to improve over the literature?
What are the goals and scope of the dissertation?
Chapter 2: Context and State of the Art
Diversity-Driven Hardware Task: a Microcode Approach
Reconfigurable Computing
Hybrid Programming Models
Microcode Fundamentals
Discussion and Gap Analysis
Literature Review
Chapter 3: Background
HAL-ASOS Accelerator Model
HAL-ASOS Introduction and Design Methodology
Accelerator Model, Hardware Kernel, and HW-Task
Limitations and Improvement Discussion
KIVIO Framework Mention
Chapter 4: Supporting Microkernel
HAL-ASOS-Based Microkernel Bring Up
Microkernel Architecture and Microprogram
Services, Resources, and Interfaces
External Interfaces
Chapter 5: Architecture Extensions
What was added?
Microprogrammed Procedure Scheduler
Procedure Runtime Manager
KFIFO Extended Features
Reconfiguration Capabilities
Overall Discussion
Chapter 6: Experimental Results
What was tested and what were the results?
Memory interface for performing extended commands
System call branching based on jump conditions
Less inferred combinational logic in the HW-Task
Ability to update the microcode memory
Result Discussion
Chapter 7: Conclusion and Future Works
Thesis summary and future improvements to make
Microcode authentication and encryption
Scatter-gather mechanism
HW-Task lockstep for radiation error protection
System call caching and argument checking
Multitask arbiter
Asynchronous extended features
Other ...

Figure 1.3.1: Thesis Structure Diagram.

2. Context and State of the Art

This chapter will be structured around the dissertation title, i.e., "Diversity-Driven Hardware Task: a Microcode Approach". Concerning this, the first section will approach reconfigurable computing (🟥) leading up to the discussion of adaptability in hybrid (hardware-implemented) operating systems to analyze the literature for the means of coping with diversity, i.e., develop diversity-driven architectures. The second section will address hybrid programming models (🟦), as a way to explore the literature for the hardware thread/task concept. Finally, in the third section, it will be provided some background on microcode mechanisms (🟧) with an analysis of related works. The chapter ends with an overall and comprehensive analysis of the state-of-the-art gaps as a whole, in the discussion section 2.4.



2.1 Reconfigurable Computing

A diversity-driven system, i.e., a system driven by diversity or that evolves by adapting, needs, first and foremost, of a mechanism that allows it to be reconfigurable, to be able to cope with change, e.g., for the correction of errors or making updates. In this section one will, firstly, present FPGA technology, analyze its evolution and reconfiguration capabilities, and, secondly, explore adaptability-related concepts in the literature pertaining to hybrid programming models (*c.f* section 2.2) to form a research basis.

Contents

2.1.1 FPGA Technology

As the name implies, a Field-Programmable Gate Array (FPGA) is a semiconductor IC, available as COTS, in which the hardware is capable of being reprogrammed after manufacturing. This characteristic solely distinguishes a FPGA system from an ASIC one, since the latter is custom manufactured for specific applications that do not need much alteration after manufacturing. Accordingly, the FPGA technology leads to faster design cycles, faster time-to-market, and cheaper implementations associated with the fact that there are no upfront NRE costs [7].

2.1.1.1 Evolution

The flexibility and computation power that the FPGA technology yields allows for a plethora of critical applications that require reliant and reconfigurable systems as their backbone, including systems that integrate FPGA architectures in existing operating systems using unified programming models. More on that later in section 2.2. The first FPGA was launched by Xilinx in 1985. At the time FPGAs were mainly used for implementing glue-logic, simple Finite State Machine (FSM), and data processing units, considering that the number of gates in original devices was too small to compete with ASIC devices at the time [7]. Nevertheless, nowadays, the technology improved so much that these systems have taken over the ASIC market and are the core of applications in areas like [21]: aerospace, supercomputing and clusters, edge computing, cryptography, high-throughput video processing, power electronics, and more, table 2.1.

Table 2.1: FPGA Application Areas and Research Cases.

Field	**Research**
Aerospace	[6, 7, 57, 52]
Supercomputing and Clusters	[71]
Edge Computing	[75]
Cryptography	[2, 63, 72, 62]
High-Throughput Video Processing	[55, 56]
Power Electronics	[61]

Note: The cases provided serve only as examples.

2.1.1.2 Granularity

FPGAs are comprised of programmable logic blocks embedded in programmable interconnect, as observed in fig. 2.1.1. The architectural granularity of FPGA devices can be distinguished as fine-, medium- or coarse-grained based on the number of connections established between logic blocks and the functionality they provide. For that reason, architectures that call for a higher amount of interconnections between blocks and relatively present much less functionality are classified as fine-grained. In contrast, as the architectural granularity of FPGA devices increases beyond medium-grained the number of connections decreases but the supported functionality increases, reaching the point in some coarse-grained architectures where a single logic block can be a complex element as a Fast Fourier Transform (FFT) [38].

2.1.1.3 Reconfiguration

Reconfigurable FPGAs have a configuration memory that specifies the function of the device, for the purpose of reprogramming [26]. This type of FPGAs typically have a SRAM-based memory, and can be partially or dynamically reconfigured. The case in which the device can only be reconfigured once at boot

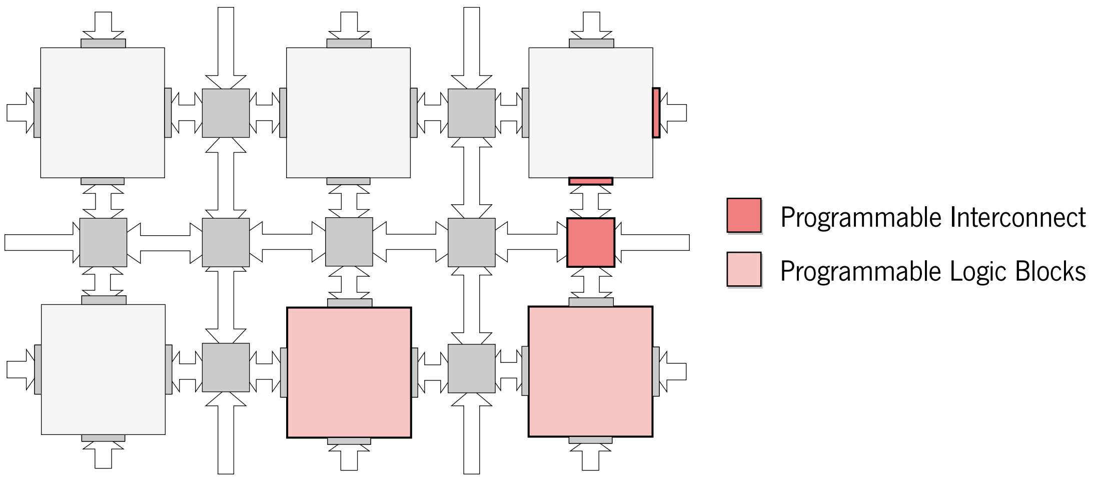


Figure 2.1.1: Underlying Field-Programmable Gate Array Fabric [38].

time is named as partial reconfiguration, and the case that allows the change of the memory contents at runtime is named dynamic reconfiguration, fig. 2.1.2. All dynamic reconfigured FPGAs can be partially reconfigured, hence the term dynamic partial reconfiguration, for the systems that use both methods.

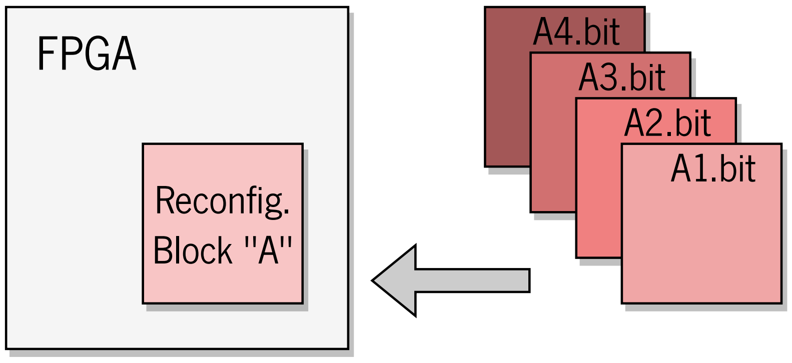


Figure 2.1.2: Basic Premise of Dynamic/Partial Reconfiguration.

2.1.2 Adaptability in Hybrid Operating Systems

In the scope of this thesis, the term “hybrid operating systems” is used to describe a unified programming model that benefits heterogeneous CPU+FPGA architectures. This topic will be discussed in more detail in section 2.2. Regarding adaptability, this type of systems can use built-in mechanisms to deal with problems or changes to the hardware architecture [63, 62] or rely on PR or DPR mechanisms, explained in section 2.1, in the last paragraph, to achieve the same goal, i.e., change hardware components implemented in the FPGA.

HAL-ASOS [63, 62] makes use of a built-in mechanism that allows reconfiguration of a microcoded unit through a AXI4 lite interface to promote system call changes without the need of PR or DPR. In this particular case, adaptability is treated as inbred and considered since the phase of design. This aspect removes the need for tacked-on approaches, i.e., approaches that rely on black-box modules to make changes to hardware by swapping the bitstream of the boxes, fig. 2.1.2, since the hardware can simply stop the hardware user space state machine, perform the desired changes, and then return to normal execution. For more details about the hardware implementation of the HAL-ASOS user space FSM, namely in the entity called "Hardware Task" (*c.f* section 3.1.4).

Fault tolerance [15, 5] is a concept that is connected to adaptability, as it has an error correction phase. Once again, the problems that originate from this error detection can be treated by built-in mechanisms or by other mechanisms that still rely on reconfiguration, section 2.1. Concerning error recovery, Avizienis *et al.* [5] identifies error handling techniques based on error elimination, e.g., rollback, rollforward, and compensation, and fault handling techniques based on fault activation prevention, e.g., diagnosis, isolation, reconfiguration, and reinitialization.

The checkpointing and rollback technique focuses on restoring the system state to a previously saved state with no errors, i.e., checkpoint. This method is used, for example, in the RE2DA architecture [51], designed for the automotive domain, to save the state of four processors. The system uses processor redundancy, i.e., processors behaving equally and comparing results, because of the nature of the target applications, i.e., the safety-critical field. This case implies a mechanism that handles change, i.e., appearing faults, so it is worth mentioning as an adaptable system.

R3TOS [24] is a case of a hybrid operating system that uses reconfiguration (DPR). It relies on an isolation technique (TRI) to exclude the faulty element from the system and prevent it from being used in future service delivery, followed by reconfiguration and diagnosis. This OS also uses rollforward to correct errors in its hardware microkernel, i.e., the system tries to correct and eliminate the present error, to achieve a new state with no errors.

Other works as SPREAD [72], FUSE [23], FOS [69], RACOS [70] can be included as hybrid OSes that use reconfiguration mechanisms.

2.1.2.1 Note on Lockstep

Lockstep is a known technique in which two entities run in parallel, whilst executing the same set of operations. Lockstep can use DMR to achieve fault detection, by comparison of the entities' outputs, or TMR to achieve fault detection and error correction, through majority voting. Concerning error recovery, lockstep can use rollback techniques [36]. In application to processors, the former is known as Dual-Core Lockstep (DCLS), and the latter as Triple Core Lockstep (TCLS) [36]. Due to the scope of the dissertation (chapter 2), one is more interested in lockstep applied at the hardware thread level, as it is the case of the work by Meisner and Platzner [41, 39, 40], and the work by Liu *et al.* [31]. This particular case of thread-level lockstep is known as Thread Shadowing (TS).

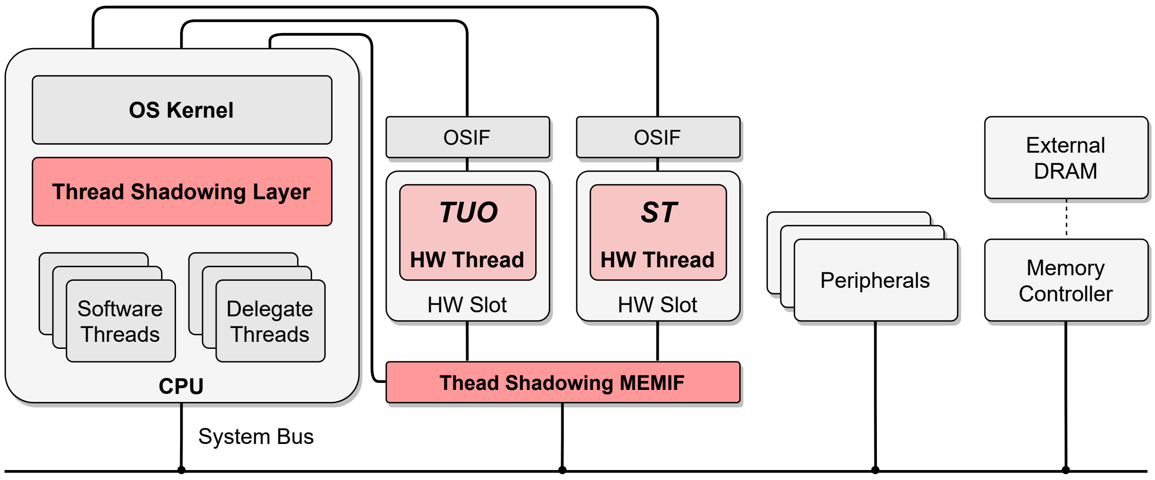


Figure 2.1.3: ReconOS architecture extended to support thread shadowing, adapted [40].

ReconOS [33, 34, 32, 1] is an example of an adaptable hybrid OS that uses DPR. This will be further mentioned in section 2.2.4.2. Additionally, the work in [40] applies TS to hardware threads within a ReconOS environment, represented as an example in fig. 2.1.3. The system makes use of DMR to attach a Shadow Thread (ST) to a Thread-Under-Observation (TUO), mirroring all the inputs and performing comparison on the outputs [40]. The work in [41] by the same authors, and referring to the same research, also specifies that these threads are not type restricted, i.e., besides being possible for a software thread to monitor a software thread, and a hardware thread to monitor a hardware thread, it is also possible for a hardware thread to monitor a software one and vice-versa (use of implementation diversity). Time diversity is implemented with a delay between the ST and TUO.

Furthermore, the research in [40, 41, 39] also introduced signature levels that specify the depth of error coverage in output comparison: (1) OS system call name checking, (2) OS system call name and parameter checking, and (3) which adds memory access comparison [39]. All of this makes ReconOS an adaptable OS aware of errors, like calling unregistered system calls, calling registered system calls with incorrect parameters, or performing memory accesses with wrong addresses.

2.2 Hybrid Programming Models

The establishment of trade-offs is an inherent aspect of technological progression for improving, for example, power consumption, performance, programmability, and portability [68]. To meet the ever-evolving requirements imposed by the end-users, one needs to take a step into novel approaches for programming models, exploring design trade-offs and the solution space. This section will go about hybrid programming models/operating systems for CPU+FPGA systems. The latter begins with some real-time considerations, followed by an analysis of traditional hardware acceleration strategies. After, one expands a bit more on the evolution of FPGA technology to justify how the hardware acceleration schemes improve within hybrid OSes and also explain the need for these hybrid models. Finally, some models are categorized into RTOS- and Linux-based, being dissected more deeply to identify advantages and disadvantages.

Contents

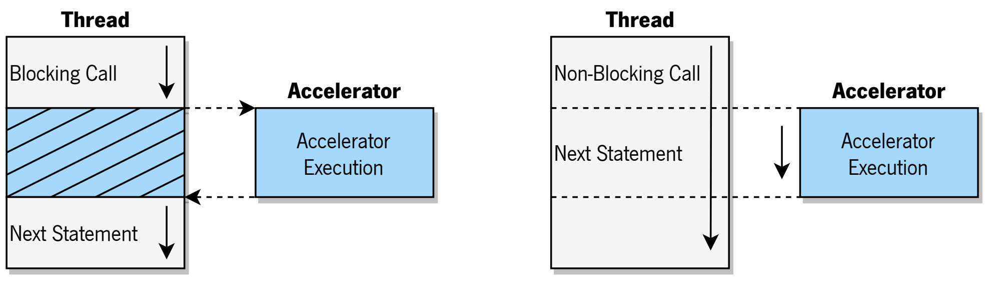
2.2.1 Real-time Concerns

A fundamental characteristic of real-time systems is the dependency upon time-constrained results. Therefore, the meeting of deadlines becomes a key factor to ensure system execution non-degradation or to safeguard functionality, depending on if the system is soft or hard real-time, respectively [47]. The accomplishment of deadlines presupposes the existence of deterministic and as-fast-as-necessary controllers that can mitigate latency and also make the system less prone to unpredictability. The lack of determinism is associated with dynamic events and is provoked by response time variation, also known as jitter. Among existing approaches that improve upon these metrics, there is one that distinguishes itself from others due to its hitherto ever-growing capabilities: hardware acceleration.

2.2.2 Traditional Hardware Acceleration

The most simple case of hardware offloading is the one where an accelerator is treated as a simple incarnation of a software function, while building the interface with the hardware based on the software function prototype [43]. Although, this is an undermining view on the potential of hardware accelerators since there is not a well-established strategy to provide services to the accelerator, as it can be seen in fig. 2.2.1. As stated by Enno Lübbers and Marco Platzner *et al.*, creators of the ReconOS [33, 34, 32, 1, 40, 41, 39]:

“ Traditional approaches integrate hardware accelerators as slave coprocessors [33]. Due to the lack of pervasive high-level programming abstractions, most reconfigurable hardware accelerators are implemented as passive coprocessor-like extensions invoked via procedure calls [32]. ”



(a) Blocking Accelerator. **(b)** Non-blocking Accelerator.

Figure 2.2.1: Overview of blocking and non-blocking hardware accelerator calls, without concurrency, in a passive coprocessor-like model, adapted [43].

Regarding the evolution of FPGAs into more complex and dense devices, as stated in section 2.1.1.1 and section 2.1.1.2, their role also shifts into being deployment targets for implementing, simultaneously, more complex control-dominated tasks and data-centric tasks that explore the true parallel nature of the hardware. However, programming models for designing such systems do not follow the same evolution, and, consequently, the need for new software-hardware boundary-crossing models arises. Data-centric tasks refer to tasks where the data-analysis resources are dynamically acquired or allocated, in response to demand. Recurrently, data-centric tasks benefit from data locality provided by any sort of caching [54]. For example, novel approaches to system call checking are using specific caches to store safe system calls and arguments to achieve faster analysis, and further protect the OS kernel from user attacks [64].

The aforementioned hybrid programming models can be classified into the following categories: RTOS-based models [4, 3, 49, 33, 34, 32, 46, 24] and Linux-based models [32, 1, 9, 66, 72, 23, 45, 25, 69, 70, 63], depending on the assisting Operating System (OS) on the processing system side. The next sections will detail a bit more on these models, starting from the RTOS-based models up to the Linux-based ones, with an analysis of the most notable research cases.

2.2.3 RTOS-Based Models

This section refers to the RTOS-based programming models, introducing Hthreads [4], ReconOS version one [34] and two [32], and some additional research cases.

2.2.3.1 Hthreads

Works like the ones by Andrews *et al.* and Peck *et al.* [4, 3, 49] broaden the outlook over hardware acceleration by introducing the new concept of hardware thread. In [4] is proposed a model that attempts to make programming across the software-hardware boundary seamless by migrating/mapping services to the hardware and abstracting them under the form of a Hardware Thread Interface (HTI). This approach makes the software- and hardware-based threads opaque and indistinguishable from the developer's point of view, opening up new options for designing compound systems. Figure 2.2.2 depicts the hybrid thread abstraction layer and system block diagram according to Andrews *et al.* [4]. The work present in [3] describes a multithreaded RTOS kernel for systems that benefit from the use of hardware and software threads. Based on a hardware/software co-design scheme to extend OS kernel services like the scheduler, interrupt processing, and semaphores to the hardware, Andrews *et al.* achieves reduced worst-case scheduling latency and interrupt latency, reduced scheduling jitter and asynchronous interrupt jitter, and also less CPU context switches, when comparing the work with a version implemented only on software.

Nonetheless, it is important to note that these hardware threads are generated using a sequential

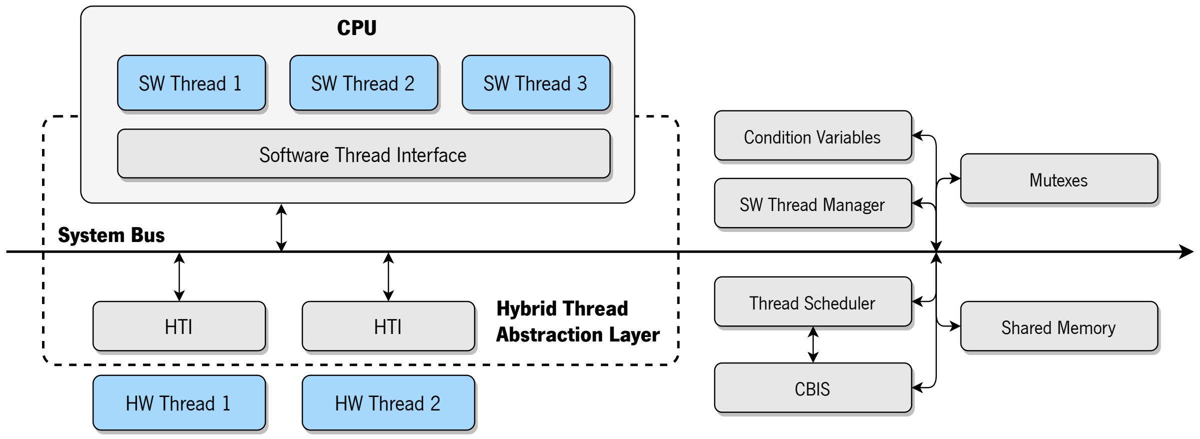


Figure 2.2.2: Hthreads: Hybrid Thread Abstraction Layer and System Block Diagram, adapted [4, 3].

language, i.e., the C programming language. The compilation flow for the translation of C code into a Hardware Description Language (HDL) like VHDL follows the steps described in fig. 2.2.3. The baseline source code (C) is compiled [16] into an Intermediate Representation (IR) (GIMPLE family), which is then optimized by GCC, passing to the Hardware Intermediate Form (HIF) characteristic of hthreads. Subsequently, after being translated by a HIF compiler, one obtains the VHDL representation, which is then integrated with the remaining hardware and included in synthesis. The Hthreads mechanism [49] can be reasonably compared to a HLS mechanism. As so, the disadvantages dwell on extracting truly parallel behavior from a sequential language and describing it, at the Register-Transfer Level (RTL).

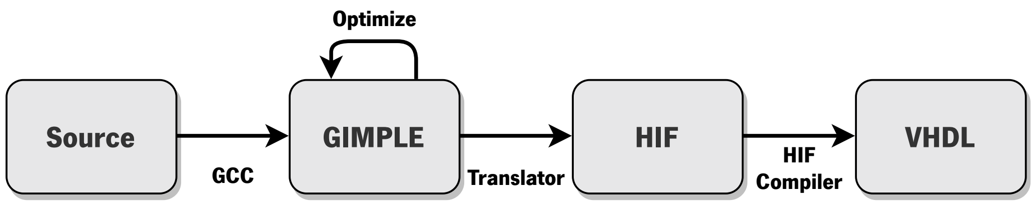


Figure 2.2.3: Hthreads: Compilation Flow using the HybridThreads Transpiler (HTT) [49].

2.2.3.2 ReconOS V1 and V2

The work in Lübbers *et al.* [34] introduces ReconOS, a RTOS kernel that supports hardware and software threads and that is built upon the eCos RTOS. The research baseline is the available and configurable RTOS (eCos) and the primal goal is to integrate hardware cores into the system, to achieve a hybrid operating system. With this purpose, three goals are identified: (1) hardware threads must have access to services offered by eCos; (2) The model should support true parallel execution: concurrent execution of one software thread and multiple hardware threads; (3) Support existing eCos API and existing hardware Intellectual Property (IP).

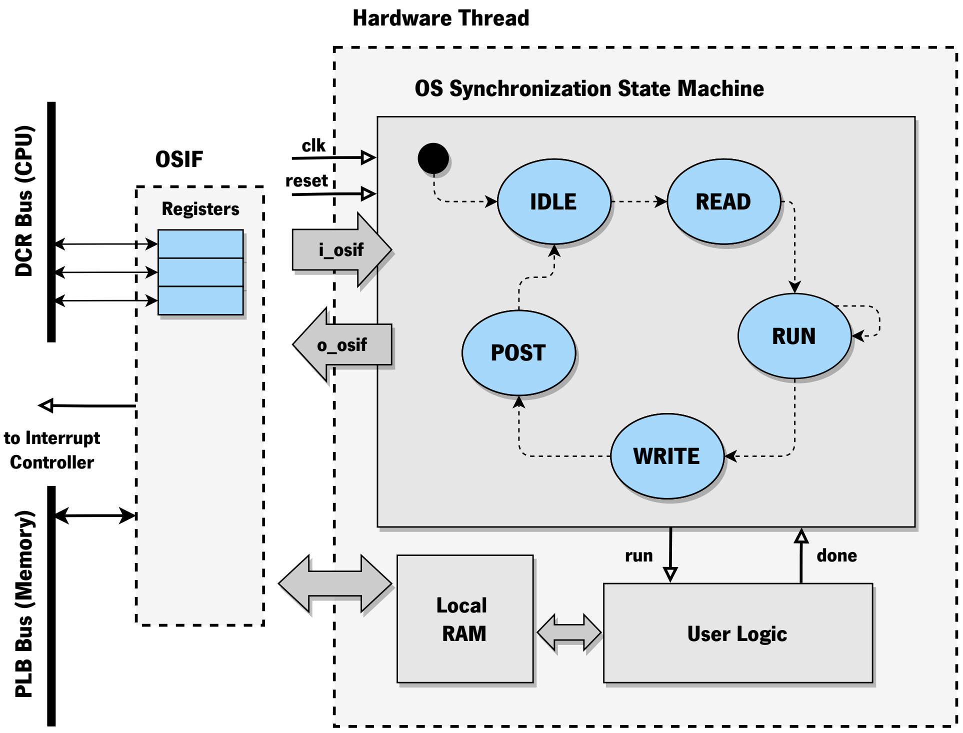


Figure 2.2.4: ReconOS: Communication between HW Thread and OSIF, adapted [34, 32].

In common software implementations, threads interact with the OS using several library APIs, existing a distinction between blocking and non-blocking calls. Additionally, threads execute sequentially. As opposed to, hardware threads are innately parallel, and as such the notion of calling an OS function or making a system call does not exist. For this, Lübbers *et al.* creates a library with VHDL procedures that wrap around the required system call, necessary for the synchronization with the operating system and other hardware threads. Accordingly, the hardware threads proposed have at least a synchronization state machine and the user logic. The synchronization state machine is dependent on OS signaling, and therefore it executes sequentially. In contrast, the hardware thread itself executes in a parallel manner. The hardware thread interface with the OS [33, 34, 32] can be seen in fig. 2.2.4.

The work by Lübbers *et al.* also specifies the concept of delegate thread to achieve transparency and allow for better Design Space Exploration (DSE). Each hardware thread created is associated with a software thread (delegate thread) that executes system calls on the behalf of that specific hardware thread. From the eCos point of view, the hardware thread is seen as a software thread with access to the Operating System Interface (OSIF). Each delegate thread has a table of objects used by the hardware thread, so it can map those resources into OS kernel resources. Whenever the OSIF generates an interrupt, the delegate thread is scheduled for execution, as seen in fig. 2.2.5. The delegate threads were introduced into the eCos ecosystem by extending its pre-existing thread class to include information relative to hardware threads. This information includes OSIF addresses, interrupt numbers, and OS object tables. By doing so, the delegate threads can benefit from existing eCos kernel services whilst providing the support the hardware threads demand. Additionally, the approach based on eCos disables the Memory Management Unit (MMU) to achieve simplified memory access by both the software and hardware threads to their virtual address space. This is a trade-off between memory protection/privilege management to attain higher performance and simpler memory transfers.

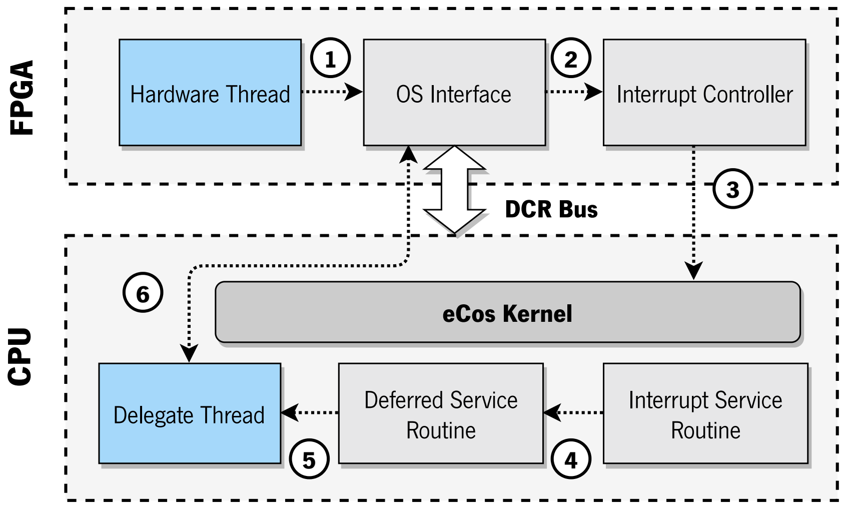


Figure 2.2.5: ReconOS: Communication between the HW and Delegate Thread in eCos [32].

However, the existence of a single interface between the host operating system and the hardware thread structure might be the cause of system bottlenecks, which is not favorable in a context where one wants to minimize the overall latency and jitter without jeopardizing performance. This is a problem resolved in future ReconOS versions, namely in version three, section 2.2.4.2.

2.2.3.3 Additional Research Cases

Other implementations of RTOSes in hardware include the works presented in [50] and [18]. Pereira *et al.* presents a hybrid implementation of FreeRTOS by migrating critical services to the hardware. Additionally, it also makes use of the Fast Simplex Link (FSL) bus support of Xilinx's Microblaze soft-core processor to connect the latter with the created hardware accelerators. The work in Pereira *et al.* [50] improves system latency and jitter associated with timers and also makes the system more predictable by reducing task-management jitter. Albeit, the approach requires a redesign of the FreeRTOS software, more specifically in terms of the HAL's functions for management and synchronization of tasks. Gomes *et al.* [18] discusses an extension of the work presented in [50] that details a bit more on the offloading of FreeRTOS into the hardware. As additional hybrid programming models based on RTOS one has SEOS [46] and R3TOS [24]. The work in [46] focuses on improving system performance and grant adaptability compared to other hardware implemented RTOSes, e.g., by implementing network connectivity and file system functionalities. Little research has been conducted in terms of elasticity and adaptability in hybrid OSes based on RTOS. To partly fill this literature gap, the research in [24] identifies a programming model for high-performance computing on FPGA that also considers elasticity mechanisms based on reconfiguration, i.e., Task Resource Isolation (TRI), through DPR.

2.2.4 Linux-Based Models

This section refers to the Linux-based programming models, analyzing ReconOS version two [32] and three [1], BORPH [66], SPREAD [72], and some additional research cases.

2.2.4.1 ReconOS V2

The second version of ReconOS present in Lübbers *et al.* [32] also adds support to the Linux OS, introduces FIFOs for hardware-thread intercommunication, and introduces a common virtual address space between hardware and software threads.

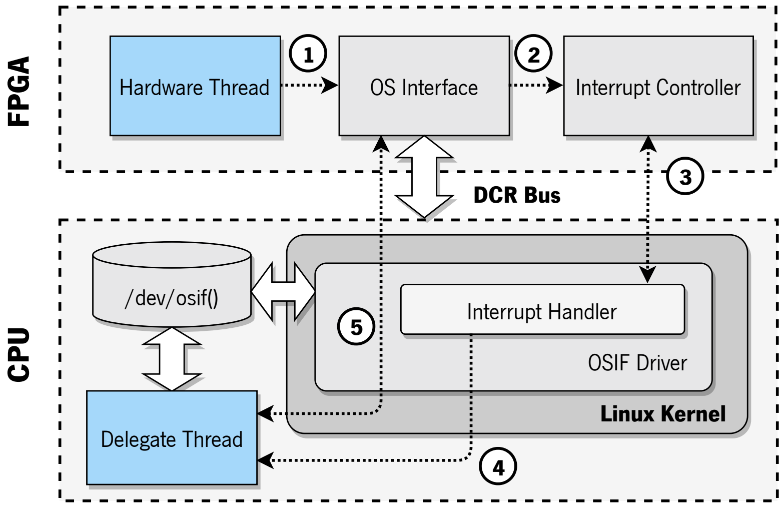


Figure 2.2.6: ReconOS: Communication between the HW and Delegate Thread in Linux [32].

With this, the programming model can now be based on the Linux kernel instead of the eCos kernel, but that brings changes to the communication framework as seen in fig. 2.2.6. The changes include creating new kernel drivers, so the delegate threads can execute privileged instructions, and changing the method for inter-thread (SW and HW) data transfers, since Linux operates over virtual addressing, and therefore the MMU is involved. Note that this was not the case on the eCos-based implementation.

2.2.4.2 ReconOS V3

The porting to the Microblaze/Linux and Microblaze/Xilkernel architectures is introduced with the third version of ReconOS [1], as well as the integration of DPR-enabled elasticity. The use of Dynamic Partial Reconfiguration (DPR) by ReconOS can be seen in fig. 2.2.7 with the implementation of reconfigurable hardware slots that host the hardware threads. Additionally, this version [1] also makes a few changes in the system by adjusting the OSFSM and introducing the Memory Interface (MEMIF), as it can be seen in fig. 2.2.8. In conclusion, Lübbers *et al.* and Agne *et al.* [33, 34, 32, 1] introduce new concepts that add upon hybrid OSes' programming models but with the trade-off of higher system latency associated with the processing of hardware-involving OS calls, given that these calls require context switches and interrupt processing. Despite this, the separation of hardware threads into a data-centric fully parallel part and a control-oriented sequential part allows for a more organized scheme that facilitates world crossing over the software-hardware boundary.

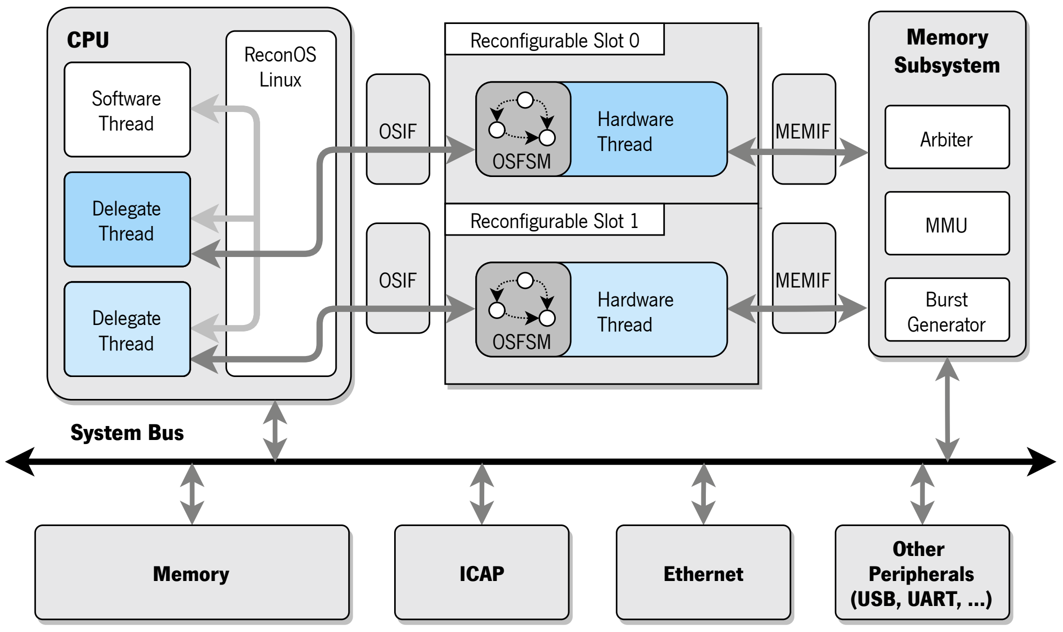


Figure 2.2.7: ReconOS: Example Hardware Architecture with DPR [1].

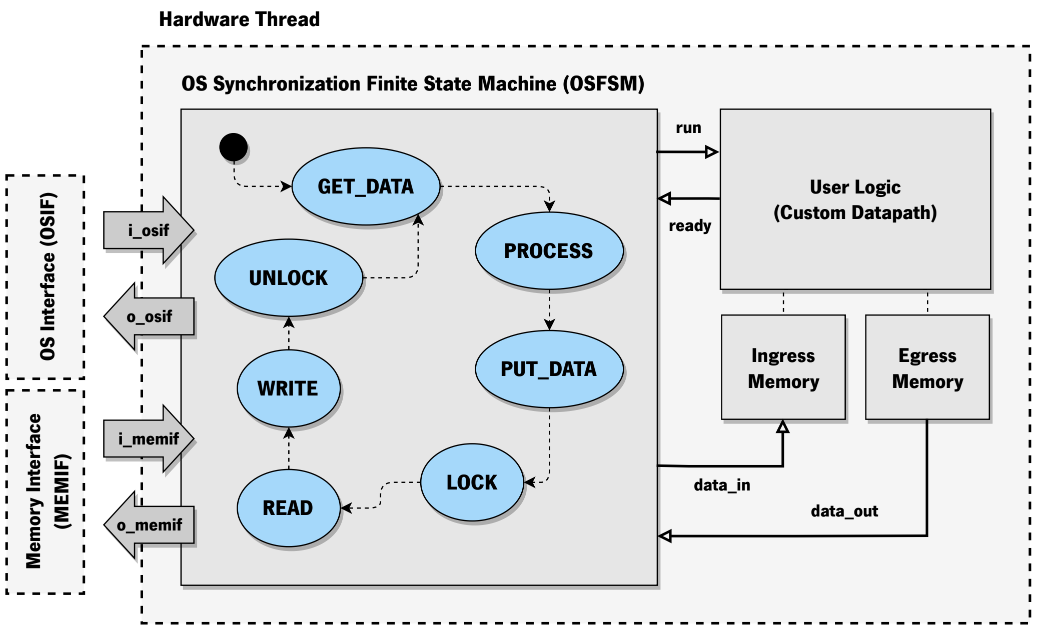


Figure 2.2.8: ReconOS: Communication between HW Thread and OSIF, adapted [1].

2.2.4.3 BORPH

In short, the literature pertaining to BORPH [9, 66] introduces another programming model, for FPGA-based systems, that unifies the hardware and software, but within a common runtime environment. The work is based on a UNIX kernel interface and therefore speeds up the development stage and promotes portability. BORPH advocates for the integration of hardware and software tasks within a familiar and language-independent framework, but also promotes the idea of granting OS runtime support services, e.g., file system access, to hardware and software tasks. The model can be categorized as coarse-grained due to the homogeneous handling of hardware and software processes at the kernel level.

The research establishes the concept of hardware process, i.e., a hardware design that behaves like a common UNIX process with access to the file system, standard I/O, and communicates through UNIX IPC mechanisms, e.g., file I/O, signal, and pipe. Similar to ReconOS [33, 34, 32, 1], BORPH provides a set of APIs that aid hybrid application designers by abstracting the complexity of kernel and userspace interactions. As the latter extends the Linux OS to provide this kind of runtime support, it is also favored by the possibility of using well-known Linux-based applications, e.g., for testing and benchmarking. The modifications include, for example, added BOF file support, the addition of APIs that allow for the load of hardware regions as kernel modules, the creation of a hardware-dedicated thread that handles the HWR allocation and configuration, software-hardware communication through packets, hardware-aware process

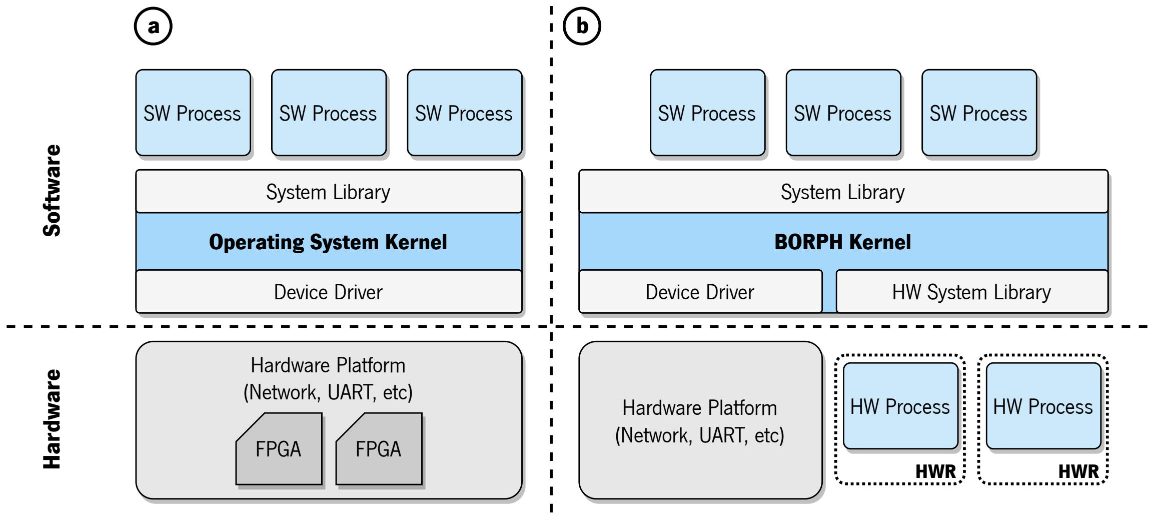


Figure 2.2.9: BORPH: a) Traditional coprocessor-based FPGA system: SW-HW has a master-slave relationship; b) BORPH-managed FPGA system: SW-HW has a peer-to-peer relationship; adapted [9, 66].

scheduler and signal handler, and software *fringes*, i.e., a software process that performs blocking file I/O operations on the behalf of a certain file system-blocked hardware process in an asynchronous manner. These software *fringes* are analogous to the concept of delegate threads in ReconOS [33, 34, 32, 1] in the sense that a piece of software is operating on the behalf of a particular executing piece of hardware. In fig. 2.2.9, one can see a typical BORPH-managed FPGA system compared to a traditional coprocessor-based FPGA system (mentioned in section 2.2.2).

In contrast to traditional approaches, BORPH does well by separating reconfigurable hardware from the hardware support platform, by creating the concept of reconfigurable Hardware Region (HWR). These regions can execute hardware designs and are denoted as hardware processes. The hardware processes are created executing BORPH Object Files (BOFs), i.e., executable binary files, that have some stipulations regarding the configuration of hardware regions, and that specify how to create the virtual files associated with the latter. This way, when the virtual file is accessed, the kernel establishes communication with the hardware and sends a packet message of the corresponding access. The works in [9, 66] also propose a Simulink-based hardware design flow for BORPH-based systems.

All things considered, BORPH allows for an improvement of hybrid pipes' latency and throughput by twelve and fourteen percent, respectively. As a downside, it does not explore Dynamic Partial Reconfiguration (DPR), but only Partial Reconfiguration (PR). Additionally, it does not employ adaptability techniques and, thus, shows conflicts in keeping up with the evolution of the Linux kernel, mainly due to extensions proposed onto the Linux source, i.e., these extensions, e.g., adding the BORPH binary files, requires adjustments in the kernel source every time there is a new Linux patch [62]. Ultimately, this OS uses what can be considered as a rudimentary communication mechanism, i.e., no software-hardware interface is established, e.g., an AXI-based interface, and the system still behaves as a passive slave since hardware processes are mapped onto the Linux folder where the software processes also exist [62]. This surely brings problems related to not having a fixed position for the resources and features of the hardware architecture, complicating a lot the job of the running user application [62]. Considering this, BORPH is a hardware-and-software-as-peers kind of OS, and not a software-as-master and hardware-as-slave OS. Due to the disadvantages of this peer-to-peer relation between hardware and software, one would still argue that an interchangeable master-slave communication would bring more advantages than disadvantages, i.e., the hardware working as master and slave and the software also working as master and slave depending on the objective. This type of mold-breaking communication mechanism, i.e., that deviates from the passive coprocessor-like model for the hardware, is present in the investigation pertaining to [62].

2.2.4.4 SPREAD

SPREAD [72] is a software-hardware unifying architecture that allows for partial reconfiguration directed to streaming applications, runtime software-hardware switching, and exploitation of data and thread parallelism. For this, the authors propose: (1) a high-throughput streaming channel, (2) a method to switch between software and hardware threads at runtime, and (3) a programming library to facilitate the design of hybrid streaming systems.

Compared to the other related works, SPREAD is more oriented for streaming applications, it presents a higher throughput streaming channel than other non-streaming architectures, e.g., BORPH supported streaming through file I/O, but the bandwidth of the latter was narrow, and others like Hthreads [4, 3, 49] or FUSE [23] base themselves on a system bus architecture, which is not directed towards streaming. Additionally, [72] allows for dynamic hardware thread creation and termination, and also a dynamical connection of the two streaming channels at runtime, depending on thread data dependency. This approach also resolves the potential problem of system data flow bottlenecks existent in system bus architectures like Hthreads [4, 3, 49] and FUSE [23], and adds more flexibility for inter-thread communication with multiple full-duplex channels between threads, opposed to the statically allocated FIFOs in ReconOS, within the OSIF [33, 34, 32, 1]. Figure 2.2.10 represents an example SPREAD system architecture.

The implementation of switchable threads is performed by having a software and hardware version of the same thread with the same behavior. In the moment of switching, the reconfiguration capabilities allow for dynamic resource allocation, context transfer, and stream redirection. This clone is created at the same time as the hardware counterpart, and it is named *stub* thread, both the *stub* thread and the hardware thread are encapsulated as a *switchable* thread. The concept of stub thread can be compared in a coarse manner to the concept of the delegate thread introduced in ReconOS [33, 34, 32, 1] and the concept of software *fringe* introduced in BORPH [9, 66]. Regarding the view over software and hardware threads, it is basically the same as portraited by the former literature, as both thread types are viewed in the same manner by the system designer with a unifying API, but adding support to pthread-compatible APIs.

Regarding disadvantages, the SPREAD architecture is mainly streaming-focused, which makes it too specific for a wide range of applications. More significantly, the architecture does not present error checking at the hardware thread-level nor adaptation strategies at the operating system kernel level, which is noteworthy given the scope of the dissertation. It only mentions these aspects in the future works section. Additionally, even though this OS improves over the traditional hardware acceleration, it still lacks ways to shift the slave coprocessor paradigm [62].

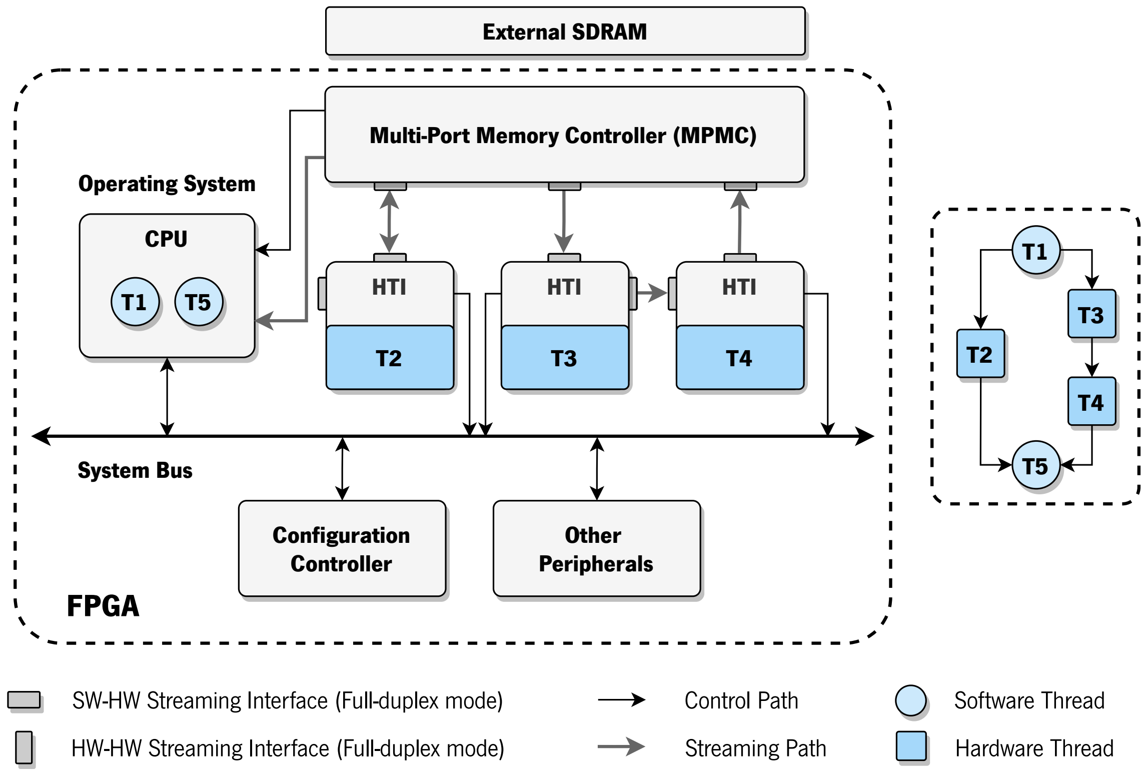


Figure 2.2.10: SPREAD: Example System Architecture, adapted [72].

2.2.4.5 Additional Research Cases

Considering Linux-based hybrid programming models, one can identify a few more research cases. FUSE [23] presents a framework for the abstraction of hardware accelerators, with a customizable interface between hardware accelerators and the OS kernel via Loadable Kernel Module (LKM). FISH [45] introduces a framework that allows hardware accelerators to make system calls via a FISH kernel module (kmodule), which talks to a host Linux system on the behalf of the hardware accelerator. RIFFA [25] is a open-source framework that includes a C library, Linux device driver and IP cores on the FPGA, connected by a PCIe bus. It presents another work that aims to bridge the gap between hardware and software, allowing for multiple accelerator management. FOS [69] is a hybrid operating with a modular development flow, two APIs for hardware abstraction with static accelerators and multiple partially reconfigurable accelerators, and a daemon for dynamic acceleration request. It can adapt to dynamic workloads and supports dynamic resource allocation, allowing for multiple types of hardware accelerators and runtime accelerator switching. RACOS introduces a hybrid OS for reconfigurable hardware accelerators that uses DPR. HAL-ASOS [62] is also another hybrid operating system based on Linux, but due to its importance, the discussion of the latter will be done in chapter 3.

2.3 Microcode Fundamentals

Microcode acts as an abstraction layer between the hardware of the CPU and the user-visible Instruction Set Architecture (ISA) instructions, being mainly used in CISC implementations. This section will explore microcode concepts to clearly define the microprogrammed approach to control unit design, ending with a broad analysis of related works.

Contents

2.3.1 Control Unit Design

Control unit design is one of the most important subjects in the field of embedded systems. This sole component can optimize the organization of the entire system by establishing the sequence of operations to perform, and also what must be executed in each operation. Not to mention that the latter can also apply to hardware and software design, bridging the gap between both, e.g., a state machine implemented in software can be simply transposed to hardware if one decides to offload it. A hardware control unit is responsible for receiving inputs that dictate the state transitions, and in each state activate the control signals necessary. The implementation of the control unit can follow two main approaches, the hardwired approach, and the microprogrammed/microcode approach, which will be explained in section 2.3.1.1 and section 2.3.1.2, respectively. Figure 2.3.1 depicts a simple FSM-based control unit.

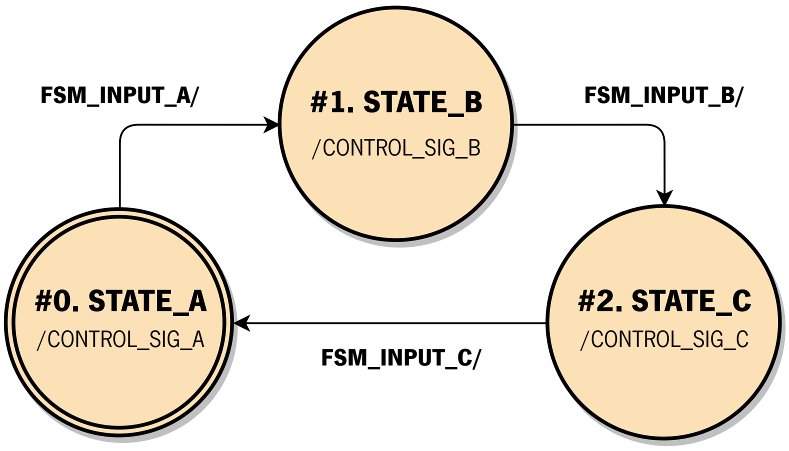


Figure 2.3.1: Example FSM Control Unit.

2.3.1.1 Hardwired Approach

The hardwired approach to control unit design is the most common when referring to hardware, as the latter is merely a Finite State Machine (FSM) and it is easier to implement. This type of control unit implies that its inputs describe the next state to go in the next clock cycle, and that, in each state, the selected outputs are effectively activated. As so, it has the functions of producing control signals that activate a certain datapath element, e.g., a multiplexer (M0) or flip-flops (FF0), activate control signals on the system bus, e.g., an AXI interface, generate control signals on an inter-module interface, or control signals in an internal module, e.g., a counter. An example module, with a FSM control unit, is depicted by fig. 2.3.2. The forementioned control unit behaves accordingly to the state diagram of fig. 2.3.1.

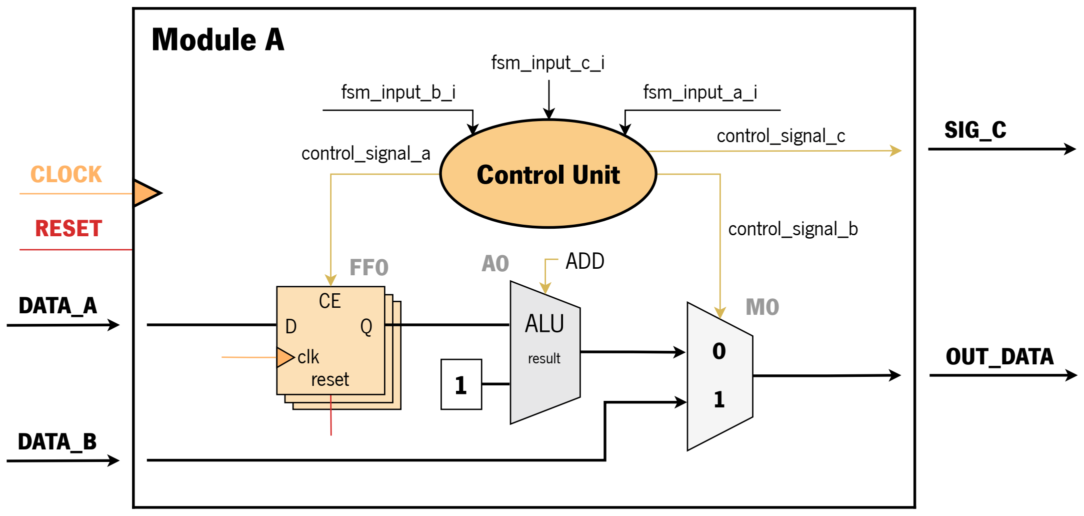


Figure 2.3.2: Example Module with a FSM Control Unit (Simplified).

2.3.1.2 Microprogrammed/Microcode Approach

A microprogrammed control unit fetches low-level microinstructions from a microcode memory or control store and decodes those microinstructions to determine the active control signals for a single clock cycle, specifying the system's behavior during that time frame [65]. The control signals outputted depend on the flow of microinstructions dictated by a series of test input signals [58]. The input signals to test are also specified by a certain field in the microinstruction. A microprogram control unit consists of two main components: the microcode memory, associated with its field decoding logic, and what is sometimes referred to in the literature as the microprogram sequencer [35], which job is to generate the next address in the microcode memory. This scheme can be seen in fig. 2.3.3. Mano [35] denotes the following address-sequencing capabilities in a **microprogram sequencer**:

- Increments the present address for control memory;
- Branches to an address as specified by the address field of the microinstruction;
- Branches to a given address if a specified status bit is equal to one;
- Transfers control to a new address as specified by an external source;
- Has a facility for subroutine calls and returns.

All the trade-offs for supplying these capabilities must be taken into consideration when designing the microprogrammed units of the dissertation.

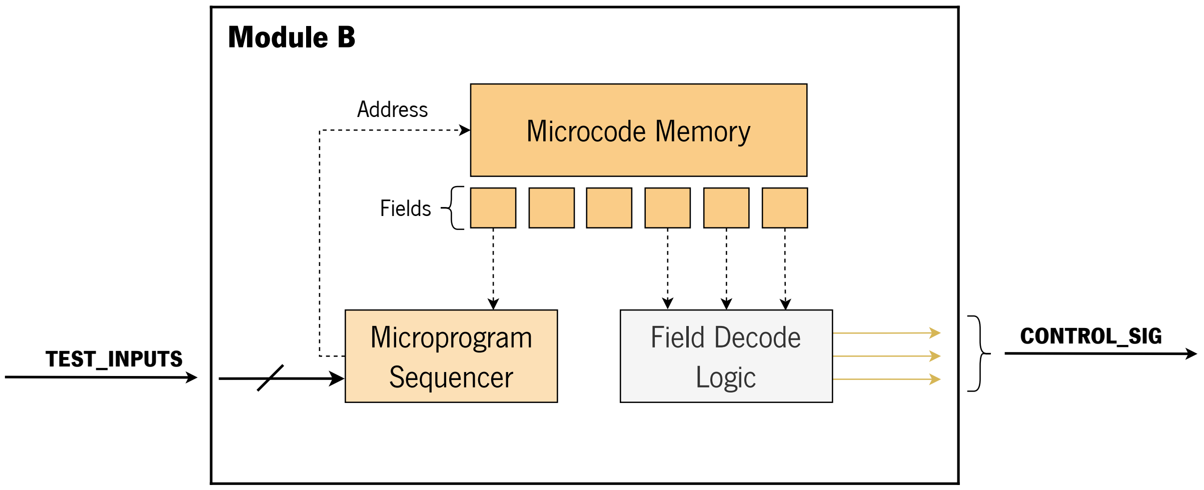


Figure 2.3.3: Example Module with a Microprogrammed Control Unit (Simplified), adapted [58].

The microcode approach implementation is easier and less error-prone than the hardwired approach, which uses complex logic for the sequencing of operations. On the contrary, the microcode approach uses basic logic for sequencing and decoding the control signals. The hardwired control unit tends to be faster than the microcode one, in some cases. Despite this, microprogramming is still popular to implement CISC architectures, e.g., the x86 architecture, while hardwired control units are more common in RISC architectures, e.g., Pentium Four used a hybrid approach [67], since it had most RISC-like instructions implemented in a hardwired approach and some other instructions implemented in microcode. Siewiorek *et al.* [60] identified some key advantages of **microprogramming**:

- **Regularity:** microprograms are easy to debug and maintain;
- **Extensibility:** it allows the addition of new features, or replacement of existing ones, e.g., through microcode updates [62];
- **Flexibility:** easy to add to new features and reduced design effort;
- **Cost-effectiveness:** can implement complex designs reducing the hardware cost in terms of resources (important if one intends to minimize area [27]).

Microinstructions. Microinstructions are control words with unique memory addresses in the microcode memory (ROM or RAM), having a particular field for specifying the control signals to activate and a field to indicate the next microinstruction to execute, given by the boolean test result, i.e., true or false, of a certain jump condition (JC), e.g., unconditional, the overflow of a counter, a module-external signal.

Microinstruction Encoding. In terms of literature taxonomy, the microinstructions can be classified into the following categories [60]:

- **Vertical:** completely encoded, fig. 2.3.4 (Bottom);
- **Horizontal:** partially encoded or no encoding, fig. 2.3.4 (Top).

On the one hand, vertical microinstructions tend to be compact, and specify a complete or maximal encoding, so multiple control signals are encoded and associated with each bit pattern. Each bit pattern is connected to a function code that performs a series of datapath operations in a sequential manner. In the vertical microinstruction paradigm, there is a trade-off between less microcode ROM memory usage and additional decoders, due to the more compressed microcode. However, this trade-off is still viable in most cases as a result of the encoder's cost being less than the cost of larger memory elements. e.g., ROMs [2]. On the other hand, horizontal microinstructions tend to be longer and minimally encoded, resulting in less hardware compared to vertical microinstructions. A subfield of a horizontal microinstruction might control a data function directly, e.g., one bit for each system bus control line and one bit for each internal control signal. Generally, despite the length of microinstructions and sequences, the goal is always directed towards reducing the microcode memory footprint, i.e., reduce the microcode store usage.

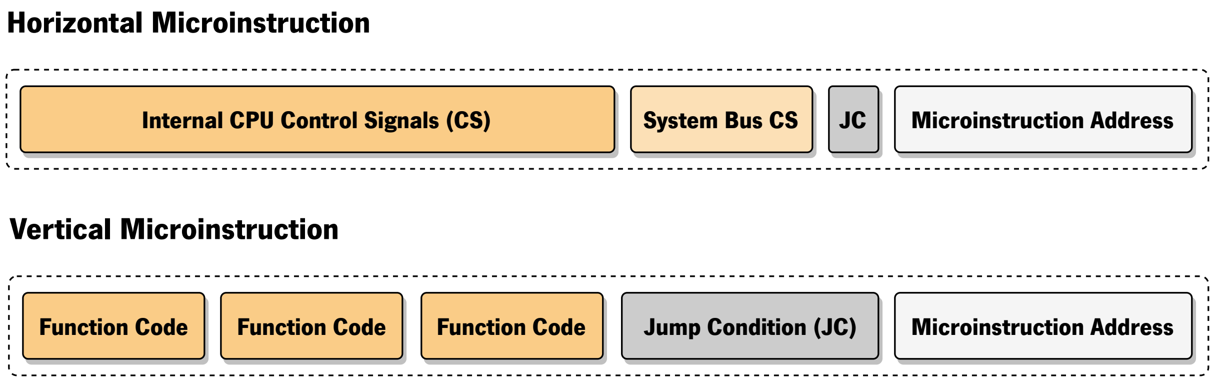


Figure 2.3.4: Horizontal vs Vertical Microinstructions, adapted [67].

2.3.2 Related Works

Wilkes introduced the term microprogram in 1951 as a novel approach to control unit design that simplified and added flexibility over the hardwired approach [73]. Although, the concept only became popular in 1964 with the introduction of IBM's System/360. The system used microprogramming to implement bidirectional (upwards and downwards) compatibility across different System/360 models [17]. By the mid-to-late seventies, eight-bit microprocessors, e.g., Intel 8080 and Zilog Z80, sixteen-bit microprocessors, e.g., Intel 8086, and thirty-two-bit microprocessors, e.g., Motorola 68k, were already microcode-based [65].

Originally, microcode was stored in read-only memory, yet later microcode updates were introduced to mitigate hardware errors, e.g., Intel Pentium floating-point division bug in 1994 [53], to deploy new security measures, e.g., Intel SGX [12], and also to protect the CPU from attacks, e.g., Spectre and Meltdown. Spectre consists of attacks that focus on hardware vulnerabilities rather than software ones. Regarding this, the attacks rely on techniques as (V1) bounds check bypass, (V1.1) bounds check bypass store, (V1.2) read-only protection bypass, (V2) branch target injection, (V3) rogue data cache load, i.e, Meltdown; among others [20]. Intel and AMD deploy microcode updates since the nineties, with the Pentium Pro (P6 microarchitecture) in 1995, and the K7 microarchitecture in 1999, respectively [29]. Since then, numerous research has been conducted in terms of microcode customization, security, and other applications.

2.3.2.1 Microcode-Level Customization and Security

Kollenda *et al.* [28] examines the reverse engineering of a COTS AMD x86 CPU microcode. Regarding this, the work explores the microcode customization and microcode-assisted defenses, proposing a myriad of defense techniques as case studies, e.g., limiting the resolution on user-accessible timers to mitigate timing side-channel attacks; implementing a microcode-assisted address sanitizer that inserts new instructions that perform checks and instrumentation for allocation and deallocation, during the compilation process, to reduce temporal faults, e.g., use-after-free bugs; performing microcoded instruction set randomization, i.e., randomizing the instruction encoding and breaking the link between x86 operation and its semantics to decrease code-reuse attacks, e.g., Just-in-Time (JIT)—Return-Oriented Programming (ROP) attacks; adding a microcode-assisted instrumentation scheme that generates microcode updates to intercept the execution flow and redirect it to an address where instrumentation is performed, resuming execution afterwards; developing authenticated microcode updates to thwart attacks, e.g., microcode trojans; introducing a trusted execution environment via µEnclave similar to Intel SGX [12], to assure that even code with kernel-level privileges cannot interfere with a certain enclave program. The enclave functionality is based on a separate Instruction Decode Unit (IDU). Similarly to [28], in [29], Koppe *et al.* discusses the reverse engineering of x86 processor microcode. Specifically, the work focuses on the reverse engineering of the AMD's K8 and K10 microarchitectures, and explores further how to develop custom microcode updates to change the CPU's behavior. Koppe *et al.* also considered the potential of microcode technology to implement CPU-assisted instrumentation for system defense based on the augmentation of preexisting instructions, and how malicious microcode updates can be the source of bug attacks, i.e., attacks based on innate computation bugs, and timing attacks, i.e., attacks based on the meticulous analysis of execution time. While [28] and [29] focus on the reverse engineering of microcode related to AMD processors, the work presented by Yang *et al.* in [74] turns its attention to the reverse engineering of Intel's microcode update structure to investigate security and form a basis for future research. Additionally, still concerning Intel processors, the work presented by Chen *et al.* [11] explores x86 microcode in what relates to security aspects.

Since microcode is mainly proprietary, works that focus on it need to reverse engineer the microcode of existing architectures [28, 29] or build their custom microcode mechanisms [2, 63, 62]. Considering this, Albartus *et al.* [2] discusses the design of a RISC-V architecture alongside a microcode development and evaluation environment to design microcode trojans. This work focuses more on the effect of malicious microcode on embedded CPUs.

The CHEx86 architecture [59] extends the well-known x86 architecture to include an instrumentation mechanism that targets spatial and temporal memory safety vulnerabilities, e.g., use-after-free, double-free, and uninitialized reads. Concerning this, the architecture implements this instrumentation by re-routing the translation of native macro-operations to the microcode RAM, which has custom microcode instrumentation translations. The work also discusses the shadow capability, i.e., the capability of tracking (statically/dynamically) allocated and deallocated memory, and the speculative pointer tracking capability.

2.3.2.2 Additional Research Cases

The work in [30] introduces the implementation of a microcode-programmable learning engine, i.e., each neuron of a spiking neural network is programmable with microcode instructions. Besides, in the future works section, Kollenda *et al.* [28] discusses the benefits of lightweight system calls implemented in microcode. Kollenda *et al.* does not implement this type of system calls, but the work by Silva *et al.* [63, 62] does. Although, since the work presented in Silva *et al.* [63, 62] is of major importance to the groundwork of the dissertation, the implementation of hardware system calls will be addressed later in the appropriate chapter. Even though the aforementioned works focus on microcode mechanisms, they cannot be directly compared to the goal of the dissertation, except [63, 62], as it acts as groundwork.

2.4 Discussion

This section will discuss the state-of-the-art gaps, considering the three main sections presented in chapter 2 (section 2.1, section 2.2, section 2.3). Table 2.2 presents a comprehensive gap analysis between some hybrid operating systems like HAL-ASOS [63, 62].

As it can be seen in table 2.2, most of the recent hybrid operating systems do not explore enough the concepts relating to adaptability mechanisms, and microcode. Generally, the literature neglects the matter, or fails to provide solutions for hybrid OSes reconfiguration without the use of a post-design technique, i.e., they typically rely on DPR. With this in mind, the exceptions are encountered in ReconOS [1] with a built-in TS mechanism based on DMR that permits different levels of system call verification, parameter checking, and memory access checking [40, 41, 39]. Although, the scheme falls onto the fault tolerance scope, whilst the part referring to the system upgrade relies on post-design DPR.

Another gap in the literature is the use of microcode. As one can observe (table 2.2), only HAL-ASOS [63, 62] focuses on this approach within a hybrid operating system. ReconOS [1] employs procedure-based system calls similar to HAL-ASOS kernel-level procedures, but to perform simpler tasks and without using microcode mechanisms. Overall, a system would benefit from the built-in microcode update features and hardware system call scheme that HAL-ASOS [63, 62] introduces, paired with the error detection and correction strategy of ReconOS with some type of design diversity, i.e., different implementations of redundant modules, to protect the system against CMF, i.e., modules that are affected equally by the same fault. This could be achieved using a dual-task lockstep scheme for error detection and correction within HAL-ASOS [63, 62] similar to the TS of ReconOS [40, 41, 39].

All of these considerations open up an opportunity to innovate in relation to the literature, by possibly maintaining the update scheme and hardware system calls of HAL-ASOS [63, 62], and improving some aspects of the latter, e.g., procedure scalability, extended feature interface, just to name a few. More on that topic in section 3.1.5. Note that most of the hybrid programming models advantages and disadvantages, were already compared in section 2.2, as well as the microprogrammed approach advantages and disadvantages in section 2.3.

Table 2.2: Gap Analysis among hybrid programming models.

	Based On			**Elasticity**	**Adaptability**		**Diversity**[5]	**µCode**
	RTOS	Linux	Other	Support	Method	Level		
Hthreads [4]	✓	✗	✗	✗	✗	✗	✗	✗
ReconOS [1]	✓	✓	✗	DPR	LST/DMR[1]	HWTL/RTL	T[3]/I[4]	✗[7]
SEOS [46]	✓	✗	✗	✗	✗	✗	✗	✗
R3TOS [24]	✓	✗	✗	PR	TRI	MSL	✗	✗
RE2DA [51]	✗	✗	✓	PR	CP/RB	MSL	✗	✗
BORPH [66]	✗	✓	✗	✗	✗	✗	✗	✗
SPREAD [72]	✗	✓	✗	PR	✗[F]	✗[F]	✗	✗
FUSE [23]	✗	✓	✗	PR	✗	✗	✗	✗
FISH [45]	✗	✓	✗	✗	✗	✗	✗	✗
RIFFA [25]	✗	✓	✓	✗	✗	✗	✗	✗
FOS [69]	✗	✓	✗	DPR	RES[6]	SL	✗	✗
RACOS [70]	✗	✓	✗	DPR	✗	✗	✗	✗
HAL-ASOS [62]	✗	✓	✗	BI/MP[2]	RBM[S]	HWTL/MSL[S]	✗[S]	HWSC[S]

(PR) Partial Reconfiguration **(BI)** Built-In **(MP)** Microprogrammable **(LST)** Lockstep Shadow Threads **(DMR)** Dual Modular Redundancy **(TRI)** Task Resource Isolation **(CP/RB)** Check-Pointing and Rollback **(HWTL)** Hardware Thread Level **(MSL)** Module and System Level **(SL)** System Level **(RES)** Resource-Elastic Scheduling **(RBM)** RAM-Based Microprogram **(HWSC)** Hardware System Calls **(T)** Time Diversity **(I)** Implementation Diversity **(M)** Microarchitectural Diversity

1 Considering the Loosely-Coupled Thread Shadowing in [40].

2 Tightly-Coupled elasticity with the evolution of Linux OS by design, through microprogramming [63].

3 In signature level three [39].

4 Trans-modal LST in [41].

5 Diversity refers to design diversity, i.e., different implementations of redundant modules.

6 RES refers to dynamically changing the allocated resources used by an accelerator based on the workload and the availability.

7 ReconOS [1] provides procedure-based system calls, but does not employ microcode mechanisms.

F Stated in the future works section but not implemented.

S Scope of the dissertation.

3. Background

In this chapter, one presents the groundwork and background for the thesis work. Firstly, one describes the HAL-ASOS tool, upon which the dissertation's microkernel infrastructure is based. Secondly, one explains the architecture existing in the software side, developed in a parallel thesis, that handles Linux interaction on the hardware microkernel's behalf.

3.1 HAL-ASOS

HAL-ASOS [63, 62] is an embedded system design tool focused on applications that intend to be offloaded to CPU+FPGA architectures, like the ones presented in section 2.2. The latter centers on bridging the gap between software and hardware by promoting specific hardware to fundamental computing units, and breaking the state-of-the-art mold of passive co-processor-like hardware extensions. This section will firstly describe the tool's design methodology, and later explain the main hardware accelerator model, alongside with some primal computing entities. Finally, the improvement gaps are analyzed.

3.1.1 Design Methodology

The HAL-ASOS design flow follows a hardware-agnostic approach to hardware design and offloading. This means that the offloading of the hardware is deeply connected with meeting the imposed requirements with a series of thought out steps, leaving the decision of the target CPU+FPGA platform to a stage where there is a better understanding of the resources needed to perform the deployment. As seen in fig. 3.1.1, a typical development cycle with HAL-ASOS starts with the description of a new application, or an existing one, through UML and task graphs, to describe the overall system, and the relations between the functional units, respectively [62]. The next step is the software refactoring, where the initial application is restructured and transitioned to one of the frameworks that HAL-ASOS supports in terms of software, e.g., POSIX, C++, or boost-based runtime. The next step, allows the transitioned application to be profiled, identifying system bottlenecks that justify the migration of a certain software task to hardware in the form of a hardware task. With the application partitioned, one can then begin the co-design stage regarding the system parts to offload. For this, the software task is represented via a hardware task, through a set of system calls that rely on hardware kernel resources, based on the hardware accelerator model. This representation is more at the software and hardware interaction level, while the hardware task's datapath represents the actual algorithm to offload. This transition of a software algorithm to hardware can be done manually, which requires more design effort and time, or with the support of HLS tools, e.g., Vitis HLS, or Matlab to perform the C/C++ to HDL transpilation, i.e., source-to-source compilation.

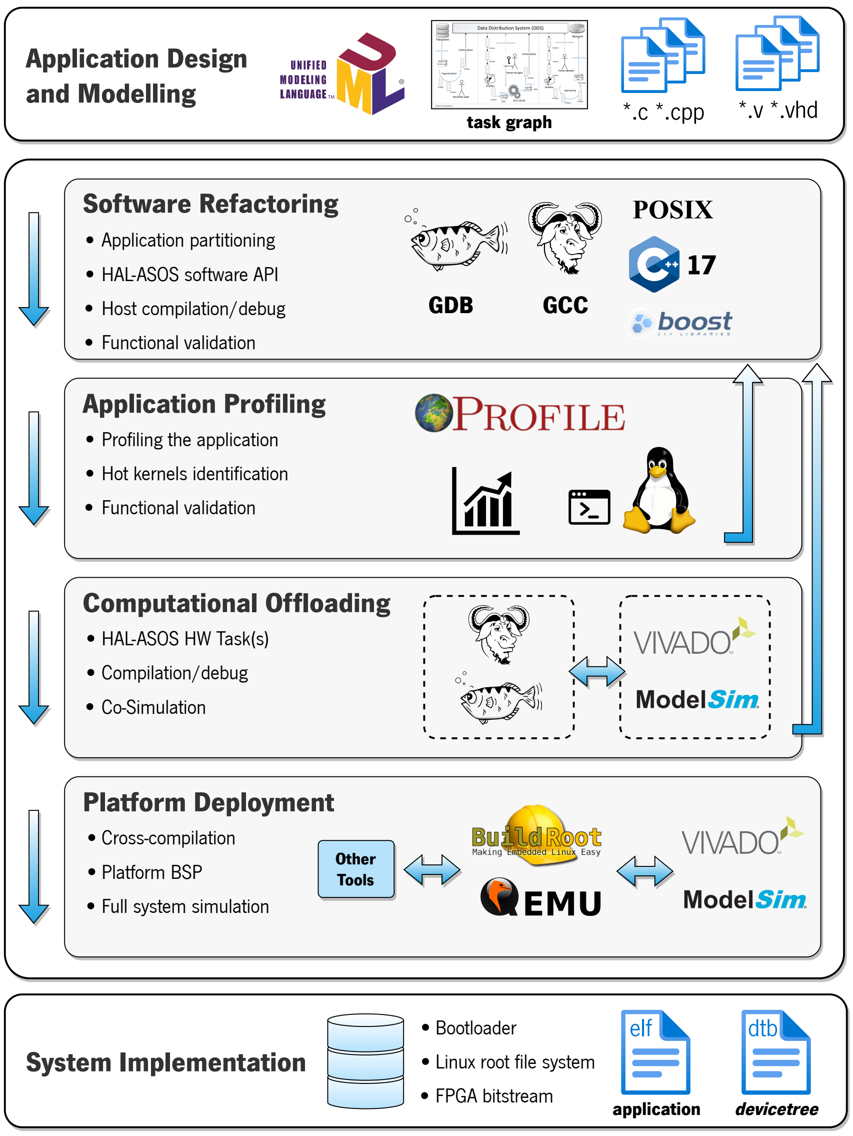


Figure 3.1.1: HAL-ASOS: Design Flow, adapted [63, 62].

The computational offloading stage culminates into the RTL simulation of the hardware accelerator (kernel) and hardware task entities within a tool like Xilinx's Vivado. The last stage is the platform deployment one, in which the designer must decide the targeting platform, typically a SoC, based on the expended resources of the overall system. This step is aided by the full simulation capabilities of HAL-ASOS, which allows for the simulation of the both system parts, i.e., hardware and software, within a QEMU environment with virtualized memory positions. For this, a custom Linux image is tailored to the application needs in Buildroot, and launched on QEMU with the platform (board) selected. This step allows for further debug of the application before actually deploying it to the SoC running Linux. The last step includes the platform validation with the comparison of initial metrics with the achieved ones.

It is important to clarify that the goal of the dissertation is not subjecting an application to this design flow stack, but to develop upon the existing hardware model, section 3.1.2, and entities, fig. 3.1.3 and fig. 3.1.4, to improve metrics and add functionality.

3.1.2 Accelerator Model

As in [63, 62], one will refer to the hardware kernel as *accelerator*, and to the entities shown in fig. 3.1.2 as *accelerator model*. The accelerator model revolves around two main entities named as hardware kernel, representing the hardware version of the software's kernel space, and the hardware task, representing the software's user space in hardware. These two entities are described in fig. 3.1.3 and fig. 3.1.4, respectively. Additionally, the accelerator model presents three distinct interfaces with the hardware kernel of HAL-ASOS, fig. 3.1.2: (1) a slave control-based interface, deemed as the control channel, (2) a slave data-based interface, deemed as the data channel, and (3) a master interface, deemed as the system channel.

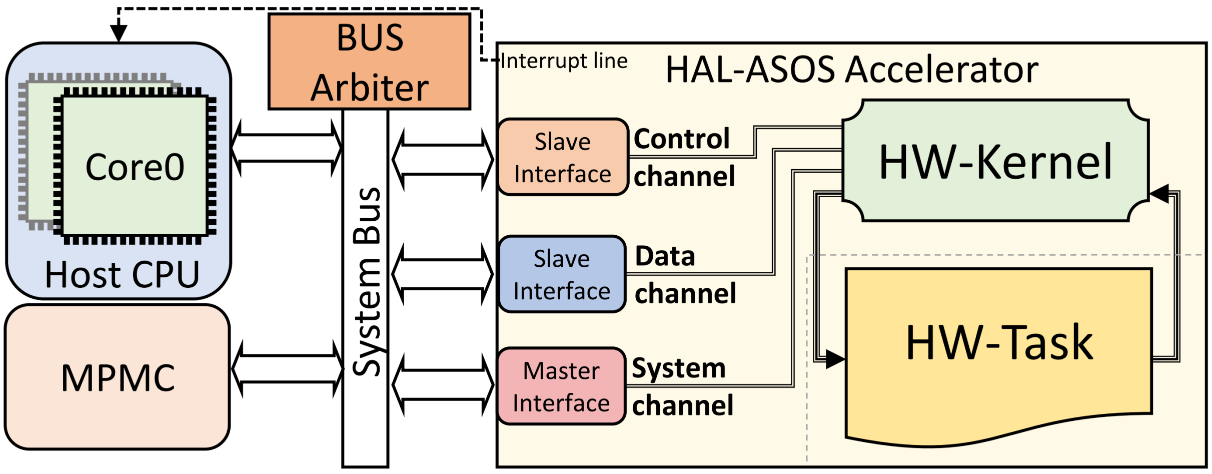


Figure 3.1.2: HAL-ASOS: Accelerator Model integrated into Host Platform, [63, 62].

The terms master and slave are applicable from the hardware kernel's point of view, and, therefore, here the paradigm of co-processor-like hardware acceleration is shifted since the hardware can now act as master through the last aforementioned interface. Depending on the accelerator model, with HAL-ASOS, one can have these interfaces implemented with a PLB or AXI bus. More specification on the purpose of the interfaces will be given in fig. 3.1.3. Finally, HAL-ASOS also has an interrupt line for synchronization with the host operating system [62].

3.1.3 Hardware Kernel

The hardware kernel of HAL-ASOS bases itself around a microcoded control unit that translates higher level system calls coming from the hardware task into lower level control signals that interact with the kernel resources present in fig. 3.1.3, or with the host system. The kernel core has a single-address microprogram entity which hosts the horizontal microcode translations of the aforementioned syscalls, e.g., ones that manipulate the writing and reading to and from the message queues, ones that write and read to and from the local memory, or that manipulate the hardware mutexes. For more information on microcode, refer to section 2.3.

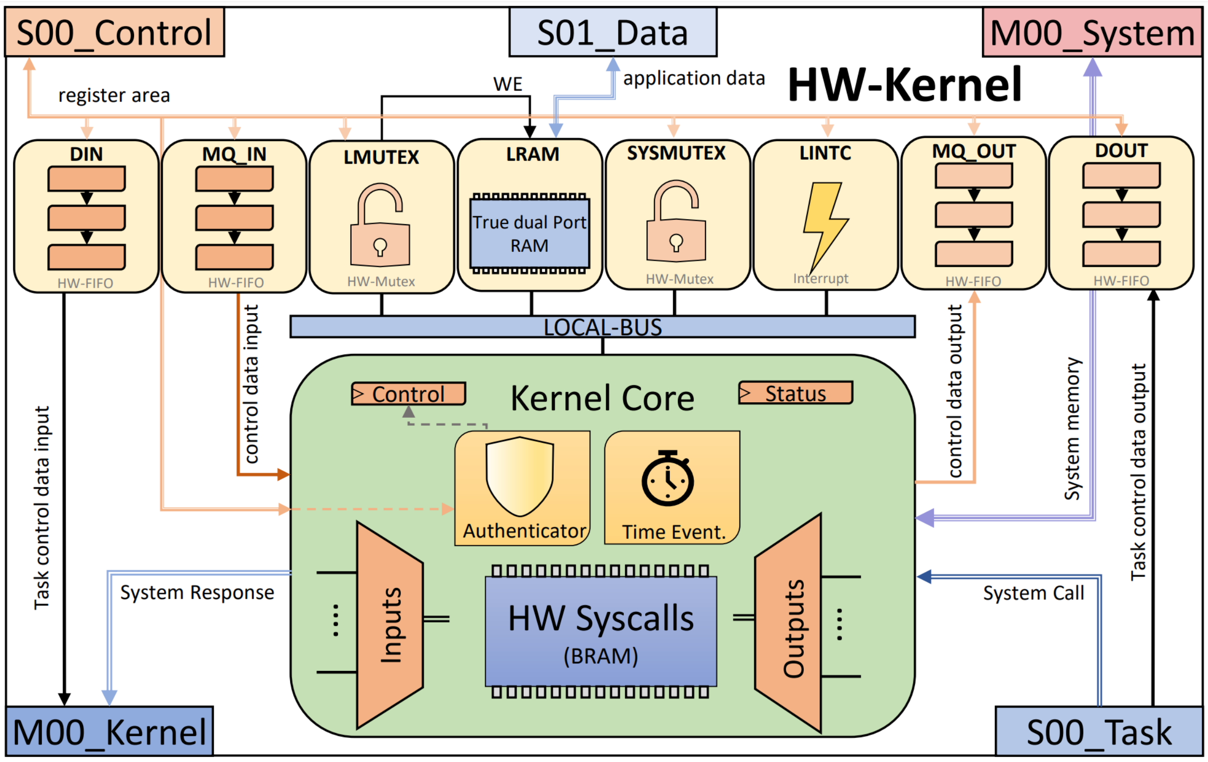


Figure 3.1.3: HAL-ASOS: Hardware Kernel Simplified Model, [63, 62].

In addition, it has a system-level datapath entity that decides the system call parameters and returns, a module that manages timing events and the sleep features of HAL-ASOS, and modules to manage the kernel's runtime, with a index managing service and a procedure scheduler. The control of the hardware kernel is made through the control and status registers of the core, made available in through the S00 Control interface, as seen in fig. 3.1.3. Although, the safe writing to kernel's control register is ensured by the authenticator module. The core also has a FSM that dictates the kernel's current state, and also indirectly the hardware task entity's state, depending on the contents of the control register. The kernel resources can be accessed from the host for read or write through the S00 Control interface, while the S01 Data interface serves as a direct communication channel connected with the local memory of the accelerator. The core also has access to the resources of the accelerator through the local bus element, which generically implements a decoder that decides the flow of data to/from the resources, the zero copy unit of HAL-ASOS, or the LRAM. The zero copy unit of HAL-ASOS comes in when one wants to make bursts from the system memory to the local memory of the accelerator or vice-versa. Moreover, the M00 System interface serves as channel between the accelerator and the host's main memory region. The kernel communicates with the hardware task through the M00 Kernel and S00 Task interfaces, which are groups of ports regarding a kernel's response or call, respectively.

3.1.4 Hardware Task

As forementioned in section 3.1.1, the hardware task entity can be divided into a control unit and datapath, like most common architectures [62]. The control unit of the hardware task determines the order of operations of the entity and manages the requests to the hardware kernel and host system. On the other hand, the datapath provides the hardware version of the software algorithm chosen to offload.

As seen in fig. 3.1.4, the entity makes use of VHDL procedures of two types: (1) user-level procedures and (2) kernel-level procedures. Typically, user-level procedures wrap the kernel ones, i.e., kernel-level procedures are called by user-level procedures. Both together define the kernel calls at any given state, while the kernel's execution results are updated through the kernel response mechanism, which is also captured by the hardware task's procedures. The kernel-level procedures abstract the microprogram system calls by specifying the system call that will be executed, its parameters, and capturing its returns, while the user-level ones usually determine the enabled services of the hardware kernel. Although, the user-level procedures also have to capture and forward the returns to the hardware task's top-level so that the results can be used in consequent states or in the same state. The hardware task makes use of VHDL procedures

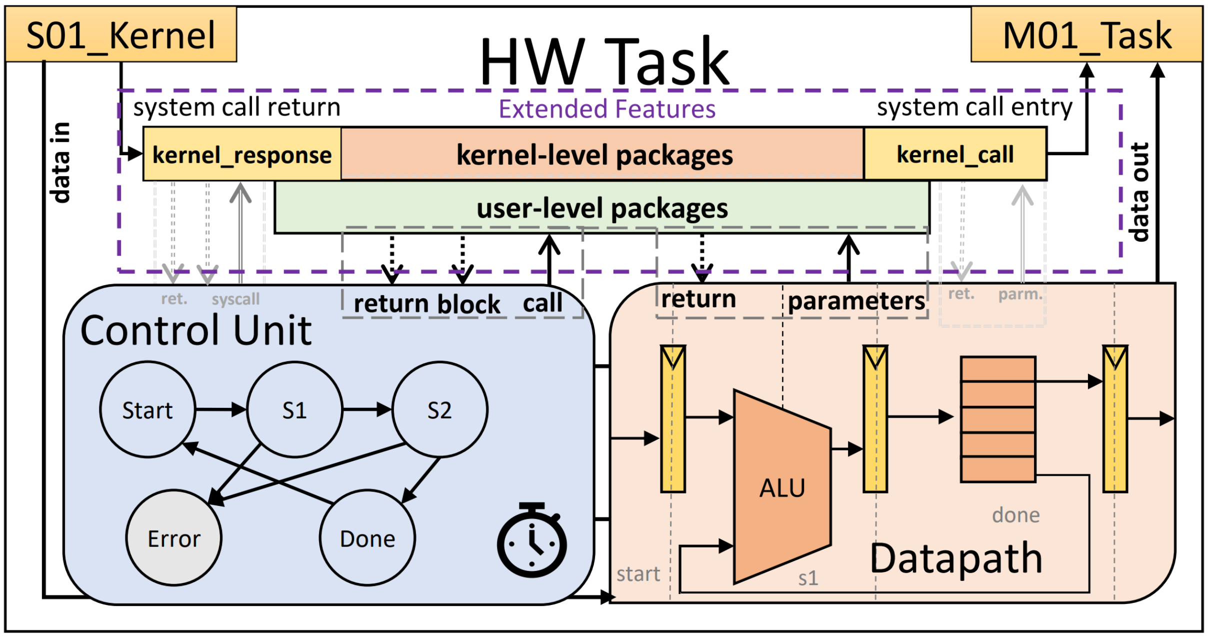


Figure 3.1.4: HAL-ASOS: Hardware Task Simplified Model, [62].

to link the signals used inside the procedure body to signals existing in the module where the procedure is called. This is doable since the working principle of VHDL procedures performs these attributions in zero simulation time, or in delta cycles. Delta cycles are non-time-consuming timesteps used by simulators to model the behavior of VHDL code. Additionally, as stated in section 3.1.3, the kernel call and kernel response are propagated to the hardware kernel through the slave kernel and master task interfaces. The hardware task's FSM might have a higher number of states depending on the functionality to implement, since the architecture only allows one user-level procedure per state.

3.1.5 Improvement Analysis

HAL-ASOS proves that it is possible to achieve an accelerator model where the accelerator is a first-class computing entity [62], focusing on microcode mechanisms to provide elasticity by design with low power consumption, but still has some limitations and openly stated issues:

- The use of one user-level procedure per state sometimes leads to the creation of user-level procedures with multiple steps to achieve functionality, with a kernel-level procedure being called in each step, leveraging the capabilities of the procedure scheduler. As the number of FSM states increases in the hardware task, the capability that allows the manipulation of external signals to the procedure, using signal drivers, rapidly becomes a liability. Multistep user-level procedures are modeled around switch statements, which are mapped to combinational logic. After synthesis, these cases can generate hardware LUTs, ROMs, or multiplexers. The procedures' zero simulation time attribution "copy-pastes" these switch statements every time a multistep user procedure is called, making the combinational logic cell usage of the hardware task to increase proportionally to the number of multistep procedures used. Thus, additions can be done to make these multistep procedures scalable.

- Although HAL-ASOS introduced and fomented the microcode approach to reconfigurable architectures based on CPU+FPGA, it still left a lot to explore since the microcode mechanism employed can generate runtime errors due to a mismatch between the version of the microcode in the hardware kernel and the system calls invoked. Additionally, the framework does not support the compilation and addition of new syscalls by the host system, which can hinder its evolutive elasticity premises if the system becomes unusable. This leaves room for improvement in terms of update strategies for the microprogram, and also in terms of its means to adapt to different runtime errors, e.g., use-after-free, or in other words, its means of coping with diversity.
- Considering all the recent versions of HAL-ASOS [62], specifically version 4 (that uses an AXI bus), it is possible to perceive that the master interface, deemed as M00 System, and which establishes a memory channel to the host system's main memory is always implemented with an AXI4 Lite Interface. Improvements on the architecture might be added by investigating the migration of this channel to AXI4, unveiling the possibility of making main memory burst transfers without the intervention of the ZeroCopy Unit (ZCU). Additionally, this memory burst feature can be utilized to simplify the extended feature scheme presented by HAL-ASOS.
- For the time being the accelerator allocates all the memory it needs from the kernel space, this can hinder future designs that require large amounts of contiguous memory. The memory could be allocated as blocks of various sizes like in GPUs, and a hardware solution could be proposed to handle those scattered blocks as a contiguous memory region, through a scatter-gather mechanism.
- The accelerator model does not have a way to deal with hardware multitasking, which means that in a scenario of, for example, two hardware tasks, representing two offloaded algorithms, one has to partition the design into two separate hardware kernels each one controlling its hardware task, with some sort of mechanism to handle inter-task communication. Developing an arbiter with scheduling

policies that allowed one hardware kernel to receive requests from more than one control unit would solve the problem.

- The HAL-ASOS accelerator model does not perform syscall parameter and bounds checking and does not have a way to bank recently used system calls. A caching mechanism like the one in [64] could be employed to resolve these issues. Although, there is a mechanism to recover from an unregistered address failure with a memory allocation request to the host system.
- The work in [19] advocates that an unmasked physical source, generated by radiation effects, may cause a fault that manifests itself with a certain probability as a single or multiple bit-flip in a certain register or memory cell. This type of error is different since it affects the underlying hardware, and at the moment HAL-ASOS is not ready to deal with these type of errors. A lockstep scheme with two hardware tasks might resolve the issue by applying some redundancy like in [40].

3.2 KIVIO

KIVIO or Kernel Interface for Virtualized I/O is the software architecture developed on a parallel thesis [42], that operates at the Linux kernel-level on the microkernel's behalf. Thus, it solely links the multi-threaded software application with the hardware task, as seen in fig. 3.2.1 task graph with the entities APP and HWT, respectively. Without going into too much depth about the inner workings of the architecture, since it is out of this dissertation's scope, the most relevant thing to note is that KIVIO relies on a Linux kernel FIFO structure, KFIFO, purposely (and natively) mapped to certain positions of the host system's main memory. This way, the hardware task has the possibility of requesting specific commands to KIVIO that it would not be able solely perform, e.g., opening a Linux pipe, binary file, or socket. For this, it is necessary that the hardware task's packages comply with the established formatting of the 64-bit commands, and perform the writing of the commands to shared memory. In the same way, via the memory-mapped KFIFO, the hardware task receives the command results, e.g., a descriptor for opening a pipe, file, or socket, in a specific region of shared memory, dedicated to the extended system call returns. One will refer to the syscalls that perform these OS-aided features as *extended features* or *extended system calls*.

Additionally, the KIVIO architecture has direct access to selected microkernel registers through the S00 Control interface, and the microkernel's local memory through the S01 Data interface, fig. 3.2.1. With this, it is possible to control the microkernel's execution, and, consequently, the hardware task's execution, and also perform some *extended syscalls* that relate to local memory operations.

Although, it is important to note that (natively) these extended features are achieved with the trade-off of more memory operations, which, in a system with multi-level cache might originate non-deterministic access delays. Even though, this is an approach worth exploring since it simplifies the one presented by HAL-ASOS.

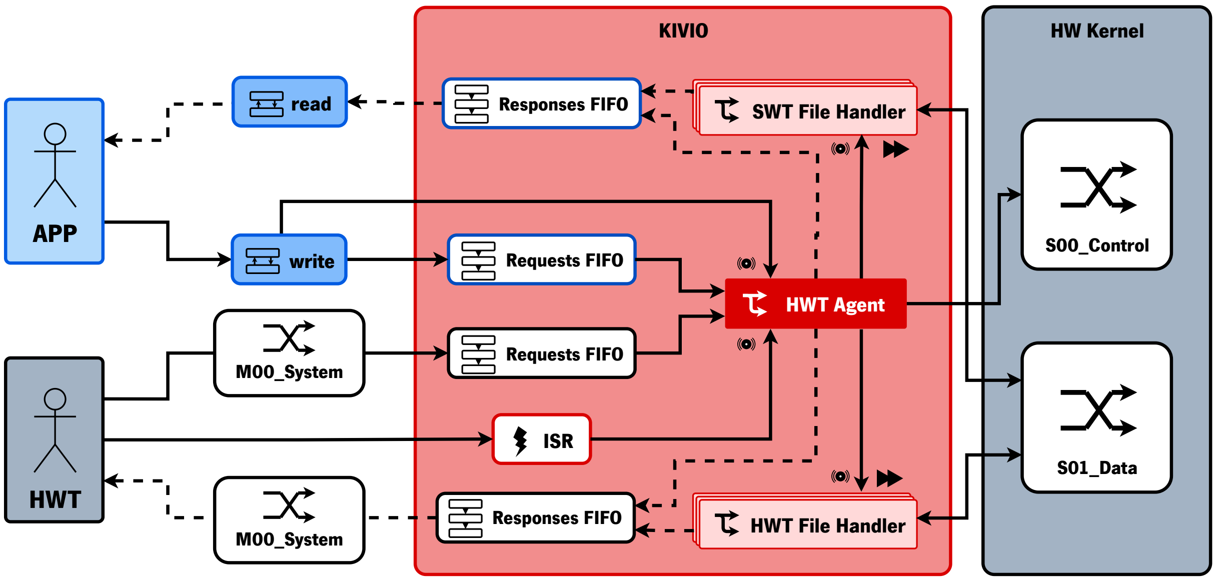


Figure 3.2.1: KIVIO: Information Flow, [42].

4. Supporting Microkernel

This section refers to the specifics about the hardware microkernel implemented to support the architecture proposed in chapter 5. Specifically, one dives deeper into the internals of the microkernel top-level module and its components, interrupt management, memory management, the hardware task's internals, and the microkernel's external interfaces to communicate with the processing system as master or slave.

4.1 Architecture Overview

The microkernel developed in this dissertation is a minimal version of the HAL-ASOS' kernel and hardware task structures, designed to ultimately achieve less combinational logic usage in the composite procedures of the hardware task's extended features, simplify the hardware kernel's and supporting OS (Linux) interactions to perform these extended features, and expand on the microcode update functionalities that HAL-ASOS brings forward in its research [63, 62]. With this in mind, the microkernel's internal scheme is kept somewhat similar to HAL-ASOS, maintaining some essential architecture modules, redesigning others, and adding new ones. This thesis is developed in pair with another thesis [42] that supports the microkernel architecture in the host system side (*c.f* section 3.2). Both the thesis together define the new interaction model for hardware-software communication.

As HAL-ASOS version four [63, 62], the microkernel establishes three transfer channels based on the AMBA AXI protocol, developed by ARM. The two channels, represented in green (🟩) in fig. 4.1.1, differentiate the slave interfaces into a control-oriented S00 interface, connected to hardware resources, and a data-oriented S01 interface connected to the local memory (LRAM), with the microkernel acting as an AXI4 lite bus slave in both scenes. In contrast, the M00 channel, represented in red (🟥) in fig. 4.1.1, stipulates interactions with the system memory (SYSRAM) with the microkernel acting as an AXI4 bus master. With the newly added interaction scheme between the microkernel and the host operating system (*c.f* section 5.1), this master interface now implements both control- and data-oriented communication through the system memory, opposed to the data-only SYSRAM mechanism employed by HAL-ASOS [63,

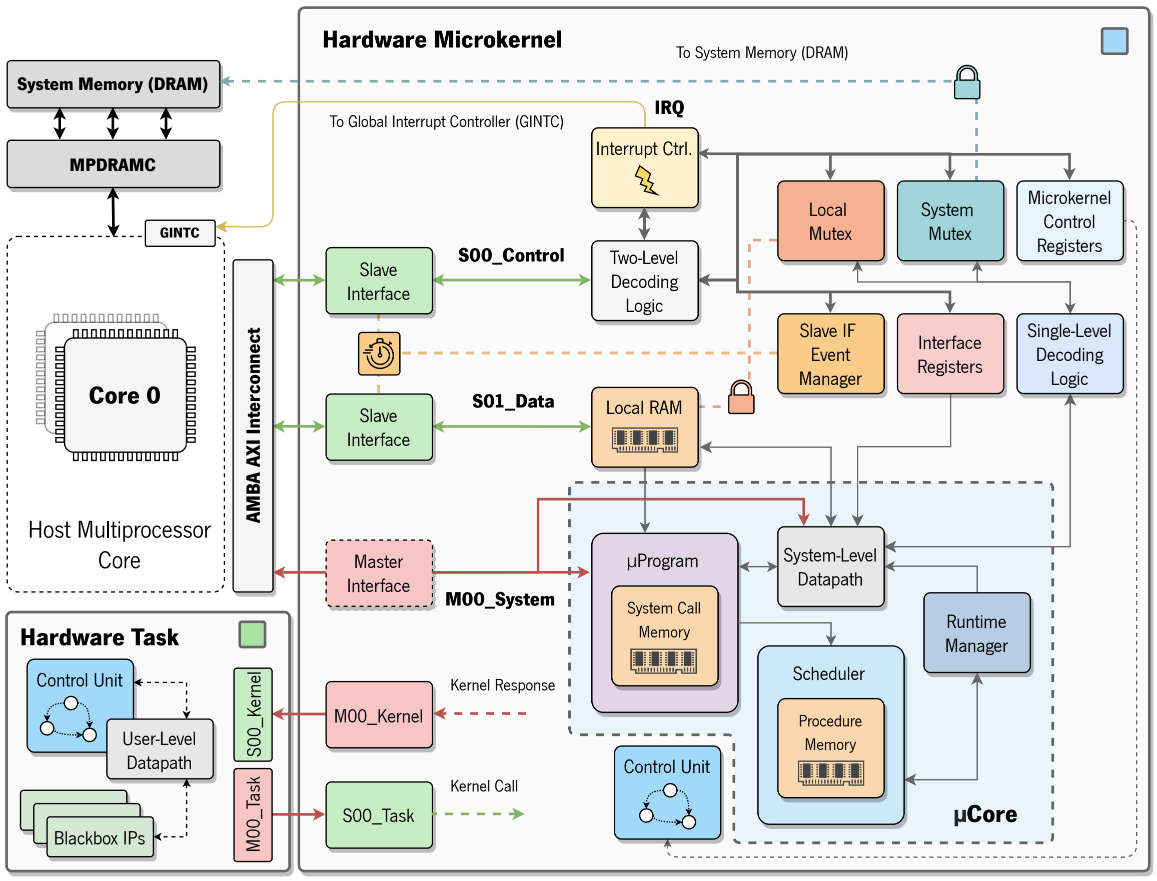


Figure 4.1.1: Microkernel Architecture Overview within Host Platform (Simplified), expanded view in fig. B.10.

62]. Additionally, when considering the data-oriented communication of HAL-ASOS through the M00 System interface, the microkernel additions now make it possible to perform burst data transfers by migrating the interface from AXI4 lite to AXI4. To improve clarity throughout the dissertation the host system's main memory will be denominated as SYSRAM, since the latter corresponds to the available portion of main memory allocated for the accelerator and used as shared memory region.

The microkernel core, fig. 4.1.1, consists of the microprogram, represented in purple (🟪), the microprogrammed procedure scheduler, represented in bright blue (🟦), the system-level datapath, represented in gray (⬜), and the runtime manager entity, represented in a darker blue (🟦). The microprogram entity, as in [63, 62], is the microprogrammed control unit responsible for executing the hardware system calls (refer to section 4.2.3). The latter takes some test input signals and acts regarding those signals to produce the specified control signals. For more information about this unit, please refer to section 2.3.1.2 for a generic overview and section 4.2.4 for a specific overview. The system-level datapath supports the microprogram to execute the aforementioned hardware system calls, but its job is to enable certain datapath logic elements, e.g., a flip-flop, or entities, e.g., the event manager, the index manager, or the address manager, section 4.2.5. The job of the procedure scheduler, section 5.1.1, is to decide the next system call for execution when using composite system calls, i.e., user-level (or hardware task) system calls that demand more than one kernel-level system call. As an example, the user-level system call that performs a safe write of data to the SYSRAM would demand the use of the kernel-level system call for locking the mutex that refers to the system memory, represented in teal (🟦), the system call for actually writing to the SYSRAM, and, ultimately, the system call for unlocking the same SYSRAM mutex. The system call is deemed as safe since the use of the mutex prevents race conditions. The procedure scheduler developed in the dissertation adds onto the incremental one used in [63, 62] since it has branch capabilities. By making the scheduler unit microprogrammable, one can achieve a flow of system calls dictated by the unit's test input signals, and, therefore, increase the procedure's adaptability to certain scenarios, e.g., a user-level procedure that acts upon a FIFO can now decide to carry through a different sequence of system calls if the FIFO is full or not full. One additional advantage of having a microprogrammed scheduler, as the microprogram, is the ability to map to both memories onto one of the microkernel's slave interfaces, so the host system side can change the sequence of system calls within a user-level procedure or the steps of a certain kernel-level system call, respectively. The runtime manager is a module closely related with the reduction of logic used by the user-level procedures in the hardware task. Thus, the latter improves the scalability of the procedures by re-using cases, more on that later in section 5.2.

As forementioned, the microkernel makes use of a local memory (LRAM), represented in orange (🟧) in fig. 4.1.1, connected to the S01 slave interface (🟩) and locked by the local mutex, represented in darker orange (🟧). As additional resources, the microkernel also has a local interrupt controller, represented in yellow (🟨) to generate the PL-to-PS interrupts, the slave interface event manager, represented in darker yellow (🟨), that manages timeouts and event waits of both of the slave interfaces (S00 and S01), and an interface register area, represented in a light-colored red (🟥), for the host system to specify parameters regarding the system memory, e.g., the SYSRAM base address, or read information concerning the local memory, e.g., LRAM depth in words.

The next sections will detail on the design, implementation, and testing of all the modules of the microkernel, with an analysis of the trade-offs and design considerations. Note that throughout the dissertation, to increase understandability, the module's channels pertaining to the host (software side) will use blue tones (🟦), and the module's channels pertaining to the hardware side will use orange tones (🟧), which is the case for the aforementioned mutexes. However, this only applies to the external interfaces and the dual port modules.

4.2 Microkernel Core

As mentioned before, the microkernel core comprises the microprogram, the procedure scheduler, the system-level datapath, and the runtime manager. This section will mainly explore the microprogram's and system-level datapath's internal architectures and their interactions. For this, one will also talk about the implemented microcoded system calls stored in the microprogram memory as well as the HDL kernel-level procedures that abstract these system calls. The implementation and discussion of the microprogrammed procedure scheduler and the runtime manager will be left to section 5.1.1 and section 5.2.1, as they belong to the chapter about the proposed architecture extensions.

4.2.1 Control Registers

The control registers module is important since it establishes an interface for the host system to control the hardware microkernel. Throughout the dissertation the hardware microkernel might be referred to as kernel to improve clarity. The design of this module is represented in fig. 4.2.1.

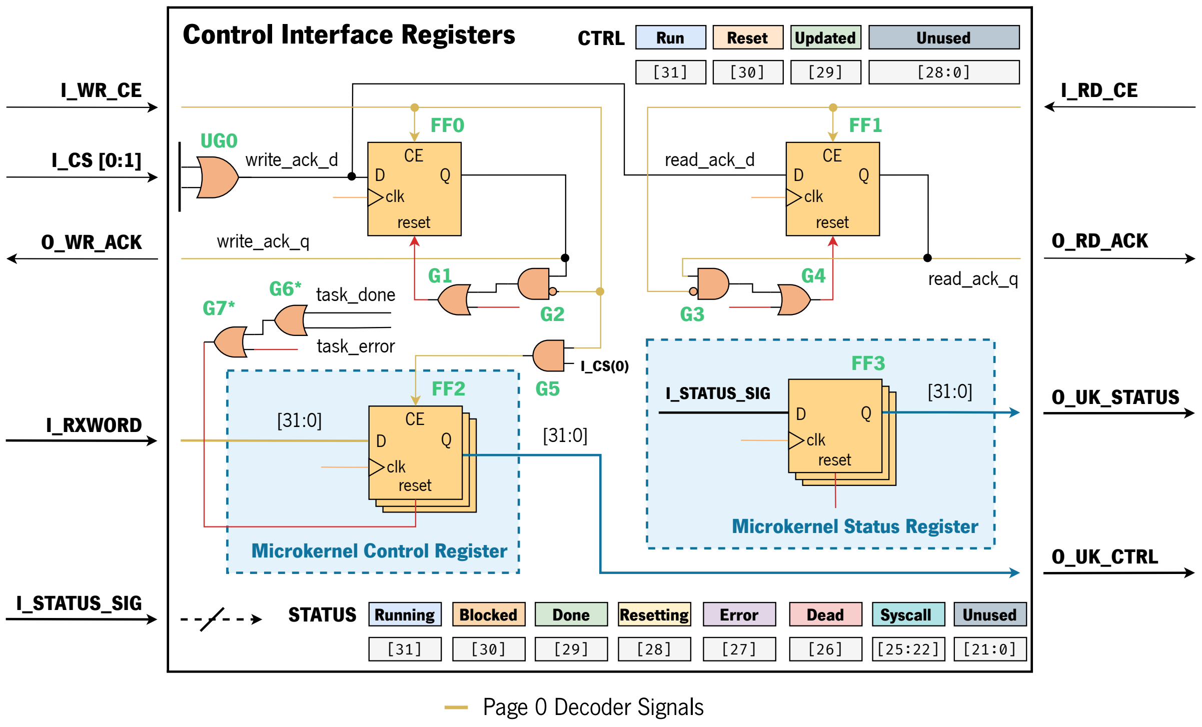


Figure 4.2.1: Microkernel Control Registers RTL Design Internal Architecture (Simplified).

As it is possible to see as well in other modules that use the AXI4 lite interface S00 Control, the control registers entity of fig. 4.1.1 has two flip-flops, FF0 and FF1, to generate the write and read slave acknowledges, respectively. This means that once the chip select, *i_cs*, and the write clock enable, *i_wr_ce*, or the read clock enable, *i_rd_ce*, are asserted one clock cycle later on will have either the *i_wr_ack* or *o_rd_ack* ports asserted as well. This methodology follows a generic bus interface, used as well in [62]. For the latter to work, the AXI4 lite interface module must be compliant with this type of bus and its control unit must react to slave confirmation signals alongside with AXI4 lite ones. Refer to section 4.6.1 for an in-depth explanation of the interface and control unit.

Additionally, the latter also has two other flip-flops, FF2 and FF3, that control the control unit of the microkernel and stand for its current status, respectively. Both of these registers, control and status, are 32-bit wide, but only the control register is writable and readable, whilst the status register is read-only, depicted by the connection to the input port, *i_rxword*, coming from the slave decoder logic element, more on that later in section 4.4.1. For the control only two bits can be asserted/cleared by the host. The MSB (bit 31) puts the microkernel in a running state, while the bit 30 resets the microkernel. Since the microkernel is deeply connected to the hardware task unit, the once one puts the kernel onto a running or reset state the hardware task follows, and enters such states as well. The status register has bits that give information on the current state of the kernel and hardware task. The running bit to indicate the kernel and hardware task are processing something, the blocked bit to indicate the hardware task is blocked and waits a confirmation from the microkernel to continue processing, the done bit to indicate the hardware task finished executing, the resetting bit to point to a restart of the hardware task, the error bit to report an error while executing the hardware task system calls, a dead bit that represents a shutdown accelerator, via a specific yield syscall, and finally a field that specifies the currently active microprogram system call. In fig. 4.2.1 there is also a mention of signals coming from the page zero decoder, because this entity is mapped to page zero of the accelerator. More on the S00 Control address space in section 4.6.3, and the page decoder in section 4.4.2. The HDL for the control and status registers, and the combinational update of the status is represented in listing 4.1.

Listing 4.1: Microkernel Control: Control and Status Registers (HDL).

```vhdl
-----------------------------------------------------------------------------------
KERNEL_CONTROL_FF2 : process (I_CLK)
-----------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if(I_RESET = '1') then
            ukernel_ctrl_q <= ukernel_vector_to_control_data((others => '0'));
        elsif I_WR_CE = '1' and I_CS(0) = '1' then
            ukernel_ctrl_q <= ukernel_vector_to_control_data(I_RXWORD);
        elsif I_TASK_DONE = '1' or I_TASK_ERROR = '1' then
            ukernel_ctrl_q <= ukernel_vector_to_control_data(('0' & I_RXWORD(
                C_MACHINE_WIDTH-2 downto 0)));
        end if;
    end if;
end process KERNEL_CONTROL_FF2;
------------------------------------------------------------------------------------------
O_UKERNEL_CTRL <= ukernel_control_data_to_vector(ukernel_ctrl_q);

ukernel_status_d.unused <= (others => '0');
------------------------------------------------------------------------------------------
UKERNEL_STATUS_D_UPDT_0 : process(ukernel_ctrl_q,I_BLOCK_TASK,
I_TASK_DONE,I_TASK_DEAD,I_TASK_ERROR,I_SYSCALL_ID,I_TASK_RUN,I_TASK_RESET)
------------------------------------------------------------------------------------------
begin
    ukernel_status_d.running     <= I_TASK_RUN;
    ukernel_status_d.hwt_blocked <= I_BLOCK_TASK;
    ukernel_status_d.done        <= I_TASK_DONE;
    ukernel_status_d.resetting   <= I_TASK_RESET;
    ukernel_status_d.error       <= I_TASK_ERROR;
    ukernel_status_d.dead        <= I_TASK_DEAD;
    ukernel_status_d.syscall_id  <= I_SYSCALL_ID;
    ---
end process UKERNEL_STATUS_D_UPDT_0;
------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------
KERNEL_STATUS_FF3 : process (I_CLK)
------------------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if(I_RESET = '1') then
            ukernel_status_q <= ukernel_vector_to_status_data((others => '0'));
        elsif I_WR_CE = '1' and I_CS(0) = '1' then
            ukernel_status_q <= ukernel_status_d;
        end if;
    end if;
end process KERNEL_STATUS_FF3;
------------------------------------------------------------------------------------------
O_UKERNEL_STATUS <= ukernel_status_data_to_vector(ukernel_status_q);
```

4.2.2 Control Unit

The microkernel control unit intends to sync the host application with the microkernel and hardware task entities. For this the latter needs to interact with both hardware entities in a way that leverages correct functioning and synchrony. The overall interaction scheme between the control unit, the hardware task, and of other units of the kernel is represented in fig. 4.2.2.

As it is possible to see the control unit receives as inputs the internal signals *run_i* and *reset_i* from the control register, section 4.2.1, the *task_done_i* signal from the hardware task when it finishes execution, and signals from the microprogram, section 4.2.4, to kill the accelerator when executing a yield syscall, *kill_acc_i*, and to indicate runtime errors that occur when executing system calls, *fault_i*. A signal to indicate a dead state of the accelerator is then used to inform the microprogram entity, and the signals referent to run and reset, *task_run* and *task_reset*, respectively, are used to synchronize the hardware task with the kernel, via the kernel response message. In a similar way the *task_done_i* signals comes within the kernel call message.

The kernel FSM starts off in state #0, ready, and moves onto the state #1, running, if the run bit (31) of

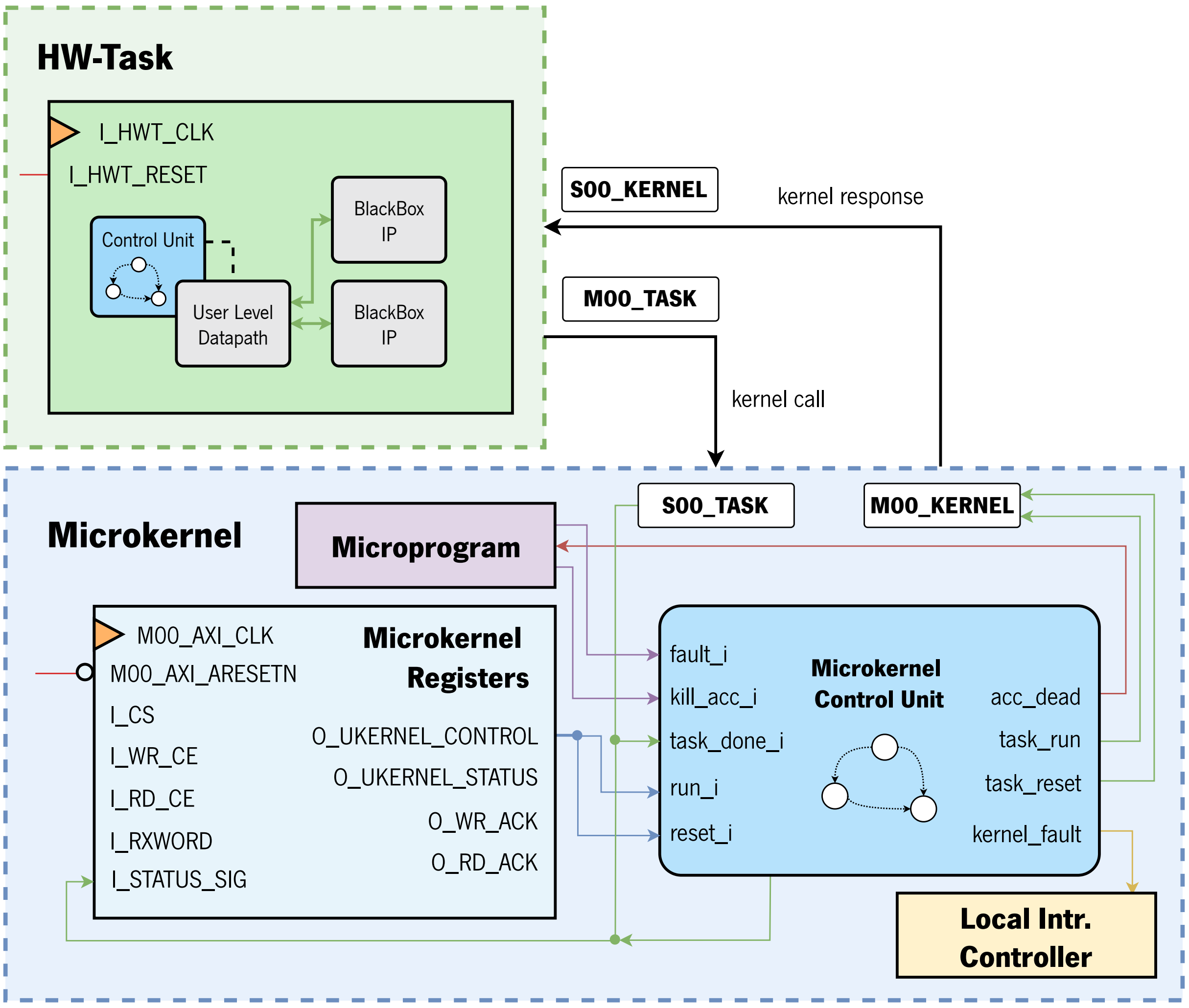


Figure 4.2.2: Microkernel Control Unit Interaction Overview.

the control register, fig. 4.2.1, is asserted. If that occurs, the FSM moves to #1 in the next clock cycle with the assignment of the *next_state* signal to the *state* signal. While in running state (#1), the kernel asserts the *task_run* signal and extends it to the hardware task entity through the kernel response message, as it is possible to see in fig. 4.2.3. In the running state the kernel can either go to #4, the dead state, or go to #2, the error state. If the signal *kill_acc* is asserted by the microprogram, fig. 4.2.3, the FSM reaches #4 in the next clock cycle and the signal *acc_dead* is asserted to conclude the work yield system call, more on that in section 4.2.3. Otherwise, if the microprogram asserts the *fault* signal instead, the FSM reaches #2 and asserts the *task_error* signal. In the case that none of these signals are asserted, the FSM keeps itself in the running state (#1) if the run register is not cleared. Moreover, from #1, the next state can also be #0, ready, if the hardware task has finished executing. When this happens the *task_done* signal is also used to clear the run register to prevent unwanted multiple-iteration execution, as seen in fig. 4.2.3 (G6*).

In either the #1 state, running, or the #0 state, idle, the FSM's next state can be the restart one, #3. This occurs if the reset register of the control module, section 4.2.1, is asserted by the host application. Upon assertion, the next state will be #3, restart, in the next clock cycle, and the *task_reset* signal will be extended to the hardware task through the kernel response message, as for the *task_run* case. This can be observed in fig. 4.2.3.

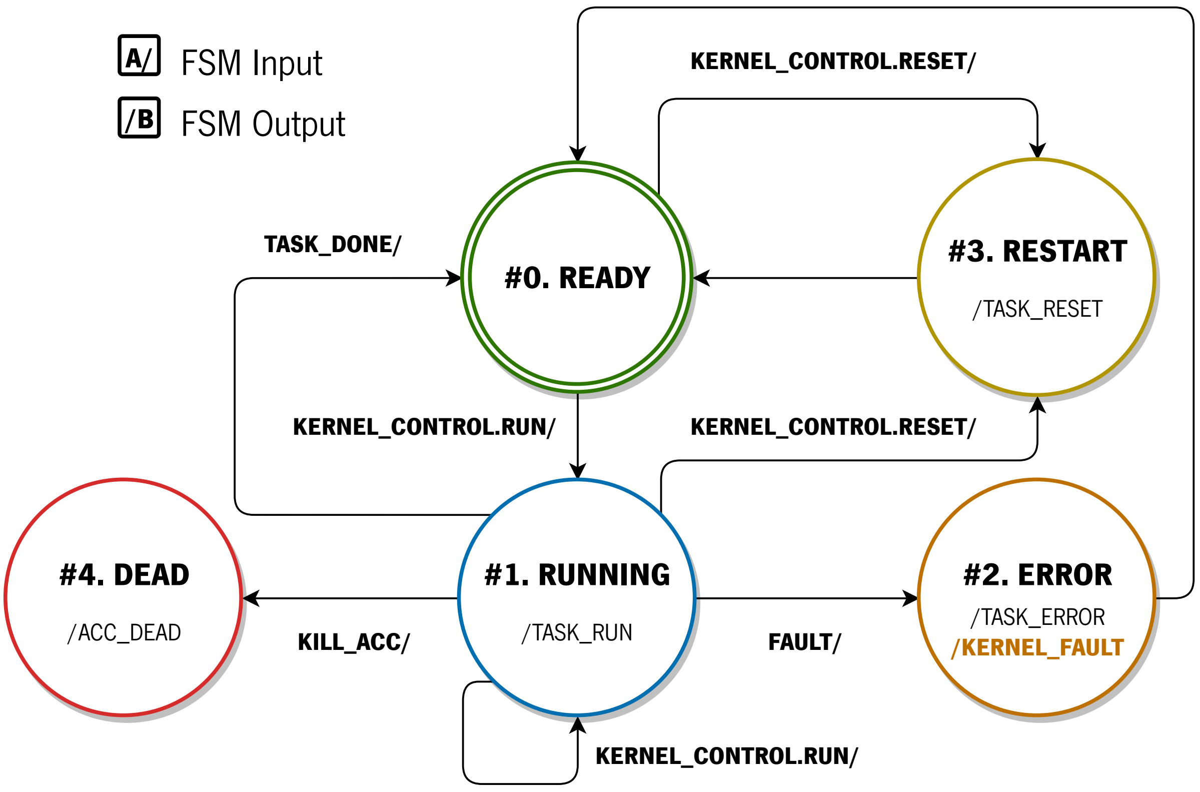


Figure 4.2.3: Microkernel Control Finite State Machine (FSM), adapted [62].

4.2.3 Microcoded System Calls

The microcoded system calls are closely related with the microprogram, section 4.2.4, and the system-level datapath, section 4.2.5, as one stores the system calls, specifies the sequence of steps to execute, and what control signals are active based on test inputs, and the other activates datapath resources associated with the system calls, respectively. These system calls can be seen as a simple sequence of steps to perform a particular function, e.g., lock or unlock a mutex. The microcoded system calls, as stated by [62], reduce the complexity of using kernel resources and are abstracted by kernel-level HDL procedures. HAL-ASOS [62] uses system calls with a maximum of four steps, and the same approach was used in the dissertation, as depicted by table 4.1. The summary describes the functionality of each of the system calls and the color identifier used for them to improve clarity. The system calls might be referred to as syscalls for better readability. Additionally, microprogram might refer to the entity responsible for storing the system calls and sequencing them, section 4.2.4, or the program composed by microinstructions or steps that makes a system call.

Table 4.1: Microkernel: Low-Level Hardware System Call Summary.

System Call	ID	Description	Steps	Identifier
SYSCALL_WORK_NONE	0	No operation	1/4*	⬜
SYSCALL_WORK_YIELD	1	Kill the accelerator	2/4*	🟥
SYSCALL_WAIT_EVENT_TIMEOUT	2	Wait a certain amout of clock cycles for an event or until timeout	3/4*	🟧
SYSCALL_LINTC_READ	3	Read from Local Interrput Controller (Control and Status Reg.)	1/4*	🟨
SYSCALL_LINTC_WRITE	4	Write to Local Interrupt Controller (Generate IRQ)	2/4*	🟨
SYSCALL_LBUS_READ	5	Read one word from the LRAM, requires LRAM offset	2/4*	🟦
SYSCALL_LBUS_WRITE	6	Write one word to the LRAM, requires data and LRAM offset	2/4*	🟧
SYSCALL_LBUS_READ_BURST	7	Read multiple words from the LRAM (section 4.5.3), in burst mode	3/4*	🟦
SYSCALL_LBUS_WRITE_BURST	8	Write multiple words to the LRAM (section 4.5.3), in burst mode	3/4*	🟧
SYSCALL_MUTEX_LOCK	9	Lock the hardware mutex (section 4.5.1) with an owner ID	4/4	🟦
SYSCALL_MUTEX_TRY_LOCK	10	Try locking the hardware mutex (section 4.5.1) with an owner ID	4/4	🟦
SYSCALL_MUTEX_UNLOCK	11	Unlock the hardware mutex (section 4.5.1) with an owner ID	4/4	🟦
SYSCALL_MBUS_READ	12	Read one word from the SYSRAM, requires SYSRAM offset	3/4*	🟪
SYSCALL_MBUS_WRITE	13	Write one word to the SYSRAM, requires data and SYSRAM offset	4/4	🟥
SYSCALL_MBUS_READ_BURST	14	Read multiple words from the SYSRAM, in burst mode	3/4*	🟪
SYSCALL_MBUS_WRITE_BURST	15	Write multiple words to the SYSRAM, in burst mode	4/4	🟥

Note: The syscalls marked with an asterisk use four steps but employ safeguard corrections in the remaining steps, like [62].

The first syscall present in table 4.1 is *syscall_work_none*. As the name indicates, this syscall does not do anything, and it is often used to keep the microprogram in a no operation state similar to a NOP instruction in some CPU architectures, e.g., Intel's x86 or RISC-V. The pseudocode for the behavior of *syscall_work_none* is represented in algorithm 1. As it is possible to see, the syscall continuously tests the auxiliary false input on step zero, originating a jump to the same step (zero) on a false condition. Since the input selected to test is always false, this syscall remains in step zero forever, only signalling the hardware task to advance with the valid signal. The behavior of this microinstruction is similar to what a jump $ would generate, for example, in 8051 assembly. Additionally, the next three steps employ safeguard corrections, as [62], making the microprogram jump to step zero if it reaches any of these other steps. Each safeguard step (one, two, and three) activates the block task signal to stop the execution of the hardware task until the flow of microinstructions returns to the initial step. This is possible because each one of the aforementioned steps also always tests the auxiliary false input, going back to step zero in the next clock cycle. For more specifications on this mechanism, refer to section 4.2.4.

Algorithm 1 Microprogram to perform the no operation system call (⬜)

```
    pseudocode SYSCALL_WORK_NONE
1:  Step 0 : test auxiliary false select null produce valid
2:      if condition false then goto step zero.
3:  Step 1 : test auxiliary false select null produce block task    ▷ Safeguard Correction
4:      if condition false then goto step zero.
5:  Step 2 : test auxiliary false select null produce block task    ▷ Safeguard Correction
6:      if condition false then goto step zero.
7:  Step 3 : test auxiliary false select null produce block task    ▷ Safeguard Correction
8:      if condition false then goto step zero.
    end pseudocode
```

The system call to kill the accelerator, represented in algorithm 2, behaves in a way that enables the *kill_accelerator* control signal in its first step while waiting for the confirmation signal *accelerator_dead*. The signal to kill the accelerator and deactivate a certain hardware task makes the microkernel FSM to go to the dead state and assert the forementioned confirmation signal, *accelerator_dead*.

The *syscall_wait_event_timeout* in algorithm 3 uses the event manager, section 4.2.5.1, in the system-level datapath, to wait on an event for a stipulated amount of clock cycles, or just wait a time-out. With this in mind, the first step of the system call stays in step zero until the event manager is ready,

Algorithm 2 Microprogram to kill the accelerator associated with a certain hw-task (🟥)

```
    pseudocode SYSCALL_WORK_YIELD
1:  Step 0 : test accelerator dead select kill accelerator produce valid
2:      if condition false then goto step zero.
3:  Step 1 : test auxiliary false select null produce valid              ▷ Exit Step
4:      if condition false then goto step zero.
5:  Step 2 : test auxiliary false select null produce block task         ▷ Safeguard Correction
6:      if condition false then goto step zero.
7:  Step 3 : test auxiliary false select null produce block task         ▷ Safeguard Correction
8:      if condition false then goto step zero.
    end pseudocode
```

advancing to step one when the latter condition is true. The syscall then stays in step one for the entirety of the specified wait, always enabling the event manager with the selection of the event manager trigger control signal, and waiting on true condition on event elapsed to move to the next step. The event elapsed signal becomes true if the specified event is asserted within the time specified, or if there is a timeout. For example, if one intended to wait on a AXI signal like *BVALID* with a timeout of sixteen clock cycles, the event elapsed would be asserted if *BVALID* had the logical value '1' within sixteen cycles or at the end of that same time, even if *BVALID* was not asserted. The step two of the syscall employs a jump on a false condition to step zero. As in [62], this favors regularity, and makes the next system call to be executed to start on step zero. This happens because the microcode memory address is given by a concatenation of the external address of the syscall, given by the syscall ID, and the two bits that specify the next step, given by the logic element that increments or loads this address part. For more information on the matter, refer to section 4.2.4. The last step of the syscall performs a safeguard correction, blocking the hardware task and going to step zero in the next clock cycle if the microprogram ever reaches this state. Additionally, this step also stores the syscall return on the return argument register, section 4.2.5.

The system call for reading from the LINTC entity is straightforward, since it is similar to the no operation syscall (*c.f* algorithm 1) in terms of the microcode stored in memory. This means that the major difference between them actually occurs in the System-Level Datapath, i.e., the no operation syscall does not perform anything in terms of datapath operation. The syscall for the LINTC read only forwards the content of the control and status registers to the hardware task.

Algorithm 3 Microprogram to perform the system call that waits for an event or timeout (🟧)

```
pseudocode SYSCALL_WAIT_EVENT_TIMEOUT
1: Step 0 : test event manager ready select null produce block task
2:     if condition false then goto step zero.
3: Step 1 : test event elapsed select event manager trigger produce block task
4:     if condition false then goto step one.
5: Step 2 : test auxiliary false select null produce valid                ▷ Exit Step
6:     if condition false then goto step zero.
7: Step 3 : test auxiliary false select null produce block task   ▷ Safeguard Correction
8:     if condition false then goto step zero.
end pseudocode
```

The syscall for writing to the LINTC entity is a bit different since it generates an interrupt request from the PL to the PS given a certain interrupt source as parameter. Thus, it stays on step zero, until the interrupt is raised, described in algorithm 4 by the test input signal selection. On the next clock cycle, the microprogram exits with the aforementioned dummy jump to step zero. For more information about the generation of the *intr_raise* signal refer to section 4.5.2.

Algorithm 4 Microprogram to perform the LINTC write system call (🟨)

```
pseudocode SYSCALL_LINTC_WRITE
1: Step 0 : test interrupt raise select null produce block task
2:     if condition false then goto step zero.
3: Step 1 : test auxiliary false select null produce valid                ▷ Exit Step
4:     if condition false then goto step zero.
5: Step 2 : test auxiliary false select null produce block task   ▷ Safeguard Correction
6:     if condition false then goto step zero.
7: Step 3 : test auxiliary false select null produce block task   ▷ Safeguard Correction
8:     if condition false then goto step zero.
end pseudocode
```

The syscall to read one word from the local RAM, *syscall_lbus_read* in algorithm 5, is very simple, with only one step referent to the read, the exit step already described earlier in this section (paragraph two), and the remaining two steps for safeguard corrections. For the most part, this syscall is almost assured by the system-level datapath, since the latter specifies the LRAM address and also activates the *CS_B* signal of the local memory. The only thing the syscall has to do is wait the lbus_rd_ack coming one clock cycle after both *CS_B* and *RD_CE* signals are active, if the memory in question has a *RD_CE* port. In this case, the local memory does not have a *RD_CE* port and, as so, the system call has no need to select the *lbus_rd_ce*. Thus, the memory generates the read acknowledge with the logical *NOT* of the *WR_CE* signal paired with a logical *AND* of the *CS* signal for both A and B ports. More on that in section 4.5.3. Additionally, the local memory read is performed without any enabling signal, i.e., opposed to the write where the *WR_CE* signal needs to be asserted to perform the write, in the read the contents of the specified address appear one clock cycle later in the LRAM's *DOUT* port.

Algorithm 5 Microprogram to perform the system call that reads one word from the LRAM (🟦)

```
    pseudocode SYSCALL_LBUS_READ
1:  Step 0 : test local bus read ack select null produce block task
2:          if condition false then goto step zero.
3:  Step 1 : test auxiliary false select null produce valid                ▷ Exit Step
4:          if condition false then goto step zero.
5:  Step 2 : test auxiliary false select null produce block task   ▷ Safeguard Correction
6:          if condition false then goto step zero.
7:  Step 3 : test auxiliary false select null produce block task   ▷ Safeguard Correction
8:          if condition false then goto step zero.
    end pseudocode
```

The *lbus* and *mutex* system calls refer to the local bus, as HAL-ASOS [62] uses this nomenclature. In the latter [62], the local bus is a decoding unit similar to the ones in section 4.4, that allows the microprogram to write words not only to the local memory but also to some hardware kernel resources, e.g., the local interrupt controller, and to the zero copy unit of HAL-ASOS. Since the dissertation develops a microkernel, there are not enough units to justify the addition of a local bus (decoder), even thought this was possible. The naming of local bus was kept to encompass future expansions of the microkernel if the need for an in-fact local bus arises. The test signals in the syscalls that refer to the local bus use signals that refer to the local bus as well, e.g., *lbus_rd_ack* or *lbus_wr_ce*, but for the aforementioned reasons the microkernel's local bus signals are only connected with the local RAM and the two mutex resources. This means that is only possible to read or write to the LRAM using the *syscall_lbus_read* or *syscall_lbus_write* syscalls, respectively, or using the equivalent syscalls in that use the burst format, namely *syscall_lbus_read_burst* or *syscall_lbus_write_burst*.

Observing the syscall to write one word to the local RAM, one notices that it is very akin to the system call for a single word read. Similarly, the *sycall_lbus_write* has to wait on the lbus_wr_ack, which in this case is the LRAM write acknowledge signal, but while selecting the *lbus_wr_ce* signal. The syscall needs to assert the *lbus_wr_ce* since the write is dependent on that signal assertion. In the same way, the *lbus_wr_ack* coming from the local memory should arrive after one clock cycle if the write was successful, i.e., if the local RAM is not locked by the local mutex with an owner from the system channel (A). For more information on the mutex resource refer to section 4.5.1.

Algorithm 6 Microprogram to perform the system call that writes one word to the LRAM (🟧)

```
   pseudocode SYSCALL_LBUS_WRITE
1: Step 0 : test local bus write ack select local bus write clock enable produce block task
2:     if condition false then goto step zero.
3: Step 1 : test auxiliary false select null produce valid                    ▷ Exit Step
4:     if condition false then goto step zero.
5: Step 2 : test auxiliary false select null produce block task     ▷ Safeguard Correction
6:     if condition false then goto step zero.
7: Step 3 : test auxiliary false select null produce block task     ▷ Safeguard Correction
8:     if condition false then goto step zero.
   end pseudocode
```

The microprogram to read multiple words from the local memory in burst mode, depicted by the *syscall_lbus_read_burst* in algorithm 7, needs the help of the address manager element in the system-level datapath to perform the increment of the address. For example, if one intends to do a read burst of five elements from the word offset four, the address needs to be incremented from four to eight. For this reason, the *syscall_lbus_read_burst* waits the address manager to be ready while selecting the control signal that triggers the address load and increment, before moving on to step one. In this step the syscall tests the local bus read acknowledge coming from the local RAM like in *syscall_lbus_read*, but this time also waits the *read_last* signal, to indicate the last element to read. This strategy was developed to mimic what happens in the AXI4 protocol with the *RLAST* signal. Thus, the microprogram will receive a read acknowledge for each element read from the LRAM but only moves to the exit step with the logic *AND* between the *lbus_rd_ack* of the last element and the *read_last* signal. As stated before, the unused step, in this case step three, employs a safeguard correction. For an in-depth description of the address manager element, refer to section 4.2.5.3.

Algorithm 7 Microprogram that burst reads multiple words from the LRAM (🟦)

```
   pseudocode SYSCALL_LBUS_READ_BURST
1: Step 0 : test address manager ready select address manager trigger produce block task
2:     if condition false then goto step zero.
3: Step 1 : test local bus read ack and read last select null produce block task
4:     if condition false then goto step one.
5: Step 2 : test auxiliary false select null produce valid                      ▷ Exit Step
6:     if condition false then goto step zero.
7: Step 3 : test auxiliary false select null produce block task       ▷ Safeguard Correction
8:     if condition false then goto step zero.
   end pseudocode
```

Once again, the syscall for writing multiple words to the local memory (LRAM) in burst mode, *syscall_lbus_write_burst* in algorithm 8, is very similar to the read syscall, *syscall_lbus_read_burst*, in terms of the microprogram steps. This syscall also relies on the address manager, section 4.2.5.3, to load and increment the address of the LRAM in each write, waiting for it to be ready and selecting the address manager trigger as active control signal. The next step, step one, focuses on performing the successive writes until the write_last signal is asserted and while selecting the local memory clock enable, since the write cannot be done without it being asserted. The only big difference between these two system calls is the fact that the write one also relies on another entity, the index manager (section 4.2.5.2), to increment the index of the hardware task buffers that hold the data to be written at a given address in the local memory. For example, if one wants to perform a burst write of six thirty-two bit words, e.g., 0×01 up to 0×06, from the LRAM address offset zero to word offset five, then one needs the address manager to specify the LRAM address per clock cycle, and also the index manager to specify the data word per clock cycle. This makes it possible to write the word 0×01 to address offset zero in the first clock cycle, and write 0×02 to address offset one in the next clock. The mention of address offset is due to the fact that these burst local bus system calls allow the specification of the initial address one is writing to. This means that the syscall might write from word offset zero to five if the address offset zero corresponds to the start of the RAM or, for example, to the RAM address four up until address nine, considering the same burst of six words, if the address offset zero corresponds to the word offset four. Note that the *syscall_lbus_read_burst* also makes use of the index manager, to increment the index of the output buffer in the hardware task, and also to know when to generate the *read_last* signal. For more information on the index manager unit, refer to section 4.2.5.2.

Algorithm 8 Microprogram that burst writes multiple words to the LRAM (🟧)

```
    pseudocode SYSCALL_LBUS_WRITE_BURST
1:  Step 0 : test address manager ready select address manager trigger produce block task
2:      if condition false then goto step zero.
3:  Step 1 : test lbus write ack and write last select lbus write clock enable produce block task
4:      if condition false then goto step one.
5:  Step 2 : test auxiliary false select null produce valid                ▷ Exit Step
6:      if condition false then goto step zero.
7:  Step 3 : test auxiliary false select null produce block task           ▷ Safeguard Correction
8:      if condition false then goto step zero.
    end pseudocode
```

The system call used to lock a hardware mutex, *syscall_mutex_lock* in algorithm 9, is used to lock both the local mutex and the system mutex of fig. 4.1.1, which are related with avoiding race conditions in both the LRAM and SYSRAM, respectively. This system call starts by analyzing the status of the mutex locked A flag to comprehend if the mutex is locked by the A channel, i.e., the host system channel (step zero). If it is indeed locked by the latter, given by a false condition on the logical *NOT* of the locked A flag this blocking syscall remains in step zero always checking the locked A status until the mutex is released by the channel A.

When the mutex is released or if it was not locked by the host in the first place, the syscall then proceeds to step one. In this step the microprogram performs a true dummy test that goes to step two regardless of the result, and selects the *lbus_wr_ce* control signal so, one cycle later, the write acknowledge is generated alongside the writing of the channel B owner ID onto the hardware mutex data register. As stated, even if the auxiliary true test results false by some glitch, the microprogram still enforces the next step of the system call to be step two. In step two, a check on the locked B flag is performed to check if the hardware mutex was properly locked by the microprogram channel, i.e., channel B. Following the same methodology

Algorithm 9 Microprogram to perform the system call that locks a hardware mutex (🟦)

```
pseudocode SYSCALL_MUTEX_LOCK
1: Step 0 : test not locked A flag select local bus read clock enable produce block task
2:         if condition false then goto step zero.
3: Step 1 : test auxiliary true select local bus write clock enable produce block task
4:         if condition false then goto step two.
5: Step 2 : test locked B flag select local bus read clock enable produce block task
6:         if condition false then goto step zero.
7: Step 3 : test auxiliary false select null produce valid                  ▷ Exit Step
8:         if condition false then goto step zero.
end pseudocode
```

as the write, the *lbus_read_ce* control signal is always selected when performing a read on the mutex, in order to generate the read acknowledge. The same applies for the step zero of this microprogram, when reading the locked A flag. Ultimately, the step three ensures that the next system call address starts on zero, by performing the exit step already explained earlier in this section. For the specifics on the inner workings of the mutex hardware resource, consult section 4.5.1.

The system call to perform the lock of a mutex in a non-blocking way, *syscall_mutex_try_lock*, is described in algorithm 10. This syscall repeats the steps zero and one, as *syscall_mutex_lock*, and differs in the implementation of the third step, step two, as it employs a jump to step three, the exit step, instead of step zero. This way, if the mutex lock was not successful the syscall exits, opposed to *syscall_mutex_lock*, where the syscall would return to step zero to perform the lock again.

The system call to perform the unlock of a hardware mutex, syscall_mutex_unlock, is represented in algorithm 11. The syscall starts by verifying if the mutex resource is locked by the system channel (A), given by the locked A flag in step zero. The tested signal is the logical *NOT* of the locked A flag, this means that the microprogram goes to step three, i.e., exits, if the mutex is owned by the system channel (A), i.e., the *NOT* on the flag is false, or proceeds to step one if the *NOT* on the locked A flag is true, indicating that the mutex is not owned by A. Despite that, in step one, is still necessary to check if the mutex is locked by the channel B before releasing it. This is important because a release of a mutex that is not locked by B could originate an uncalled-for lock. This can happen because the release of the mutex resource also relies on the write of the same owner ID to the data register as the mutex lock does. For more information

Algorithm 10 Microprogram to perform the system call that tries to lock a hardware mutex (🟦)

```
    pseudocode SYSCALL_MUTEX_TRY_LOCK
1:  Step 0 : test not locked A flag select local bus read clock enable produce block task
2:      if condition false then goto step zero.
3:  Step 1 : test auxiliary true select local bus write clock enable produce block task
4:      if condition false then goto step two.
5:  Step 2 : test locked B flag select local bus read clock enable produce block task
6:      if condition false then goto step three.
7:  Step 3 : test auxiliary false select null produce valid                    ▷ Exit Step
8:      if condition false then goto step zero.
    end pseudocode
```

on the mutex refer to section 4.5.1. To achieve this the step one of the microprogram tests the locked B flag, while activating the *lbus_rd_ce* control signal to generate the mutex read acknowledge after one clock cycle, as mentioned already throughout this section, section 4.2.3. The next step in the flow of microinstructions is either the exit step, if the mutex is not locked by B, or the step two, if the mutex is indeed locked by the hardware channel (B). In step two, the microprogram perform a safety check on the mutex free flag to evaluate if the mutex was properly released.

Algorithm 11 Microprogram to perform the system call that unlocks a hardware mutex (🟦)

```
    pseudocode SYSCALL_MUTEX_UNLOCK
1:  Step 0 : test not locked A flag select local bus read clock enable produce block task
2:      if condition false then goto step three.
3:  Step 1 : test locked B flag select local bus read clock enable produce block task
4:      if condition false then goto step three.
5:  Step 2 : test free flag select local bus write clock enable produce block task
6:      if condition false then goto step zero.
7:  Step 3 : test auxiliary false select null produce valid                    ▷ Exit Step
8:      if condition false then goto step zero.
    end pseudocode
```

This particular step needs to activate the *lbus_wr_ce* control signal to write the channel B owner ID onto the mutex data register, and consequently perform the intended release of the resource.

The system calls to write one or multiple words to the system memory, *syscall_mbus_read(_burst)*, are described by algorithm 12. For the system calls referent to the master bus, the microprogram behaves as an AXI4 bus master. This means that it must comply with the AXI4 protocol, testing AXI4 signals while it activates AXI4 control signals in response. The decision to use a AXI4 interface even in *syscall_mbus_read*, where a AXI4 lite master interface would suffice to perform single word reads, was made to favor master bus regularity. The AXI4 protocol is similar to the AXI4 lite protocol in terms of signals, but the latter does not allow the burst mode, and, therefore, does not account for burst-related signals. For an in-depth description of the AXI4 protocol without the burst mode (AXI4 lite), refer to section 4.6.1.1. For an overall view on the AXI4 protocol, refer to fig. 4.2.4 and fig. 4.2.5.

Algorithm 12 Microprogram that reads one or multiple words from the SYSRAM (🟪)

```
    pseudocode SYSCALL_MBUS_READ(_BURST)
1:  Step 0 : test AXI read address ready select AXI read address valid produce block task
2:          if condition false then goto step zero.
3:  Step 1 : test AXI read valid and bus read last select AXI read ready produce block task
4:          if condition false then goto step one.
5:  Step 2 : test auxiliary false select null produce valid                    ▷ Exit Step
6:          if condition false then goto step zero.
7:  Step 3 : test auxiliary false select null produce block task     ▷ Safeguard Correction
8:          if condition false then goto step zero.
    end pseudocode
```

As described by fig. 4.2.4 and fig. 4.2.5, the AXI4 bus master, in this case the microprogram, defines the read address from the system memory and activates the *ARVALID* signal to identify the start of a read transfer. At the same time, the master waits for the address confirmation from the slave, given by the *ARREADY* signal. This is described in step zero of algorithm 12, where the microinstruction activates the *m_axi_arvalid* control signal, and tests the *m_axi_arready* signal. The microprogram remains in step zero until the ready signal from the slave, i.e., *ARREADY*, is asserted. If one continues to analyze fig. 4.2.4 and fig. 4.2.5, it is possible to see that, following the assertion of the *ARREADY* signal, the master asserts the *RREADY* signal to indicate that it is ready to receive the specified data. This translates to the selection of the *m_axi_rready* control signal in step one of algorithm 12. The slave then asserts the *RVALID* signal while the data to transmit is valid, and finally asserts the *RLAST* signal to indicate to the master which element is the last to be read. This can be viewed in algorithm 12 with the testing of the logical *AND* of *m_axi_rvalid* and *m_axi_rlast*. The microprogram only goes to the exit step, step two, when the data is valid concerning the last element to be transferred, staying in step one otherwise.

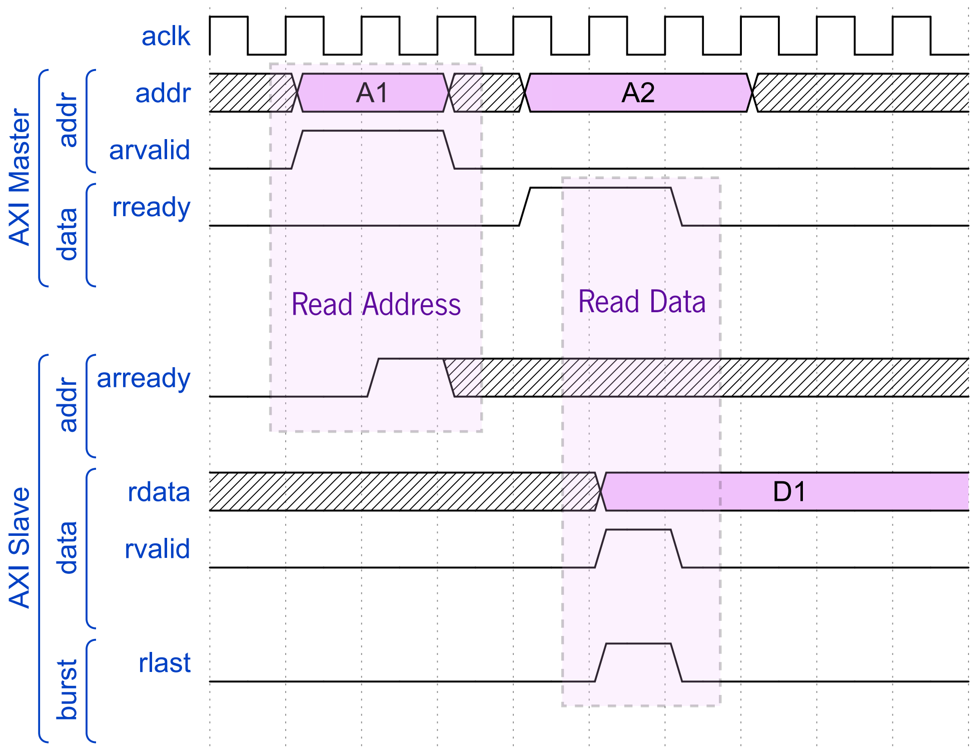


Figure 4.2.4: AXI4 Protocol Waveform: Single Read Transfer.

Concerning this, the master bus read last signal, *RLAST*, comes at the same time as the *RVALID* signal, in *syscall_mbus_read* (fig. 4.2.4), as there is only one data element to read from the system memory, or when the slave data changes to the last element, in *syscall_mbus_read_burst* (fig. 4.2.5). As seen earlier in this section, the last step of the two system calls described utilizes a safeguard correction that redirects the flow of microinstructions to step zero, if a glitch occurs.

The system calls to write one or multiple words to the system memory, *syscall_mbus_write(_burst)*, are represented in algorithm 13. As it can be seen in the waveforms of fig. 4.2.6 and fig. 4.2.7, the AXI4 write transfer starts with the specification of the write address and the assertion of the *AWVALID* signal by the AXI4 bus master. The latter then waits for the slave to assert the *AWREADY* signal to indicate the acceptance of the address. Microprogram-wise, this can be observed in step zero of algorithm 13, where one tests the m_axi_awready signal and selects the m_axi_awvalid as the active control signal only proceeding to step two with a true condition on the assertion of *AWREADY*. The next phase of the protocol is the write data phase. In this phase, the master stipulates the data to send and activates the

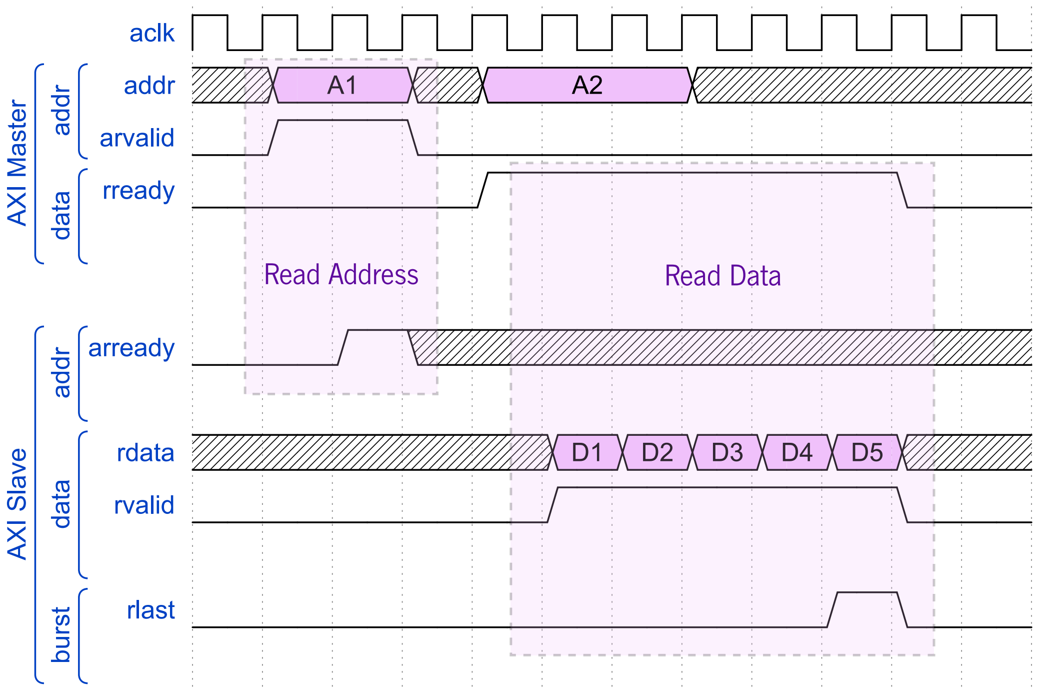


Figure 4.2.5: AXI4 Protocol Waveform: Multiple Read Transfer.

WVALID signal, waiting for the slave response with the *WREADY* signal. If there is only one element to write the master asserts the write last signal, *WLAST*, as soon as the data to write is determined, fig. 4.2.6, in this case, this job pertains to system-level datapath, section 4.2.5. With this setup, making a single write to the system memory represents a burst write of size one. Otherwise, if the burst size is greater than one element the *WLAST* signal is asserted by the master when writing the last element, fig. 4.2.7. This is depicted in algorithm 13 with the testing of the logical *AND* between the *m_axi_wready* and *m_axi_wlast* signals, and the selection of the m_axi_wvalid control signal, in the step one of the microprogram. The AXI4 write transfer is considered complete when the master asserts the bus ready signal, *BREADY*, and cycles after, the slave asserts the bus valid signal, *BVALID*. Typically, the protocol revolves around the valid-ready handshake, meaning that the first asserted signal is valid followed by a ready signal. In this case, as the master operates faster, it asserts its ready signal before the slave asserts its valid signals. The bus valid-ready handshake that completes the write transfer is represented in step two of the microprogram of algorithm 13. Thus, the microinstruction specifies the testing of the *m_axi_bvalid* signal and selection of the *m_axi_bready* control signal. Additionally, it is important to note that the write transfers microprogrammed by this system call make use of the event manager unit, section 4.2.5.1, to specify a timeout on the *BVALID* signal. The event manager unit generates the timeout signal if *BVALID* is asserted within the number of clock cycles specified or with the unit's counter overflow. With this purpose, the step two also selects the control signal that triggers the event manager unit, and it is continuously testing the timeout of the latter, before moving to the exit step, in this case, step three.

Algorithm 13 Microprogram that writes one or multiple words to the SYSRAM (🟥)

```
    pseudocode SYSCALL_MBUS_WRITE(_BURST)
1:  Step 0 : test AXI write address ready select AXI write address valid produce block task
2:      if condition false then goto step zero.
3:  Step 1 : test AXI write ready and bus write last select master AXI write valid produce block task
4:      if condition false then goto step one.
5:  Step 2 : test AXI bvalid and EVM timeout select AXI bready and EVM trigger produce valid
6:      if condition false then goto step two.
7:  Step 3 : test auxiliary false select null produce valid                    ▷ Exit Step
8:      if condition false then goto step zero.
    end pseudocode
```

Both of the master bus system calls, *syscall_mbus_read_burst* and *syscall_mbus_write_burst*, rely on the index manager unit section 4.2.5.2, for the same reasons specified in the read and write burst syscalls referent to the local memory, i.e., for incrementing the index of the buffers on the hardware task.

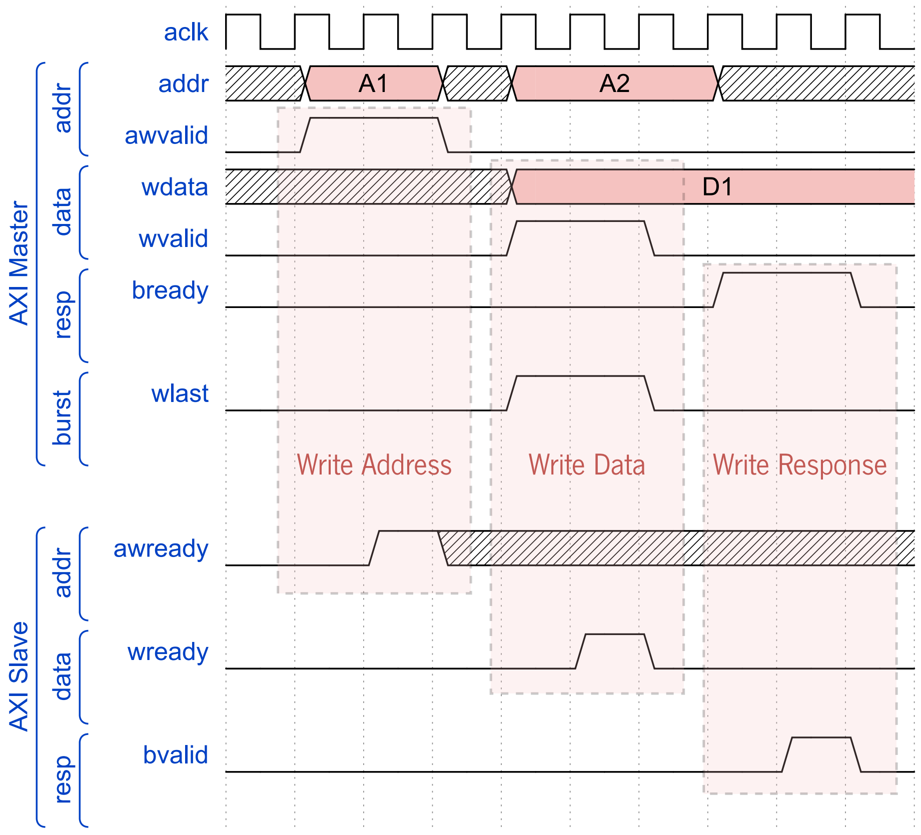


Figure 4.2.6: AXI4 Protocol Waveform: Single Write Transfer.

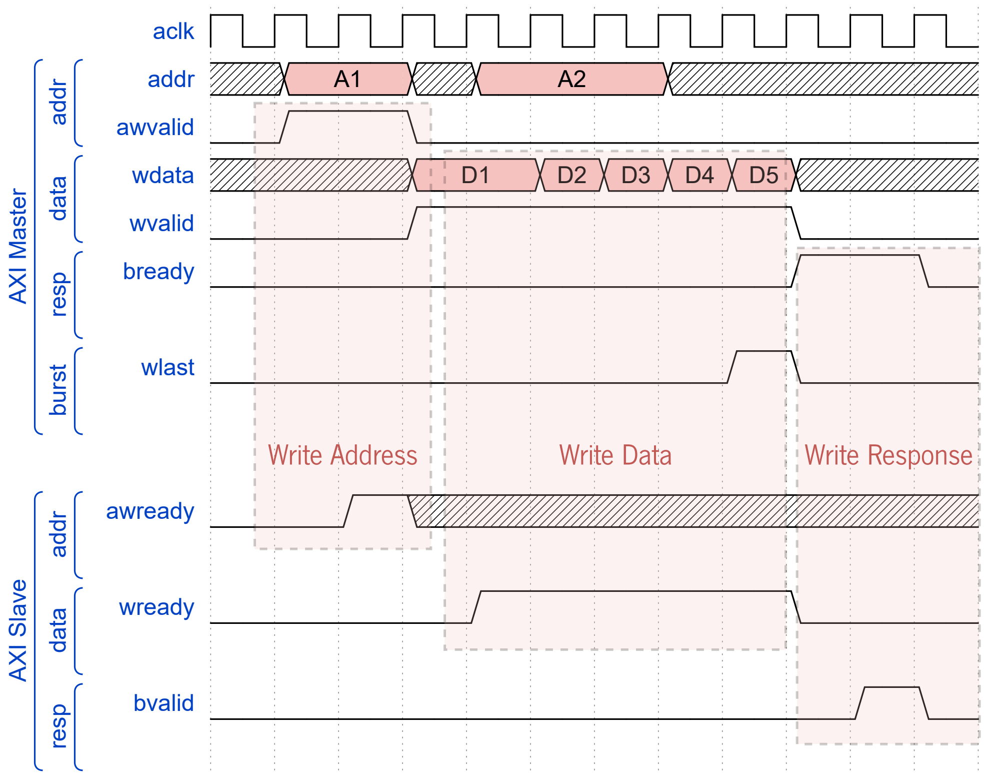


Figure 4.2.7: AXI4 Protocol Waveform: Multiple Write Transfer.

4.2.4 Native Microprogram

The microprogram unit of the dissertation is responsible for the execution of the system calls described earlier in section 4.2.3, i.e., defines the active control signals until the next clock transition, and is also responsible for their sequencing depending on the testing inputs. Similarly to [62], the microprogram uses horizontal microinstructions as forementioned in section 2.3.1.2, with single address, meaning that the field decode of the contents gives origin to a series of control signals with minimal encoding, and that is needed only one address to specify the location of a certain microinstruction in the microcode ROM, respectively.

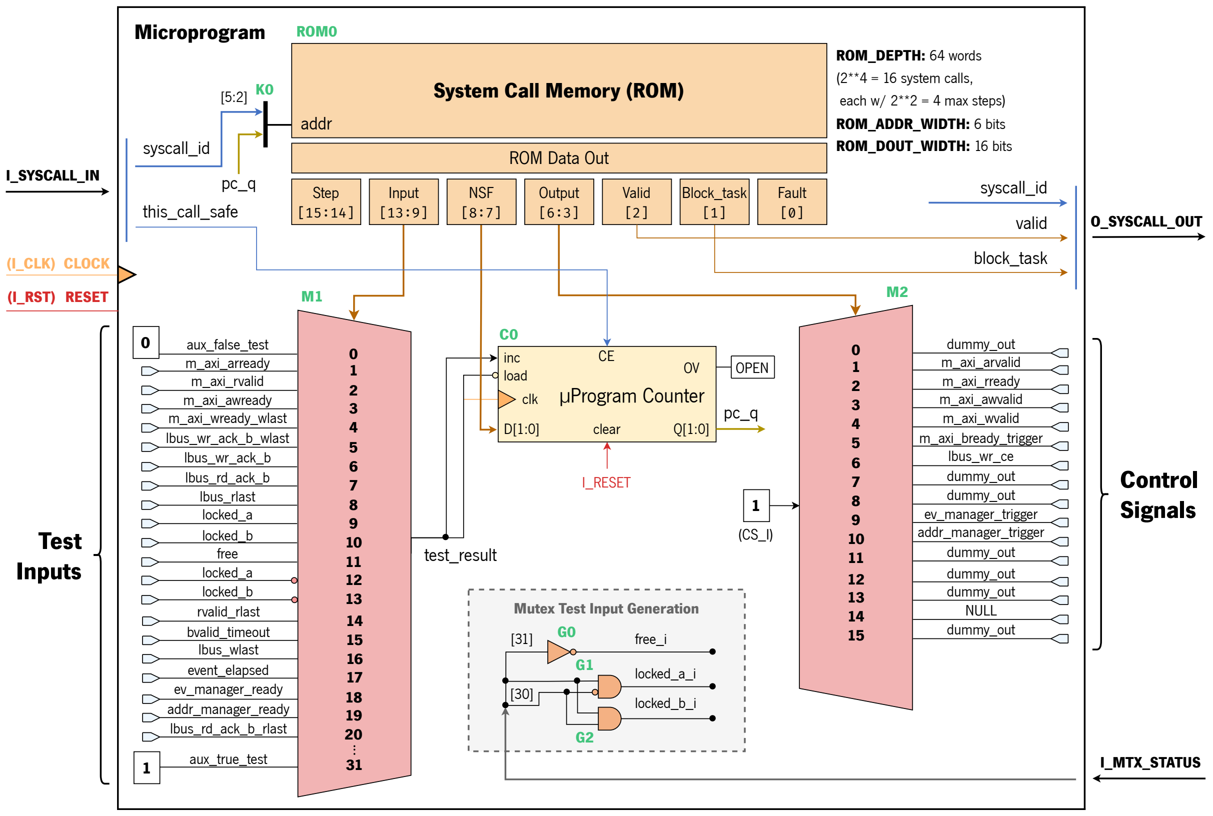


Figure 4.2.8: Microprogram RTL Design Internal Architecture, adapted [62].

Figure 4.2.8 represents the microprogram RTL internal architecture. In it one can see three main elements that constitute the microprogram: (1) the microcode memory, in this case a ROM (🟧); (2) the associated field decode multiplexer M2, which defines the active control signals; (3) a counter with load capabilities (🟨), that together with the multiplexer M1 (🟥), stipulates the flow of microinstructions, i.e., sequences the microinstructions. The third logic element mentioned can be considered the microcode sequencer to a certain extent, section 2.3.1.2, following the literature's nomenclature. Additionally, the microprogram also has some logic (⬜) to decode the mutex status word into the locked A, locked B, and free flags.

The microcode memory can be implemented as a ROM or RAM. As stated earlier in section 2.3.2, the microcode storage started with read-only memories, and later, as the need for microcode updates arose, i.e., the demand for changing the contents of the microcode due to hardware errors increased, the microinstructions started being stored in random access memories (RAMs). This particular section will refer to the microcode memory element as a ROM, represented in orange (🟧), fig. 4.2.8. As it is possible to see, the microcode ROM depth is sixty-four words, meaning that it can hold up to sixteen system calls of four steps each, this system calls are the ones described in section 4.2.3. The ROM address width is six bits because one needs four bits to specify the system call out of the sixteen possible, and another two bits to stipulate the current step of the latter. The microcoded system calls are represented in binary format in the listing 4.2.

Listing 4.2: System Call Memory: Combinational ROM (HDL).

```
-- MORE
subtype rom_word_t is std_logic_vector(ROM_DATA_WIDTH-1 downto 0);
type syscall_rom_t is array (0 to ROM_DEPTH-1) of rom_word_t;
signal syscall_rom : syscall_rom_t := (

    -- SYSCALL_WORK_NONE           -- SYSCALL_LBUS_WRITE       -- SYSCALL_MBUS_READ
    0 =>  "0000000001110100",      24 => "0000110000110010",   48 => "0000001000001010",
    1 =>  "0100000001110010",      25 => "0100000001110100",   49 => "0101110010010010",
    2 =>  "1000000001110010",      26 => "1000000001110010",   50 => "1000000001110100",
    3 =>  "1100000001110010",      27 => "1100000001110010",   51 => "1100000001110010",

    -- SYSCALL_WORK_YIELD          -- SYSCALL_LBUS_READ_BURST  -- SYSCALL_MBUS_WRITE
    4 =>  "0010101001000010",      28 => "0010011001010010",   52 => "0000011000011010",
    5 =>  "0100000001110100",      29 => "0110100011110010",   53 => "0100100010100010",
    6 =>  "1000000001110010",      30 => "1001000001110100",   54 => "1001111100101010",
    7 =>  "1100000001110010",      31 => "1100000001110010",   55 => "1100000001110100",

    -- SYSCALL_WAIT_EVENT_TIMEOUT  -- SYSCALL_LBUS_WRITE_BURST -- SYSCALL_MBUS_READ_BURST
    8 =>  "0010010001110010",      32 => "0010011001010010",   56 => "0000001000001010",
    9 =>  "0110001011001010",      33 => "0100101010110010",   57 => "0101110010010010",
    10 => "1000000001110100",      34 => "1000000001110100",   58 => "1000000001110100",
    11 => "1100000001110010",      35 => "1100000001110010",   59 => "1100000001110010",

    -- SYSCALL_LINTC_READ          -- SYSCALL_MUTEX_LOCK       -- SYSCALL_MBUS_WRITE_BURST
    12 => "0000000001110100",      36 => "0001100000111010",   60 => "0000011000011010",
    13 => "0100000001110010",      37 => "0111111100110010",   61 => "0100100010100010",
    14 => "1000000001110010",      38 => "1001010000111010",   62 => "1001111100101010",
    15 => "1100000001110010",      39 => "1100000001110100",   63 => "1100000001110100");

    -- SYSCALL_LINTC_WRITE         -- SYSCALL_MUTEX_TRY_LOCK
    16 => "0000000001110100",      40 => "0001100000111010",
    17 => "0100000001110010",      41 => "0111111100110010",
    18 => "1000000001110010",      42 => "1001010110111010",
    19 => "1100000001110010",      43 => "1100000001110100",

    -- SYSCALL_LBUS_READ           -- SYSCALL_MUTEX_UNLOCK
    20 => "0000111001110010",      44 => "0001100110111010",
    21 => "0100000001110100",      45 => "0101010110111010",
    22 => "1000000001110010",      46 => "1001011000110010",
    23 => "1100000001110010",      47 => "1100000001110100",

begin
    O_DOUT <= syscall_rom(to_integer(unsigned(I_ADDR)));
end architecture rtl;
```

As it is possible to observe, the memory is fully combinational to avoid any latency associated with the reads performed, and it has a data out width of sixteen bits to accommodate the microinstruction fields. Namely, the step field, the input field, the "next step when false" (NSF) field, the output field, the valid bit, the block task bit, and ultimately, the fault bit.

Figure 4.2.9 depicts the memory representation of four microinstructions from two system calls that read and write to the master bus. These syscalls can also be seen in listing 4.2 as they are stored in the system call ROM of the microprogram. With this schematic one intends to portray the translation of the binary fields of the system call to actual signals. In the case of the first step of *syscall_mbus_read*, the two MSB identify the step as zero, selects input one for testing, in this case the *ARREADY* signal, determines the next step on a false test, NSF is zero, asserts the *ARVALID* control signal, and blocks the hardware task, with the assertion of the block bit. The other microinstructions present behave in the same way, with all the decoding of test and control signals represented in listing 4.3 and listing 4.6.

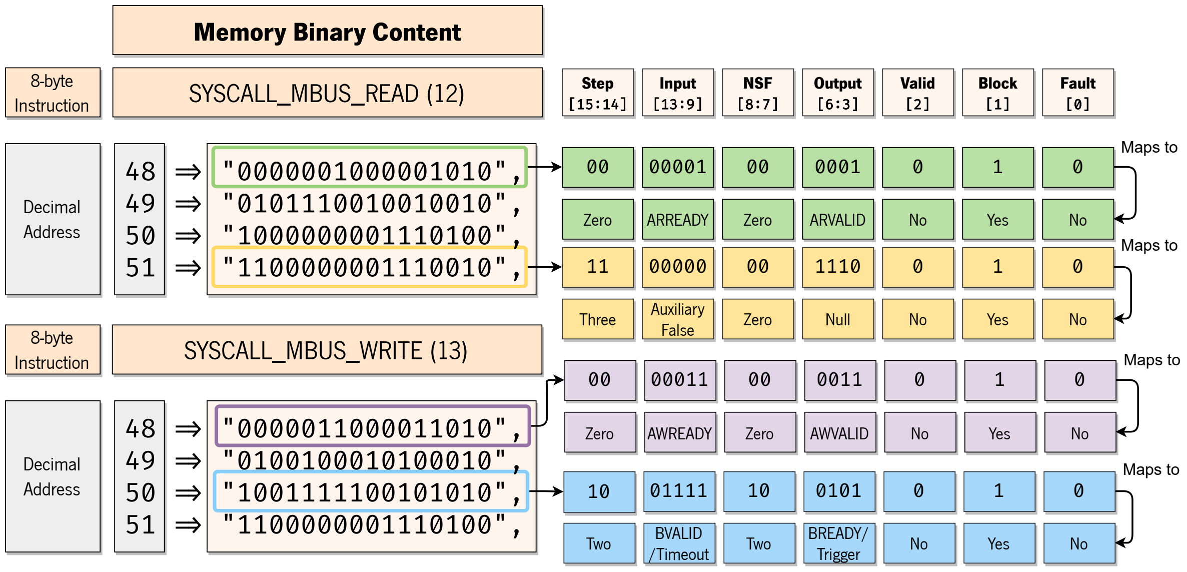


Figure 4.2.9: Memory representation of four microinstructions, extended in table B.1 and table B.2.

Concerning purpose, the step field assures that the microinstruction is identified by its step bits, this makes it easier to update the microcoded system calls manually. The step field only has two bits because that is what it takes to represent all the four doable steps of a certain microinstruction. Additionally, one makes room, so the latter can be propagated to the top-level module of the microkernel in future implementations, having a representation, for example, on the kernel response interface or in some status register available to the host. With this, one means that there is space for exposing the current step of the system call to the co-designer either in software or hardware, but the same applies to the other fields as well.

The input field of the microinstruction dictates the currently active line of the test input multiplexer M1 (🟥). Thus, it has five bits to encompass all the tests from zero to thirty-one, i.e., two to the power of five, minus one. Each line of the aforementioned tests can represent directly a single bit signal, e.g., *m_axi_awready* or *locked_a*, or represent a logical function of two single bit signals, e.g., the *bvalid_timeout* input is actually implemented as a logical *AND* between the *m_axi_bvalid* signal and the timeout signal coming from the event manager in the system-level datapath, section 4.2.5.1 and section 4.2.5, respectively. Auxiliary false and true test lines are also available to permit the microinstruction flows that use unconditional jumps.

Listing 4.3: Microprogram: Input Multiplexer (HDL).

```vhdl
-- Input decoding
--------------------------------------------------------------------------------------
INPUT_DECODING_M1 : process (rom_dout(SYSCALL_INPUT'RANGE),I_M_AXI_ARREADY,
I_M_AXI_RVALID,I_M_AXI_AWREADY,I_M_AXI_WREADY,I_M_AXI_BVALID,locked_a_i,
locked_b_i,free_i,I_M_AXI_RLAST,I_LBUS_WR_ACK_B,I_M_AXI_WLAST,I_EVENT_ELAPSED,
I_LBUS_RLAST,I_EV_MANAGER_READY,I_LBUS_WLAST,I_LBUS_RD_ACK_B,I_ADDR_MANAGER_READY,
I_M_AXI_TIMEOUT)
--------------------------------------------------------------------------------------
    variable select_M1 : integer range 0 TO (2**C_SYSCALL_INPUT_WIDTH)-1;
begin
    select_M1 := to_integer(unsigned(rom_dout(SYSCALL_INPUT'RANGE)));
    data_M1 <= '0';
    case select_M1 is
        when C_AUX_TEST_FALSE_OFFSET        => data_M1 <= '0';                   --00
        when C_ARREADY_OFFSET               => data_M1 <= I_M_AXI_ARREADY;       --01
        when C_RVALID_OFFSET                => data_M1 <= I_M_AXI_RVALID;        --02
        when C_AWREADY_OFFSET               => data_M1 <= I_M_AXI_AWREADY;       --03
        when C_WREADY_WLAST_OFFSET          => data_M1 <= I_M_AXI_WREADY and     --04
                                                          I_M_AXI_WLAST;         --""
        when C_LBUS_WR_ACK_B_WLAST_OFFSET   => data_M1 <= I_LBUS_WR_ACK_B and    --05
                                                          I_LBUS_WLAST;          --""
        when C_LBUS_WR_ACK_B_OFFSET         => data_M1 <= I_LBUS_WR_ACK_B;       --06
        when C_LBUS_RD_ACK_B_OFFSET         => data_M1 <= I_LBUS_RD_ACK_B;       --07
        when C_LBUS_RLAST_OFFSET            => data_M1 <= I_LBUS_RLAST;          --08
        when C_LOCKED_A_OFFSET              => data_M1 <= locked_a_i;            --09
        when C_LOCKED_B_OFFSET              => data_M1 <= locked_b_i;            --10
        when C_FREE_OFFSET                  => data_M1 <= free_i;                --11
        when C_NOT_LOCKED_A_OFFSET          => data_M1 <= not(locked_a_i);       --12
        when C_NOT_LOCKED_B_OFFSET          => data_M1 <= not(locked_b_i);       --13
        when C_RVALID_RLAST_OFFSET          => data_M1 <= I_M_AXI_RVALID and     --14
                                                          I_M_AXI_RLAST;         --""
        when C_BVALID_TIMEOUT_OFFSET        => data_M1 <= I_M_AXI_BVALID or      --15
                                                          I_M_AXI_TIMEOUT;       --""
        when C_LBUS_WLAST_OFFSET            => data_M1 <= I_LBUS_WLAST;          --16
        when C_EVENT_ELAPSED_OFFSET         => data_M1 <= I_EVENT_ELAPSED;       --17
        when C_EV_MANAGER_READY_OFFSET      => data_M1 <= I_EV_MANAGER_READY;    --18
        when C_ADDR_MANAGER_READY_OFFSET    => data_M1 <= I_ADDR_MANAGER_READY;  --19
        when C_LBUS_RD_ACK_B_RLAST_OFFSET   => data_M1 <= I_LBUS_RD_ACK_B and    --20
                                                          I_LBUS_RLAST;          --""

        when C_INTR_RAISE_OFFSET            => data_M1 <= I_INTR_RAISE;          --22
        -- ...                                                                   ----
        when C_AUX_TEST_TRUE_OFFSET         => data_M1 <= '1';                   --31
        when others                         => null;
    end case;
end process INPUT_DECODING_M1;
--------------------------------------------------------------------------------------
test_result <= data_M1;
test_resultn <= not(data_M1);
```

The NSF field is closely related with the loadable counter element C0, represented in yellow (🟨), fig. 4.2.8, so it is important to firstly understand this logic element before proceeding. The internal architecture of this counter with load is represented in fig. 4.2.10. As the name indicates, the counter presented has capabilities to increment, load, and remain in standby, by loading the same value of the last clock transition. Taking a closer look at the RTL, one notices that a multiplexer M0 (🟥) is used to decide the combinational input of the counter register FF0 (🟨). When the concatenation of the *I_LOAD* and *I_INC* signals gives "00", i.e., zero, the counter register FF0 (🟨) combinational input *D* is the concatenation (K1) of the counter output *Q* with the counter overflow register output *Q*, assuring that in the next clock cycle the value of *O_Q* will be the same.

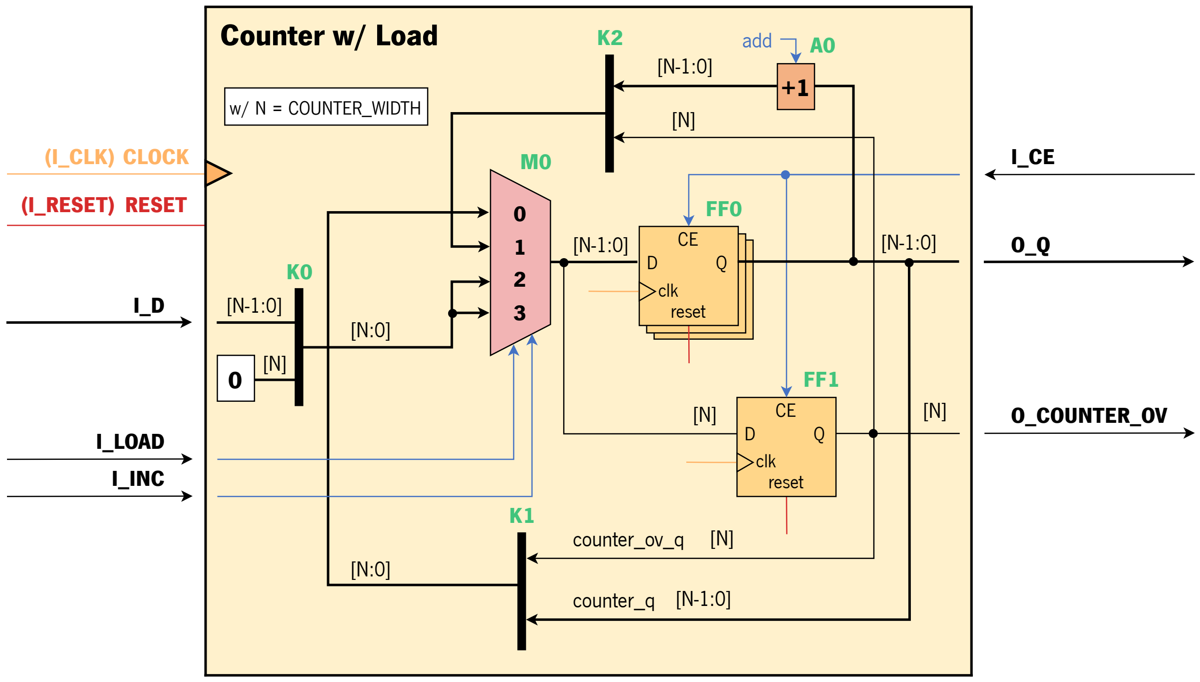


Figure 4.2.10: Loadable Counter RTL Design Internal Architecture.

Additionally, when the concatenation of *I_LOAD* and *I_INC* gives "01", i.e., one, the logic element behaves as a normal counter and increments the value on the register FF0 (🟨). This is depicted by the add operation performed by the ALU element A0 (🟧). One makes sure that, when incrementing, the counter overflow given by the FF1 register is kept, with the concatenation in K2. When the M0 (🟥) selection is "10" or "11", i.e., two or three, the counter loads the value at *I_D* and clears the overflow bit in K0. When both the load and increment signals are asserted, "11", the module gives priority to the load operation. The HDL representation of the design of fig. 4.2.10 is depicted by listing 4.4.

Listing 4.4: Counter with Load (HDL).

```
-- Counter data decoding
---------------------------------------------------------------
COUNTER_MUX_C0 : process (I_INC,I_LOAD,counter_Q,I_D,counter_ov_Q)
---------------------------------------------------------------
```

```
    variable select_C0 : std_logic_vector(1 downto 0);
begin
    select_C0 := I_LOAD & I_INC;
    counter_D <= (others => '0');
    case select_C0 is
        when "00" =>
             counter_D <= counter_ov_Q & unsigned(counter_Q);
        when "01" =>
             counter_D <= (counter_ov_Q & ZEROS) OR ('0' & unsigned(counter_Q) + 1);
        when "10" =>
             counter_D <= '0' & unsigned(I_D);
        when "11" =>
             counter_D <= '0' & unsigned(I_D);
        when others =>
            null;
    end case;
end process COUNTER_MUX_C0;
------------------------------------------------------------------
-- Counter register and overflow register
------------------------------------------------------------------
COUNTER_REG_C0 : process (I_CLK)
------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if I_RESET = '1' then
            counter_Q    <= (others => '0');
            counter_ov_Q <= '0';
        elsif (I_CE = '1') then
            counter_Q    <= std_logic_vector(counter_D(COUNT_WIDTH-1 downto 0));
            counter_ov_Q <= counter_D(COUNT_WIDTH);
        end if;
    end if;
end process COUNTER_REG_C0;
------------------------------------------------------------------
O_COUNTER_OV <= counter_ov_Q;
O_Q <= counter_Q;
```

By understanding the counter element, it is now conceivable to explain how it acts as the microprogram counter. The combinational output of multiplexer M1 (🟥) generates the test result that will serve as the increment and load signals, as one can see in fig. 4.2.8. Regarding that, the *test_result* signal itself is connected to the *INC* input port of the counter C0, and the logical *NOT* of the *test_result* signal is connected to the counter's *LOAD* input port. With this one achieves an increment of the system call step on a true test of the input flags and a jump to a different step on a false result. The latter is possible since the microcode ROM address is given by the concatenation of the *sycall_id* signal, stipulated in a certain kernel-level procedure, and the output *Q* of the microprogram counter C0, which gives the next step of the system call. The counter in question is enabled by the *this_call* signal, connected to its clock enable (CE) input port. The mention to the *this_call_safe* signal in fig. 4.2.8 is justified since the *this_call* signal is debounced somewhere outside the module. In this specific case the overflow port of the counter is left *OPEN* because it is not used within the microprogram. Although, it is useful in other scenarios like in the event manager unit of section 4.2.5.1. The HDL containing the instantiation of the microcode ROM and the microprogram counter is portrayed by listing 4.5, with the constants and definitions used in the configuration package in listing A.1.

Listing 4.5: Microprogram: Microcode ROM and Microprogram Counter (HDL).

```
rom_addr(SYSCALL_MSB'RANGE) <= std_logic_vector(to_unsigned(SYSCALL_T'POS(I_SYSCALL_INPUT.
    SYSCALL_ID),SYSCALL_MSB'LENGTH));

--System Call Memory (ROM)
--------------------------------------------------------------------------
SYSCALL_ROM0 : entity syscall_mem
--------------------------------------------------------------------------
port map(
I_ADDR     => rom_addr,
O_DOUT     => rom_dout );
--------------------------------------------------------------------------
-- Microprogram Counter
--------------------------------------------------------------------------
UPROGRAM_COUNTER_C0 : entity COUNTER_LOAD
--------------------------------------------------------------------------
generic map(
COUNT_WIDTH => C_SYSCALL_NSF_WIDTH)
port map(
I_CLK        => I_CLK,
I_CE         => I_SYSCALL_INPUT.THIS_CALL,
I_RESET      => I_RESET,
I_INC        => test_result,
I_LOAD       => test_resultn,
I_D          => rom_dout(SYSCALL_NSF'RANGE),
O_Q          => rom_addr(SYSCALL_MSB'LOW-1 downto 0),
O_COUNTER_OV => OPEN );
--------------------------------------------------------------------------
```

Continuing with the explanation of the microinstruction fields, as forementioned, the output field is used to specify the active control signals, and, therefore, it is used as the select signal of the multiplexer M2, represented in dark rose (🟥), fig. 4.2.8. The HDL for the multiplexer M2 is represented in listing 4.6, alongside with the mutex status decode logic and output assignment in the listing 4.7.

Listing 4.6: Microprogram: Output Multiplexer (HDL).

```
-- Output encoding
--------------------------------------------------------------------------------------
OUTPUT_ENCODING_M2 : process (rom_dout(SYSCALL_OUTPUT'RANGE),CS_I)
--------------------------------------------------------------------------------------
    variable select_M2 : integer range 0 TO (2**C_SYSCALL_OUTPUT_WIDTH)-1;
begin
    select_M2 := to_integer(unsigned(rom_dout(SYSCALL_OUTPUT'RANGE)));
    O_M_AXI_ARVALID         <= '0';
    O_M_AXI_RREADY          <= '0';
    O_M_AXI_AWVALID         <= '0';
    O_M_AXI_WVALID          <= '0';
    O_M_AXI_BREADY          <= '0';
    O_LBUS_WR_CE            <= '0';
    O_LBUS_RD_CE            <= '0';
    O_EV_MANAGER_TRIGGER    <= '0';
    O_ADDR_MANAGER_TRIGGER  <= '0';
    O_M00_TIMEOUT_TRIGGER   <= '0';
    case select_M2 is
        when C_ARVALID_OFFSET               => O_M_AXI_ARVALID         <= CS_I; --01
        when C_RREADY_OFFSET                => O_M_AXI_RREADY          <= CS_I; --02
        when C_AWVALID_OFFSET               => O_M_AXI_AWVALID         <= CS_I; --03
        when C_WVALID_OFFSET                => O_M_AXI_WVALID          <= CS_I; --04
        when C_BREADY_M00_TRIGGER_OFFSET    => O_M_AXI_BREADY          <= CS_I; --05
                                               O_M00_TIMEOUT_TRIGGER   <= CS_I; --""
        when C_LBUS_WR_CE_OFFSET            => O_LBUS_WR_CE            <= CS_I; --06
        when C_LBUS_RD_CE_OFFSET            => O_LBUS_RD_CE            <= CS_I; --07
        -- ...                                                                  ----
        when C_EV_MANAGER_TRIGGER_OFFSET    => O_EV_MANAGER_TRIGGER    <= CS_I; --09
        when C_ADDR_MANAGER_TRIGGER_OFFSET  => O_ADDR_MANAGER_TRIGGER  <= CS_I; --10
        -- ...                                                                  ----
        when C_NULL_OFFSET                  => null;                            --14
        when others                          => null;
    end case;
end process OUTPUT_ENCODING_M2;
--------------------------------------------------------------------------------------
```

Finally, the *valid* and *block_task* bits are related with the communication with the hardware task module. The *block_task* bit stops the hardware task, since it is used in the flip-flop that holds the next state of the hardware task's FSM, section 4.3. The valid bit is related with a valid return argument in the return argument register of the system-level datapath, section 4.2.5, and also it indicates that the hardware task can proceed its execution to the next state, where another system call can be executed, considering single syscall procedures.

Listing 4.7: Microprogram: Mutex Status Decode Logic and Output Assignment (HDL).

```
-- Mutex status decode
-----------------------------------------------------------------------
locked_a_i <= I_STATUS(C_LOCKED_BIT) and not(I_STATUS(C_CHANNEL_ID_BIT));
locked_b_i <= I_STATUS(C_LOCKED_BIT) and I_STATUS(C_CHANNEL_ID_BIT);
free_i     <= not(I_STATUS(C_LOCKED_BIT));
-----------------------------------------------------------------------
O_SYSCALL_OUTPUT.valid      <= rom_dout(C_SYSCALL_VALID_BIT);
O_SYSCALL_OUTPUT.block_task <= rom_dout(C_SYSCALL_BLOCK_TASK_BIT);
O_SYSCALL_OUTPUT.syscall_id <= I_SYSCALL_INPUT.syscall_id;
```

4.2.5 System-Level Datapath

The system-level datapath unit functions as a supporting entity of the microprogram, section 4.2.4. Thus, it contains all the datapath elements for the specification of the system call arguments and returns, and also entities that help the microprogram manage timeouts and events, section 4.2.5.1, and indexes/addresses, section 4.2.5.2 and section 4.2.5.3, respectively. The RTL description of the system-level datapath is present in fig. 4.2.11. This section will detail on the datapath entity as a whole, regarding design and implementation, its subunits, i.e., the event manager, index manager, and the address manager, the specification of the syscall's arguments and return bit fields, and also the interactions regarding the M00 system interface for the system calls related to the master bus, refer to algorithm 13 and algorithm 12.

Looking closely at fig. 4.2.11, one can see that the system level datapath is almost fully combinational, only having sequential elements regarding the events, index, and address units, and also concerning the system call return flip-flop FF0. The main component of the SLD is the multiplexer M0, represented in light yellow (🟨). This logic element establishes the parameters, the returns and the active units of a certain system call. This section will detail on the aspects of the System-Level Datapath that allow the execution of the system calls and also its established parameters and returns.

The system call that waits a certain amount of clock cycles for an event or timeout, represented earlier

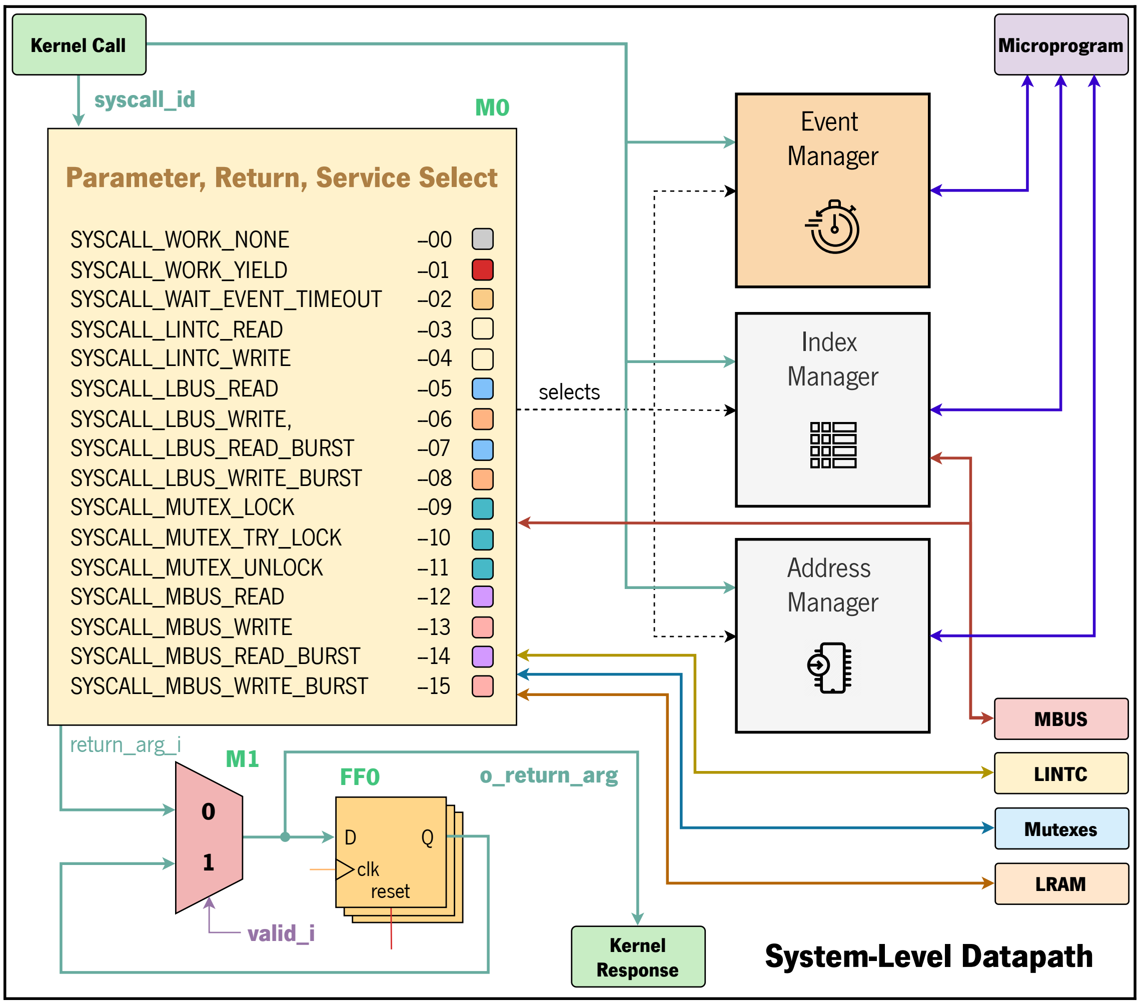


Figure 4.2.11: System-Level Datapath (SLD) RTL Design Internal Architecture (Simplified), expanded in fig. B.1.

in algorithm 3, specifies its calling parameters and return arguments as depicted by fig. 4.2.12. Thus, it only needs one bit to stipulate the event to monitor and sixteen bits to indicate the maximum time on intends to wait for that event or until timeout, if no event is specified. Moreover, the system call returns the timeout status, which is asserted if a timeout occurs, and the time remaining until timeout. Both of the parameters are signals connected with the event manager unit, and both of the return arguments are signals coming from the same unit, as it can be seen as yellow lines (🟨) in fig. 4.2.11. For a detailed explanation of the inner workings of the event manager unit, i.e., how it uses these input signals and how it outputs the forementioned output signals, refer to section 4.2.5.1.

Concerning this, and analyzing the HDL in the SLD multiplexer M0, listing 4.8, the syscall uses the MSB (bit 63) to return the timeout status, in line 10, and the bits from 15 to 0 to return the remaining time, in line 11. Regarding parameters, the bit 62 is used for the event to monitor, in line 12, and the

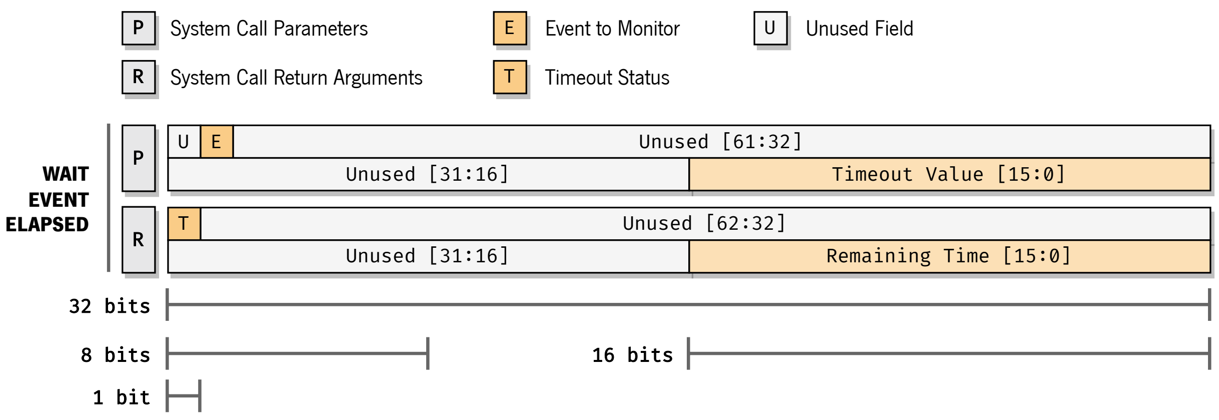


Figure 4.2.12: System-Level Datapath: Syscall Wait Event Timeout Parameters and Returns.

timeout value is specified in line 13. Note that it is important to use the bits from 15 to 0 for the timeout value and remaining time in case one intends to expand the width of the event manager beyond 16 bits. As fig. 4.2.11 and listing 4.8 depict, the system call parameters come directly from the kernel call signal, namely from its parameter field, lines 12 and 13. All the constants and definitions used in the system calls in the SLD are represented in listing A.3.

Listing 4.8: System-Level Datapath: Syscall Wait Event Timeout Parameters and Returns (HDL).

```
 1  PARAM_SERVICE_SELECT_M0 : process (I_KERNEL_CALL.syscall_id,I_KERNEL_CALL.parameters,CS_I,
        timeout_i,time_remaining_i,I_M00_AXI,I_LRAM_DOUT,I_MUTEX_STATUS)
 2  ------------------------------------------------------------------------------------------
 3  begin
 4  --MORE
 5  ------------------------------------------------------------------------------------------
 6  case I_KERNEL_CALL.syscall_id is
 7  --MORE
 8  ------------------------------------------------------------------------------------------
 9  when SYSCALL_WAIT_EVENT_TIMEOUT =>
10    return_arg_i(RPARAM_TIMEOUT_STATUS)         <= timeout_i;          --[63]
11    return_arg_i(RPARAM_REMAINING_TIME'RANGE)    <= time_remaining_i;   --[15:0] (16 bits)
12    event_i       <= I_KERNEL_CALL.parameters(PARAM_EVENT_TO_MONITOR);  --[62]
13    timeout_val_i  <= I_KERNEL_CALL.parameters(PARAM_TIMEOUT_VAL'RANGE); --[15:0] (16 bits)
14  ------------------------------------------------------------------------------------------
```

Regarding the syscalls to read and write to the local memory (LRAM), aforementioned in section 4.2.3 as local bus system calls, one stipulates their parameters and return as represented in fig. 4.2.13 and fig. 4.2.14. In the write of one word to the local memory, one needs to specify the data to write, represented as source data, and the offset to which one intends to write to. This will translate later to an actual address of the Local Random-Access Memory through the use of the address manager, section 4.2.5.3. The only difference between the latter syscall and the burst format one, is that the burst length parameter and return field differs from one word to multiple words, fig. 4.2.13. Since this is a write operation one only returns the amount of words written to memory as result of the syscall, and this applies to both syscall formats, i.e., burst format or single-word format.

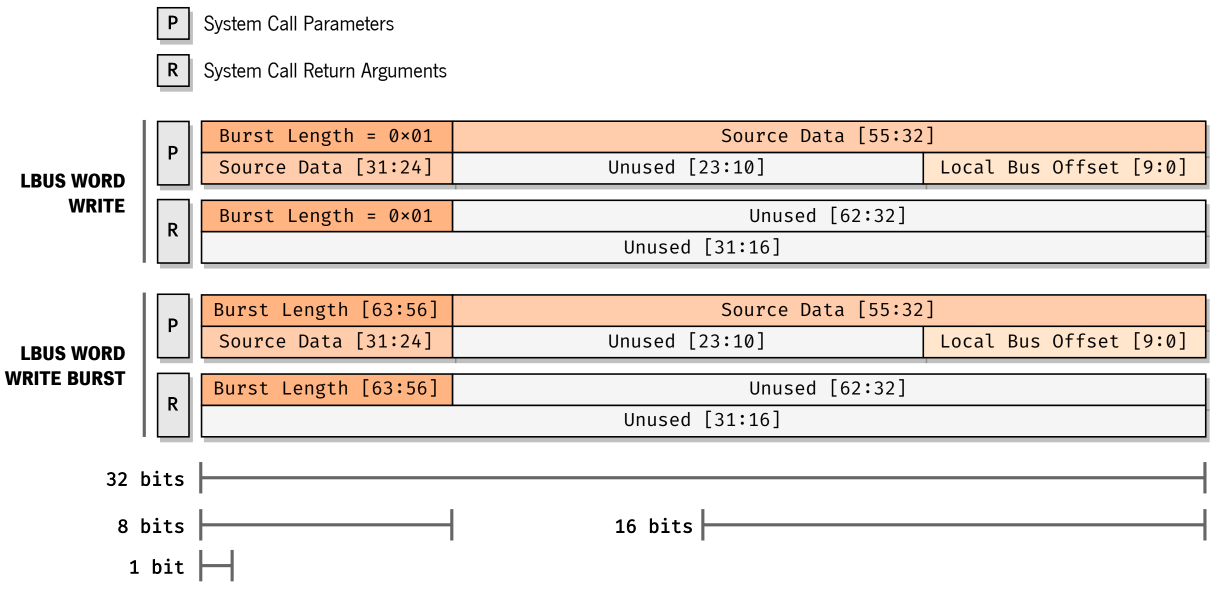


Figure 4.2.13: System-Level Datapath: Local Bus Write Syscalls Parameters and Returns.

Concerning the read operations from the LRAM, one needs to indicate the number of words, as in the write, and from where to read, with the local bus offset. As for returns, now it makes sense to return the read data and keep returning the amount of words read from memory. Once again, the burst format only differs in the amount of words. One made sure to keep the system call parameters and return fields mirrored throughout the design to increase clarity, i.e., for example, the local bus offset is maintained in the least significant 10 bits in writes and reads and makes room for the expansion of this offset up to 24 bits, as it is possible to see in fig. 4.2.14. Additionally, the read data is always returned in the same position as the source data for the writes, from bit 55 to bit 24.

The HDL representation of fig. 4.2.13 and fig. 4.2.14 can be seen in listing 4.9. As it is possible to see, the number of words to write or read, is always placed from bits 63 to 56, lines 10, 16, 22, and 31. This field is named as “burst length” even in single word operations, since this nomenclature is used as well in master bus system calls. The master bus is an AXI4 interface, thus the single word transfers are treated as single word bursts. Even tho the local memory bus does not impose these restrictions, one opted to use the name “burst length” to keep consistency. The single word read, line 9, returns the read word within a 64 bit return from bits 55 to 24, line 11. The same happens for each word read in burst format, in line 23. All the syscalls for write and read also specify the offset from bit 9 to 0, lines 12, 18, 25, and 34, and need to enable the port B chip select of the LRAM, *CS_B*, in lines 13, 19, 26,

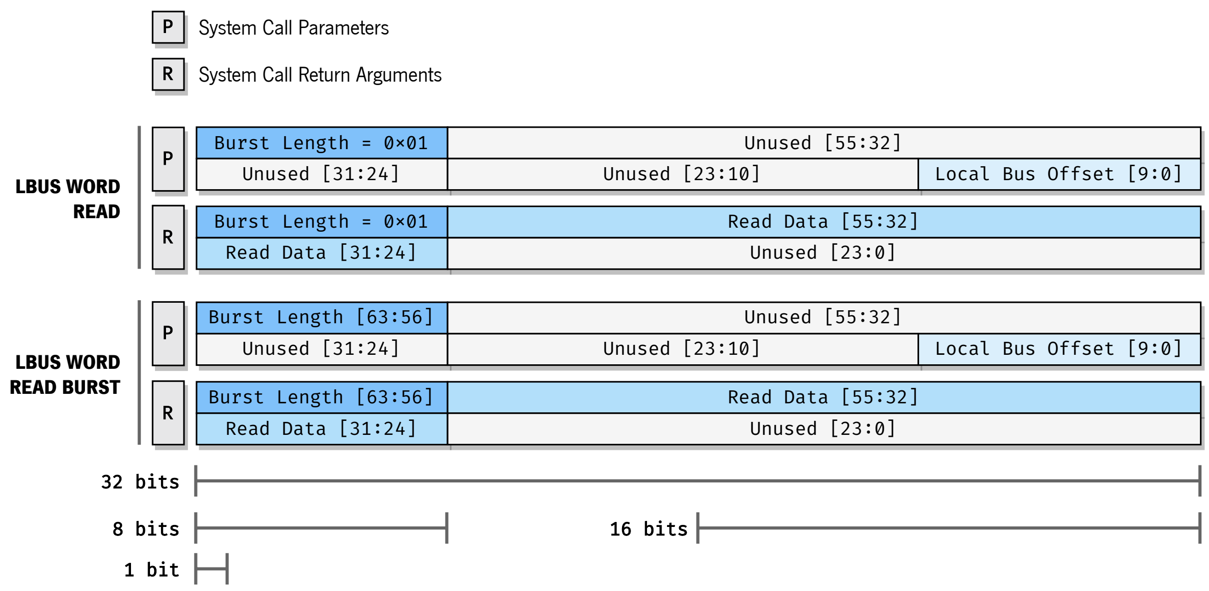


Figure 4.2.14: System-Level Datapath: Local Bus Read Syscalls Parameters and Returns.

and 35. The burst write/read system calls also enable the address and index manager in lines 27/37 and 28/38, respectively, and specify the burst length in lines 24 and 32. Additionally, the single write and the write burst also determine the source data from bits 55 to 24, this can be seen in lines 17 and 33, respectively. The use of the subtype *param_mbus_source* (that can be consulted in listing A.3) is used because, as forementioned, the local bus system calls were designed to mimic the master bus ones in terms of parameters and returns where it was possible. This means that there is no need to create another VHDL subtype to represent the same parameter bit field.

Listing 4.9: System-Level Datapath: Local Bus Write/Read (Burst) Syscalls Parameters and Returns (HDL).

```
1   PARAM_SERVICE_SELECT_M0 : process (I_KERNEL_CALL.syscall_id,I_KERNEL_CALL.parameters,CS_I,
        timeout_i,time_remaining_i,I_M00_AXI,I_LRAM_DOUT,I_MUTEX_STATUS)
2   ------------------------------------------------------------------------------------------
3   begin
4   --MORE
5   ------------------------------------------------------------------------------------------
6   case I_KERNEL_CALL.syscall_id is
7   --MORE
8   ------------------------------------------------------------------------------------------
9   when SYSCALL_LBUS_READ =>
10    return_arg_i(RPARAM_LBUS_BURST_LEN'RANGE) <= I_KERNEL_CALL.parameters(
          PARAM_LBUS_BURST_LEN'RANGE); --[63:56] (8 bits)
11    return_arg_i(RPARAM_LBUS_SOURCE'RANGE) <= I_LRAM_DOUT; --[55:24] (32 bits)
12    lbus_address_i  <= I_KERNEL_CALL.parameters(PARAM_LBUS_OFFSET'RANGE); --[9:0] (10 bits)
13    O_CS_B <= CS_I; -- enable LRAM port b chip select
14  ------------------------------------------------------------------------------------------
15  when SYSCALL_LBUS_WRITE =>
16    return_arg_i(RPARAM_LBUS_BURST_LEN'RANGE) <= I_KERNEL_CALL.parameters(
          PARAM_LBUS_BURST_LEN'RANGE); --[63:56] (8 bits)
17    source_i <= I_KERNEL_CALL.parameters(PARAM_MBUS_SOURCE'RANGE); --[55:24] (32 bits)
18    lbus_address_i <= I_KERNEL_CALL.parameters(PARAM_LBUS_OFFSET'RANGE); --[9:0] (10 bits)
19    O_CS_B <= CS_I; -- enable LRAM port b chip select
20  ------------------------------------------------------------------------------------------
21  when SYSCALL_LBUS_READ_BURST =>
```

```
  return_arg_i(RPARAM_LBUS_BURST_LEN'RANGE) <= I_KERNEL_CALL.parameters(
      PARAM_LBUS_BURST_LEN'RANGE); --[63:56] (8 bits)
  return_arg_i(RPARAM_LBUS_SOURCE'RANGE) <= I_LRAM_DOUT; --[55:24] (32 bits)
  burst_len_i<= to_integer(unsigned(I_KERNEL_CALL.parameters(PARAM_LBUS_BURST_LEN'RANGE)));
       --[63:56] (8 bits)
  lbus_address_i  <= I_KERNEL_CALL.parameters(PARAM_LBUS_OFFSET'RANGE); --[9:0] (10 bits)
  O_CS_B          <= CS_I; -- enable LRAM port b chip select
  enable_addr_i   <= CS_I; -- enable address manager
  enable_index_i  <= CS_I; -- enable index manager
------------------------------------------------------------------------------------------
when SYSCALL_LBUS_WRITE_BURST =>
  return_arg_i(RPARAM_LBUS_BURST_LEN'RANGE) <= I_KERNEL_CALL.parameters(
      PARAM_LBUS_BURST_LEN'RANGE); --[63:56] (8 bits)
  burst_len_i<= to_integer(unsigned(I_KERNEL_CALL.parameters(PARAM_LBUS_BURST_LEN'RANGE)));
       --[63:56] (8 bits)
  source_i <= I_KERNEL_CALL.parameters(PARAM_MBUS_SOURCE'RANGE); --[55:24] (32 bits)
  lbus_address_i  <= I_KERNEL_CALL.parameters(PARAM_LBUS_OFFSET'RANGE); --[9:0] (10 bits)
  O_CS_B          <= CS_I; -- enable LRAM port b chip select
  enable_addr_i   <= CS_I; -- enable address manager
  enable_index_i  <= CS_I; -- enable index manager
------------------------------------------------------------------------------------------
```

As previously mentioned in this section, the system calls to write and read from the system/host memory are similar to the ones that write/read to the local memory to a certain extent. The parameter and return fields are described in fig. 4.2.15 and fig. 4.2.16. Concerning parameters in the write syscalls for the SYSRAM, one has a field for the number of words to be written to memory, a field for the source data to write to a certain memory position, and a master bus offset that determines that memory location. For returns, the syscall gives again the amount of words written to memory, but in the MBUS case, also gives information relative with the timeout of the bus. This is important since as explained earlier in section 4.2.3, the MBUS syscalls, specially the write ones, are bound to wait on bus signals like *BVALID*. For this the write syscalls related to the system memory follow a return scheme similar to the one seen previously in the *wait event timeout* syscall, returning a bit for the bus timeout status and also the remaining time until timeout, as it is possible to see in fig. 4.2.15. As in the LBUS syscalls, the MBUS burst syscalls differ from the single word ones only in the burst length parameter. However, in the case of single word transfers, the burst length must be assigned to one since the interface is AXI4. This means that every transaction uses the burst format even on single word transfers, and that applies for writes and also reads.

The MBUS system calls for reads are depicted in fig. 4.2.16. This system calls do not differ too much from the local bus ones since one does not need to perform any timeout related returning. This happens because in the read there is not a bus acknowledgement stage. Regarding this, the read in both formats also needs a burst length parameter for the same reasons explained earlier for the SYSRAM write syscalls, and a parameter to designate the memory location to read from, named as master bus offset. Additionally, it then returns the amount of words read from memory and the read data, in the locations specified by fig. 4.2.16.

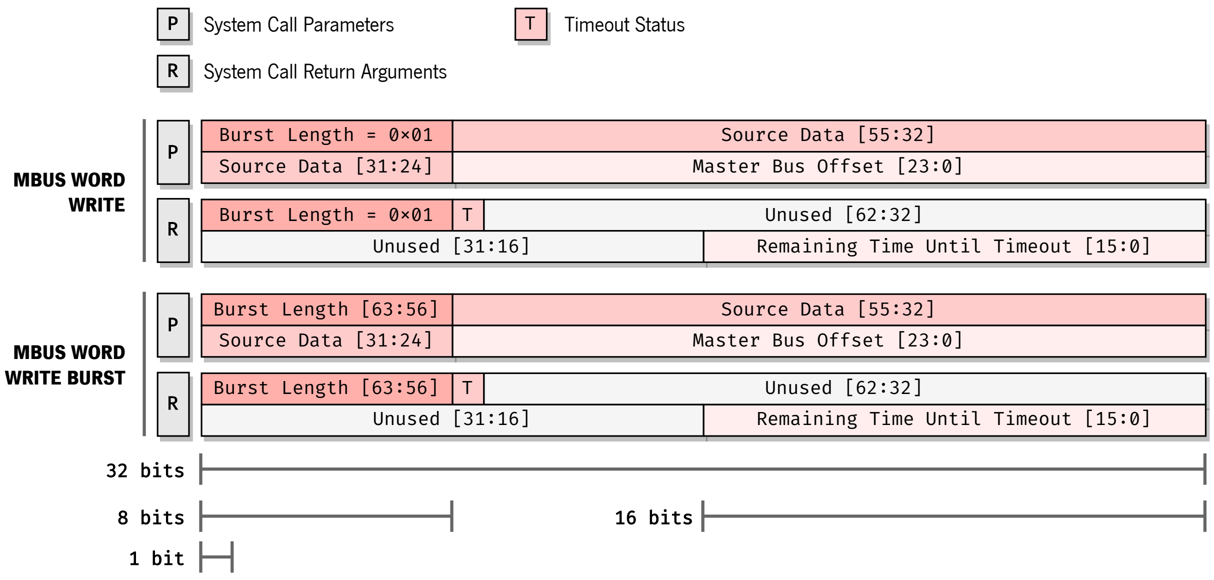


Figure 4.2.15: System-Level Datapath: Master Bus Write Syscalls Parameters and Returns.

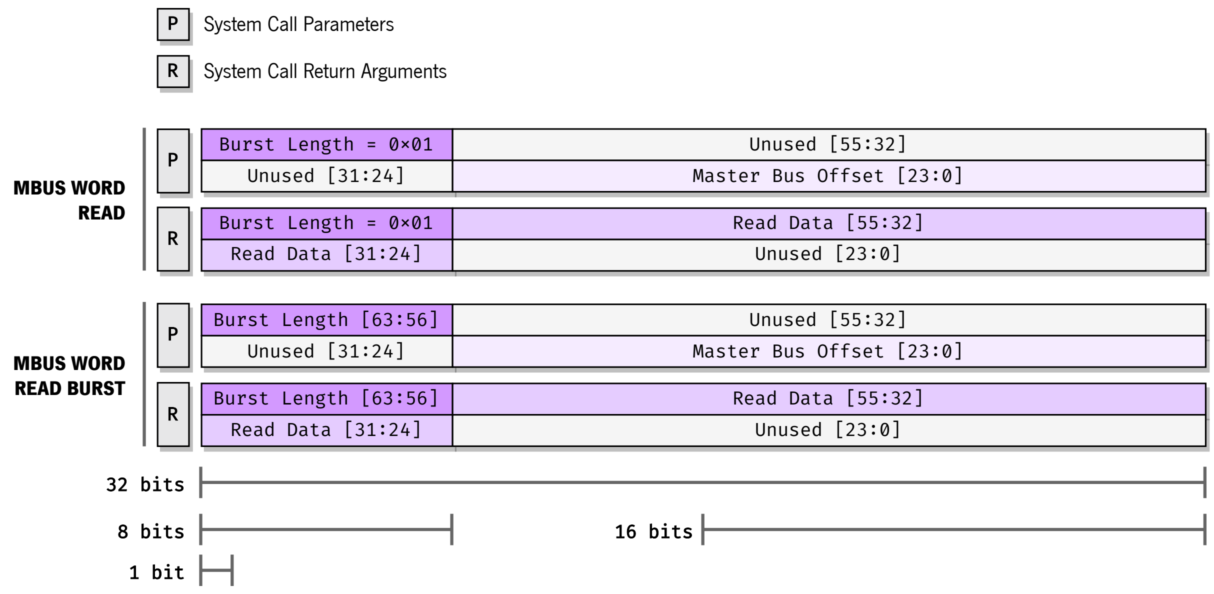


Figure 4.2.16: System-Level Datapath: Master Bus Read Syscalls Parameters and Returns.

In the system calls for single word reads and writes from/to the SYSRAM, line 9 and 15, respectively, one specifies the burst length as one, lines 13 and 22, since the master bus follows an AXI4 interface, with all the transfers occurring in burst format. As part of the AXI4 protocol, for the same reasons, the syscall for single word writes also has to assert the *WLAST* signal in line 21. Regarding timeouts, the latter syscall and the burst write also specify the *BVALID* as the event to monitor, lines 24 and 43, and wait for its assertion within the determined timeout value, lines 25 and 44. This timeout value is given by the constants *c_mbus_timeout_value* and *c_mbus_burst_timeout_value*, which stipulate 10 and 150 cycles for timeout, respectively. These values were adjusted through experimentation with the AXI interface, and set this way since the *BVALID* signal for burst transfers tends to arrive later, i.e., the bus confirmation for the slave took longer with multiple word transactions. Additionally, as seen in fig. 4.2.15, the source data comes within the bits 55 to 24 of the parameters, and it is assigned to an internal signal, *source_i*, in lines 19 and 39. Considering returns, the reads only devolve the number of words read, bits 63 to 56, and the read data, from bits 55 to 32, lines 10/28 and 11/29. As for writes, one also returns the amount of words written, from bits 63 to 56, lines 16 and 35, but also the timeout information, i.e., the timeout status (bit 55), lines 17 and 36, and the time remaining until timeout (bits 15 to 0), lines 18 and 37.

As it happened with the local bus system calls, the burst ones considering the MBUS also have to enable the index manager unit of the SLD, section 4.2.5.2, lines 32 and 41, to be able to handle the read and write from and to buffers in the hardware task entity. These burst syscalls get the number of words to read or write from the parameter bits 63 to 56, lines 30 and 38, which leaves room for multiple word bursts of 256 words of 32 bits, i.e., 1K byte transfers. The system calls to write and read from and to the system memory in both formats specify the MBUS address by adding the MBUS offset (parameter bits 23 to 0, fig. 4.2.15 and fig. 4.2.16) to the base SYSRAM address coming from a host-writable register in the interface registers module, section 4.5.5. This can be seen in lines 12, 20, 31, and 40. The syscalls would need extra address setup if one was considering writes or reads across multiple host memory pages. In this case, pages are fixed on 1K bytes, and one does not consider bursts bigger than the page size.

Listing 4.10: System-Level Datapath: Master Bus Write/Read (Burst) Syscalls Parameters and Returns (HDL).

```
1  PARAM_SERVICE_SELECT_M0 : process (I_KERNEL_CALL.syscall_id,I_KERNEL_CALL.parameters,CS_I,
       timeout_i,time_remaining_i,I_M00_AXI,I_LRAM_DOUT,I_MUTEX_STATUS)
2  ------------------------------------------------------------------------------------------
3  begin
4  --MORE
5  ------------------------------------------------------------------------------------------
6  case I_KERNEL_CALL.syscall_id is
7  --MORE
8  ------------------------------------------------------------------------------------------
9  when SYSCALL_MBUS_READ =>
10  return_arg_i(RPARAM_MBUS_BURST_LEN'RANGE) <= std_logic_vector(to_unsigned(1,return_arg_i(
        RPARAM_MBUS_BURST_LEN'RANGE)'LENGTH)); --[63:56] (8 bits)
11  return_arg_i(RPARAM_MBUS_SOURCE'RANGE) <= I_M00_AXI_rdata; --[55:32] (32 bits)
12  mbus_address_i <= std_logic_vector(unsigned(I_SYSRAM_BASE_ADDR) + unsigned(
        std_logic_vector(to_unsigned(to_integer(unsigned(I_KERNEL_CALL.parameters(
        PARAM_MBUS_OFFSET'RANGE))),mbus_address_i'LENGTH)))); --[23:0] (24b padded to 32)
13  burst_len_i <= 1; -- force burst length to one
14 ------------------------------------------------------------------------------------------
15 when SYSCALL_MBUS_WRITE =>
16  return_arg_i(RPARAM_MBUS_BURST_LEN'RANGE) <= std_logic_vector(to_unsigned(1,return_arg_i(
        RPARAM_MBUS_BURST_LEN'RANGE)'LENGTH)); --[63:56] (8 bits)
17  return_arg_i(RPARAM_M00_TIMEOUT)          <= timeout_i; --[55]
18  return_arg_i(RPARAM_REMAINING_TIME'RANGE) <= time_remaining_i; --[15:0] (16 bits)
19  source_i <= I_KERNEL_CALL.parameters(PARAM_MBUS_SOURCE'RANGE); --[55:32] (32 bits)
20  mbus_address_i <= std_logic_vector(unsigned(I_SYSRAM_BASE_ADDR) + unsigned(
      std_logic_vector(to_unsigned(to_integer(unsigned(I_KERNEL_CALL.parameters(
      PARAM_MBUS_OFFSET'RANGE))),mbus_address_i'LENGTH)))); --[23:0] (24b padded to 32)
  mbus_wlast_i  <= CS_I; -- declare as last write
  burst_len_i   <= 1; -- force burst length to one
  event_i       <= I_M00_AXI_BVALID; -- setup event to monitor as BVALID
  timeout_val_i <= std_logic_vector(to_unsigned(C_MBUS_TIMEOUT_VALUE,timeout_val_i'LENGTH));
 -------------------------------------------------------------------------------------
  when SYSCALL_MBUS_READ_BURST =>
   return_arg_i(RPARAM_MBUS_BURST_LEN'RANGE) <= I_KERNEL_CALL.parameters(
       PARAM_MBUS_BURST_LEN'RANGE); --[63:56] (8 bits)
   return_arg_i(RPARAM_MBUS_SOURCE'RANGE) <= I_M00_AXI_rdata; --[55:24] (32 bits)
   burst_len_i<= to_integer(unsigned(I_KERNEL_CALL.parameters(PARAM_MBUS_BURST_LEN'RANGE)));
        --[63:56] (8b)
   mbus_address_i <= std_logic_vector(unsigned(I_SYSRAM_BASE_ADDR) + unsigned(
       std_logic_vector(to_unsigned(to_integer(unsigned(I_KERNEL_CALL.parameters(
       PARAM_MBUS_OFFSET'RANGE))),mbus_address_i'LENGTH)))); --[23:0] (24b padded to 32)
   enable_index_i <= CS_I; -- enable index manager
 -------------------------------------------------------------------------------------
 when SYSCALL_MBUS_WRITE_BURST =>
  return_arg_i(RPARAM_MBUS_BURST_LEN'RANGE) <= I_KERNEL_CALL.parameters(PARAM_MBUS_BURST_LEN
      'RANGE); --[63:56] (8b)
  return_arg_i(RPARAM_M00_TIMEOUT)          <= timeout_i; --[55]
  return_arg_i(RPARAM_REMAINING_TIME'RANGE) <= time_remaining_i; --[15:0] (16 bits)
  burst_len_i <= to_integer(unsigned(I_KERNEL_CALL.parameters(PARAM_MBUS_BURST_LEN'RANGE)));
       --[63:56] (8b)
  source_i <= I_KERNEL_CALL.parameters(PARAM_MBUS_SOURCE'RANGE); --[55:24] (32 bits)
  mbus_address_i <= std_logic_vector(unsigned(I_SYSRAM_BASE_ADDR) + unsigned(
      std_logic_vector(to_unsigned(to_integer(unsigned(I_KERNEL_CALL.parameters(
      PARAM_MBUS_OFFSET'RANGE))),mbus_address_i'LENGTH)))); --[23:0] (24b padded to 32)
  enable_index_i <= CS_I; -- enable index manager
  event_i       <= I_M00_AXI_BVALID; -- setup event to monitor as BVALID
  timeout_val_i <= std_logic_vector(to_unsigned(C_MBUS_BURST_TIMEOUT_VALUE,timeout_val_i'
      LENGTH));
 -------------------------------------------------------------------------------------
```

The system calls related to the hardware mutex, section 4.5.1, behave similarly in terms of the System-Level Datapath. This means that they have as parameters the owner ID that intends to perform an action on the mutex, and a bit to select which one of the two mutexes is the target, i.e., if one intends to lock or unlock the system mutex or the local mutex, for example. Note again that the local mutex protects the LRAM against race conditions, and that the system mutex protects the host memory/SYSRAM against race conditions, likewise. Since the inner workings of the hardware mutex entity function around writing the owner ID to the mutex data register for every operation, in terms of the SLD nothing really changes. This happens because the mutex changes its current state, e.g., from locked to unlocked or from unlocked to locked, by always writing the owner ID to the aforementioned register. The mutex selector bit is fed onto the mutex decoder module, to select either one of the two mutexes, more on that later in section 4.4.3. The mutex syscalls return the data present in the status register of the mutex, which gives information on the mutex state, i.e., locked or unlocked, the mutex channel which performed the lock, and the mutex owner's ID. These fields for parameters and returns can be seen in fig. 4.2.17.

As it is possible to see in listing 4.11, the targeted mutex is selected with the bit 54 of the kernel call parameters, lines 12, 18, and 24, the mutex chip select is enabled, lines 13, 19, and 25, and the owner

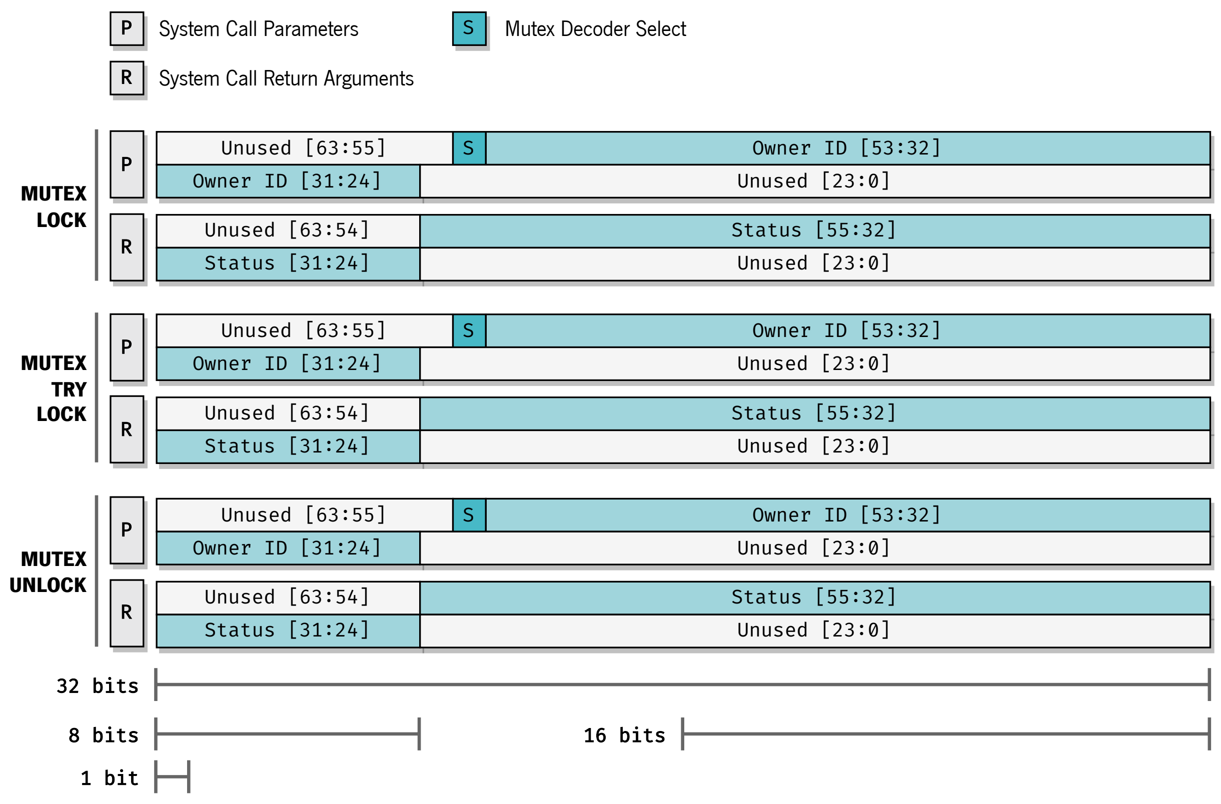


Figure 4.2.17: System-Level Datapath: Mutex Syscalls Parameters and Returns.

ID is assigned to the internal signal *mutex_owner_id_i*, from bits 53 to 24 of parameters, according to fig. 4.2.17. These syscalls then return the status of the mutex, from bits 55 to 24, which corresponds to the word present in the status register of the latter, lines 10, 16, and 22.

Listing 4.11: System-Level Datapath: Mutex Syscalls Parameters and Returns (HDL).

```
1  PARAM_SERVICE_SELECT_M0 : process (I_KERNEL_CALL.syscall_id,I_KERNEL_CALL.parameters,CS_I,
       timeout_i,time_remaining_i,I_M00_AXI,I_LRAM_DOUT,I_MUTEX_STATUS)
2  ------------------------------------------------------------------------------------------
3  begin
4  --MORE
5  ------------------------------------------------------------------------------------------
6  case I_KERNEL_CALL.syscall_id is
7  --MORE
8  ------------------------------------------------------------------------------------------
9  when SYSCALL_MUTEX_LOCK =>
10   return_arg_i(RPARAM_MUTEX_STATUS'RANGE) <= I_MUTEX_STATUS; --[55:24] (32 bits)
11   mutex_owner_id_i <= std_logic_vector(to_unsigned(to_integer(unsigned(I_KERNEL_CALL.
         parameters(PARAM_MUTEX_OWNER_ID'RANGE))),mutex_owner_id_i'LENGTH)); --[53:24] (30
         bits) (padded to 32)
12   O_MUTEX_ADDR      <= I_KERNEL_CALL.parameters(PARAM_MUTEX_DEC_ADDR'RANGE); --[54] (1 bit)
13   O_MUTEX_CS        <= CS_I;
14 ------------------------------------------------------------------------------------------
15 when SYSCALL_MUTEX_TRY_LOCK =>
16   return_arg_i(RPARAM_MUTEX_STATUS'RANGE) <= I_MUTEX_STATUS; --[55:24] (32 bits)
17   mutex_owner_id_i <= std_logic_vector(to_unsigned(to_integer(unsigned(I_KERNEL_CALL.
         parameters(PARAM_MUTEX_OWNER_ID'RANGE))),mutex_owner_id_i'LENGTH)); --[53:24] (30
         bits) (padded to 32)
18   O_MUTEX_ADDR      <= I_KERNEL_CALL.parameters(PARAM_MUTEX_DEC_ADDR'RANGE); --[54] (1 bit)
19   O_MUTEX_CS        <= CS_I;
20 ------------------------------------------------------------------------------------------
```

```
when SYSCALL_MUTEX_UNLOCK =>
  return_arg_i(RPARAM_MUTEX_STATUS'RANGE) <= I_MUTEX_STATUS; --[55:24] (32 bits)
  mutex_owner_id_i <= std_logic_vector(to_unsigned(to_integer(unsigned(I_KERNEL_CALL.
      parameters(PARAM_MUTEX_OWNER_ID'RANGE))),mutex_owner_id_i'LENGTH)); --[53:24] (30
      bits) (padded to 32)
  O_MUTEX_ADDR     <= I_KERNEL_CALL.parameters(PARAM_MUTEX_DEC_ADDR'RANGE); --[54] (1 bit)
  O_MUTEX_CS       <= CS_I;
------------------------------------------------------------------------------------------
```

The system call for reading the local interrupt controller does not need intervention of the system level datapath, and, thus, it is called with no parameters, as seen in fig. 4.2.18. On the other hand, the write to same entity, which generates a PL-to-PS interrupt request, only needs as parameter the selected interrupt source. For returns, the read returns the current contents of the LINTC control and status registers, and the write as no return. As forementioned, and similarly to the LRAM, the LINTC entity could be included onto the local bus and one could have a dedicated decoding element that dictated the flow of data, in an approach like-minded to [62]. Although, as the number of elements has not that high, one opted to use one-to-one communication between the System-Level Datapath (SLD) and hardware resources like the LRAM and the LINTC, for example.

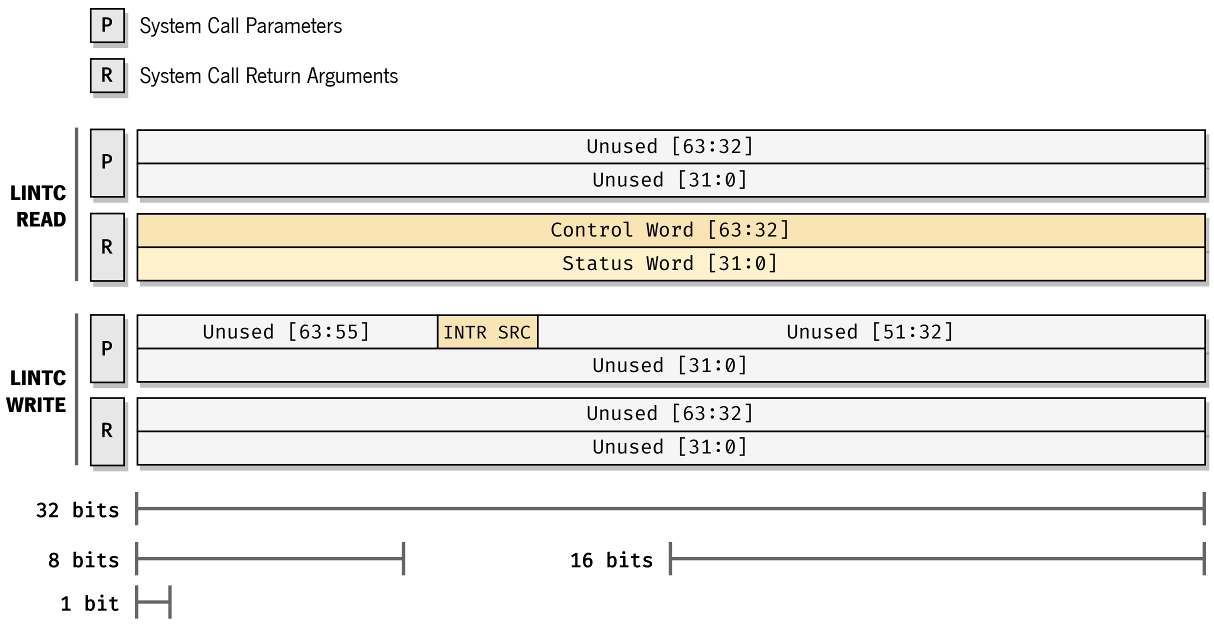


Figure 4.2.18: System-Level Datapath: Local Interrupt Controller Syscalls Parameters and Returns.

Listing 4.12 is the HDL representation of fig. 4.2.18. Thus, one can see the return of the control and status words from the LINTC entity in lines 10 and 11, and also the selection of the interrupt source from bits 54 to 52 of the kernel call parameters. For more information on the LINTC entity refer to section 4.5.2.

Listing 4.12: System-Level Datapath: LINTC Syscalls Parameters and Returns (HDL).

```
PARAM_SERVICE_SELECT_M0 : process (I_KERNEL_CALL.syscall_id,I_KERNEL_CALL.parameters,CS_I,
    timeout_i,time_remaining_i,I_M00_AXI,I_LRAM_DOUT,I_MUTEX_STATUS)
-------------------------------------------------------------------------------------------
begin
--MORE
-------------------------------------------------------------------------------------------
case I_KERNEL_CALL.syscall_id is
--MORE
-------------------------------------------------------------------------------------------
when SYSCALL_LINTC_READ =>
  return_arg_i(RPARAM_LINTC_CONTROL'RANGE) <= I_LINTC_CONTROL_W; --[63:32] (32 bits)
  return_arg_i(RPARAM_LINTC_STATUS'RANGE)  <= I_LINTC_STATUS_W;  --[31:0]  (32 bits)
-------------------------------------------------------------------------------------------
when SYSCALL_LINTC_WRITE =>
  O_LINTC_INTR_SELECT <= I_KERNEL_CALL.parameters(PARAM_LINTC_INTR_SELECT'RANGE); --[54:52]
-------------------------------------------------------------------------------------------
```

4.2.5.1 Event Manager

The microkernel's time management is dictated by the event manager unit. The approach used in the dissertation follows the one proposed in [62] but disregards the sleep feature and, thus, cannot wait for asynchronous event sources, i.e., cannot wait for an event with no specified timeout. Although, this is not something difficult to work around since one can always specify higher timeout values.

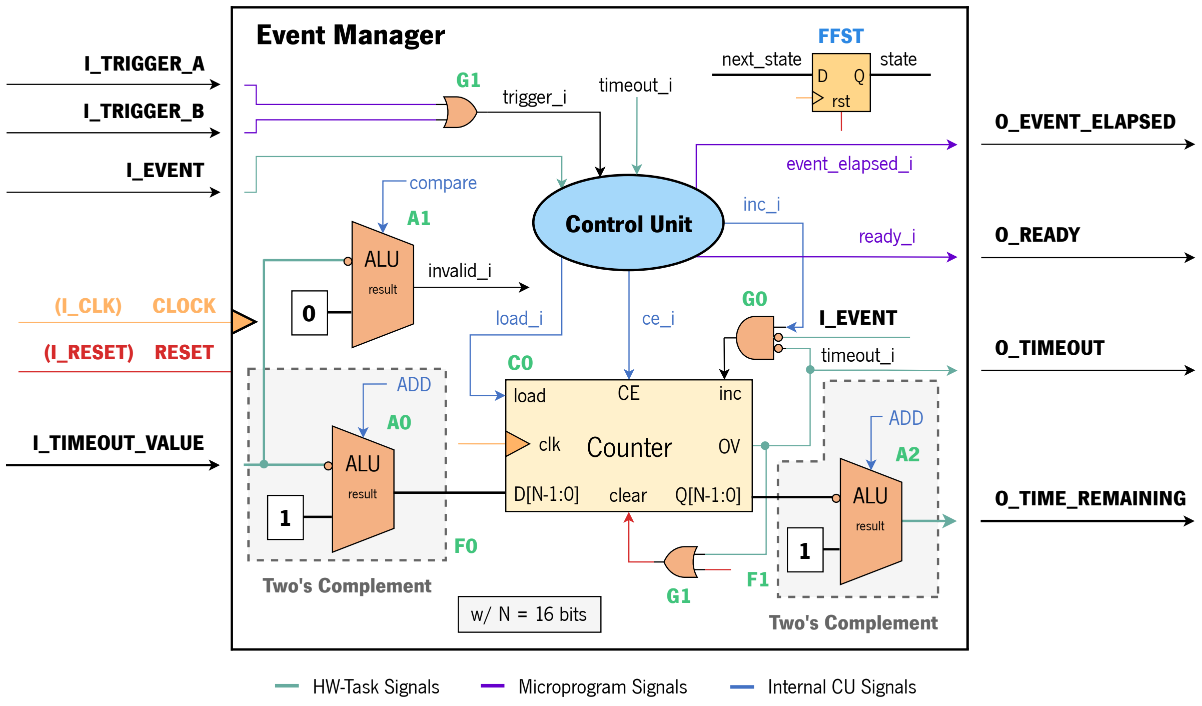


Figure 4.2.19: Event Manager RTL Design Internal Architecture, adapted [62].

The event manager is used in the *wait for event timeout* syscall and also in the *mbus write* syscalls for the waiting on the bus signals with a timeout, as explained earlier in section 4.2.5. For more information on the system calls refer to the table 4.1 of section 4.2.3.

The RTL representation of the event manager unit is depicted by fig. 4.2.19. The unit bases itself in an upwards counter C0, represented in yellow (🟨), that increments the given value specified by *i_timeout_value*. For the event manager to count from that value to zero, one uses the ALU A0, represented in orange (🟧), to turn the timeout value into a two's complement version by inverting all the bits and adding one. The current count number is then reverted to its original form to output the remaining time in A2. The counter only increments when the logical result on G0 is true, which means that the counter stops with an overflow and also when the specified event to monitor, given by the *i_event* port, is asserted. When these two conditions are false, the counter then increments by command of the module's control unit with the internal signal *inc_i*. In a similar way the counter's clock enable and load inputs are given by the control unit. When the counter overflows the *o_timeout* port is asserted to indicate that a timeout occurred while waiting on an event.

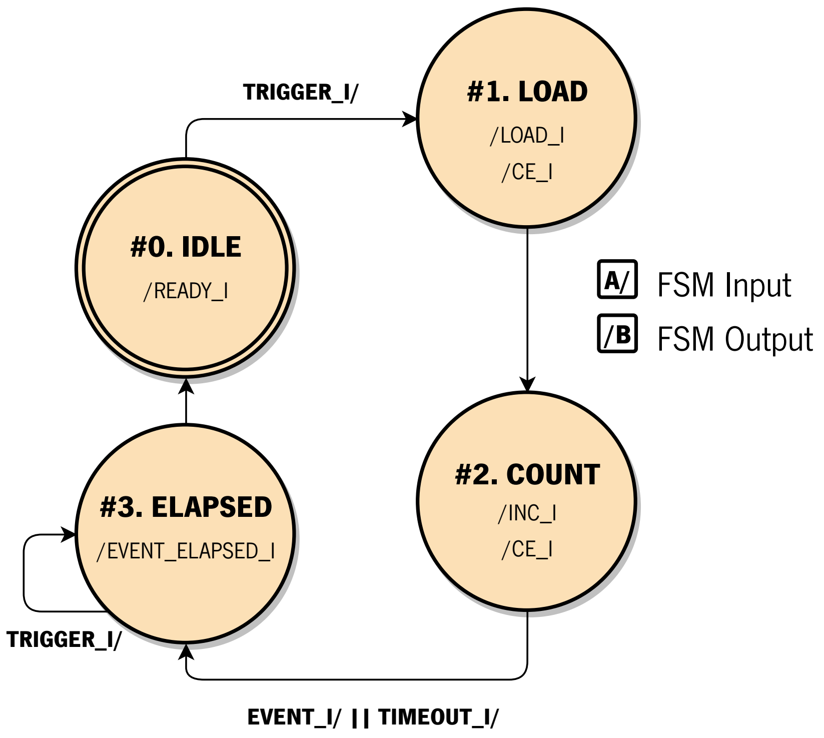


Figure 4.2.20: Event Manager Finite State Machine (FSM), adapted [62].

The event manager's FSM is represented in fig. 4.2.20. The latter starts in state #0, deemed as idle, and moves onto the next state in the next clock cycle if the trigger signal, *trigger_i*, is asserted. As it is possible to see in fig. 4.2.19, this signal is the logical combination of two trigger sources A and B. Most of the time, when waiting for events using the *wait for event timeout* syscall, the unit will use only one trigger source, since there is usually only one event to wait on. The addition of another trigger source was added, so the module could be used within the slave interface event manager to manage waits on AXI4 lite signals coming from the S00 Control and S01 Data interfaces. More on that later in section 4.5.4. Moreover, idle (#0) also asserts the *ready_i* signal to synchronize the beginning of the syscall with the microprogram. In the state #1, deemed as load, the *ce_i* and *load_i* signals are asserted to, respectively, enable the C0 counter (🟨), and also enable the load of the timeout value. Unconditionally, in the next clock cycle, the FSM moves onto the count state (#2) where it continues to enable the counter with the *ce_i* signal, but now asserts the *inc_i* signal to start the incrementing. The FSM proceeds to the elapsed state (#3) if the event that one is monitoring occurs or if there is a timeout of the counter. In the elapsed state (#3) the event elapsed is asserted to inform the microprogram, as depicted in purple in fig. 4.2.19. Refer to section 4.2.3 for more information on how the microprogram leverages these signals for the *wait for event timeout* syscall. Additionally, as in [62], the FSM is kept in the elapsed state until the *trigger_i* signal is not asserted anymore. When that happens the FSM naturally returns to the idle state in the next clock cycle. The sequence diagram of fig. B.11 depicts the interaction between the entities involved in a single syscall waiting procedure, *wait event elapsed*, called from the hardware task, i.e., using a procedure that only calls the *wait for event timeout* syscall.

4.2.5.2 Index Manager

As mentioned earlier in section 4.2.3, the index manager unit allows for the writing and reading to consecutive buffer positions in the hardware task entity. Due to this, the unit is related with the LBUS and MBUS syscalls. To deal with this, the index manager entity has a counter C0, represented in yellow (🟨), as the main logic element, which increments the current index, given by *o_index*. The module also implements the same index but delayed one clock cycle (FF0) (🟨), *o_index_d1*, since in the LBUS syscalls the LRAM has a one clock cycle latency to output its data on *dout*, and because depending on the syscall it might be needed to read and write to the same buffer simultaneously. For this, having two indexes allows one a similar scheme to the one existing in software, for example in kernel circular buffers, with a head and a tail index. With the head representing the point in which the producer inserts elements into the buffer, and the tail representing the point in which the consumer retrieves elements from the buffer.

The counter C0 (🟨) found in fig. 4.2.21 is the same exact logic element already explained in section 4.2.4, with load and increment functionalities. As it possible to observe, the counter's *load* input is fed with the logical *and* between the enable of the index service in the SLD, *i_syscall_en_index*, the

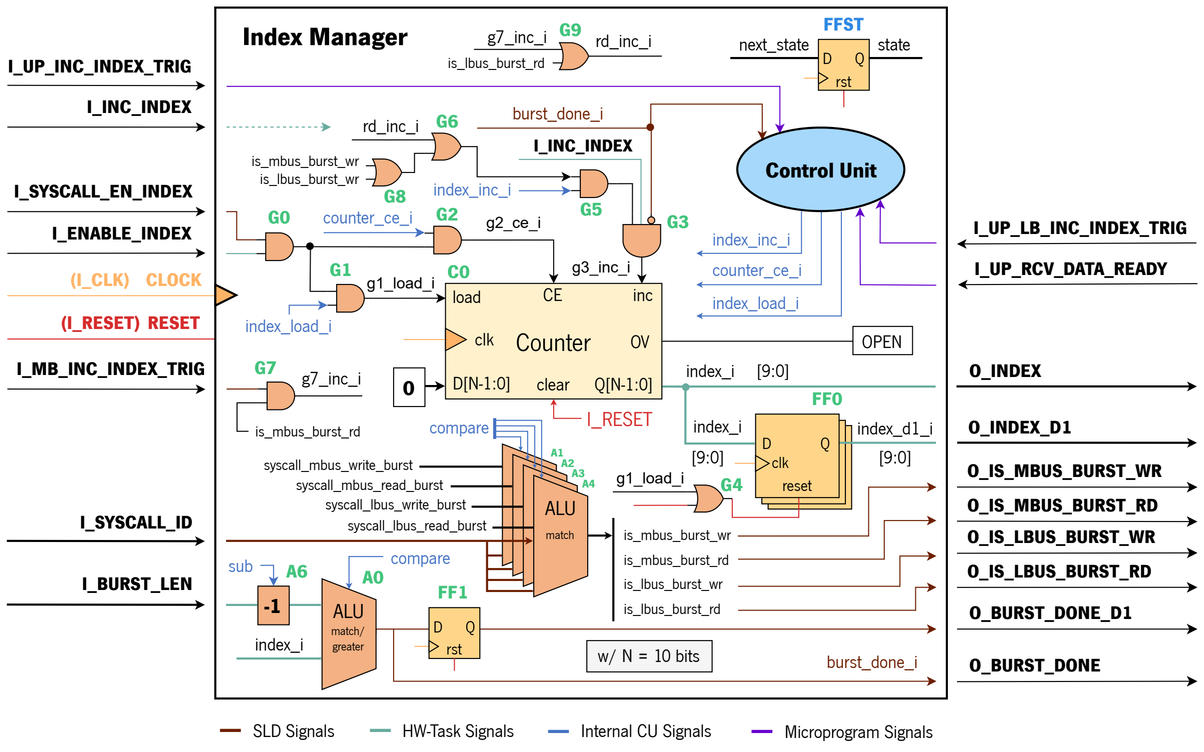


Figure 4.2.21: Index Manager RTL Design Internal Architecture.

enable coming from the hardware task, *i_enable_index*, and also the control unit signal *index_load_i* (G0 and G1). The counter is enabled with the logical *and* of the output of G0 with the control unit signal *counter_ce_i*. The increment of the counter is activated whenever one has the *i_inc_index* signal asserted in the hardware task (entering G3), and the MBUS trigger coming from the SLD (entering G7) paired with one of the MBUS or LBUS syscalls (G6, G7, G8, and G9) and also the control unit signal *index_inc_i* (entering G5). Additionally, the increment stops with the assertion of *burst_done_i* (entering G3), i.e., when the burst syscall is finished. The *burst_done_i* signal is generated in A0, represented in orange (🟧), with the comparison of the current index, *index_i*, with the burst length, *i_burst_len*, minus one. This happens since, for example, a burst of ten elements should stop at index nine, and for that reason the *o_burst_done* is also used to generate the MBUS and LBUS *wlast*, while the one-cycle delayed version, *o_burst_done_d1*, is used to generate the LBUS *rlast*. This can be seen in fig. 4.2.11, alongside with the trigger and data ready inputs of the index manager unit. Moreover, the module also does some comparison regarding the system call ID to determine the current burst syscall (from A1 to A4). The result is used internally to the index manager but also in the address manager unit, section 4.2.5.3.

The index manager's FSM is represented in fig. 4.2.22. The latter starts in the state #0, which represents the *load* of the counter with index zero, fig. 4.2.21. Thus, the state #0 asserts the *index_load_i*

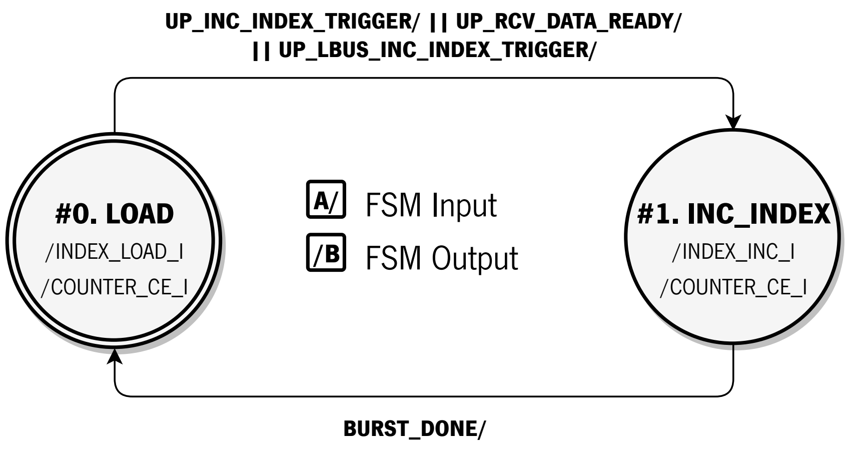


Figure 4.2.22: Index Manager Finite State Machine (FSM).

signal, and enables the counter with the *counter_ce_i* signal. If the other signals from the SLD and the hardware task are asserted, the C0 counter will perform the initial load, and the FSM will move on onto state #1, *inc_index*, if the condition of fig. 4.2.22 is true. This means that the one move to the increment state if the one of the microprogram triggers is asserted. Upon reaching the *inc_index* state the FSM asserts the *index_inc_i* and *counter_ce_i* signals, to perform the counting and enable the counter element, respectively. The FSM is in the state #1 until the burst finished, returning to *load* in the next clock cycle. The sequence diagrams of fig. B.12 and fig. B.13 depict the interaction between the entities involved in a single syscall procedure that writes/reads in an unsafe manner, i.e., without mutex protection, to the SYSRAM. The call is made from the hardware task entity, using a procedure that only calls the *mbus write* or *mbus read* syscalls in single-word or burst format.

4.2.5.3 Address Manager

The address manager unit functions similarly to the index manager but with the purpose of incrementing the address of the LRAM in LBUS burst syscalls.

Regarding this, the latter also has a counter with load (C0), represented in yellow, that initially loads a specific address, *i_start_addr*, and starts to increment from that address until the local memory burst is finished. In fig. 4.2.23 it is possible to see that the C0 counter is enabled by the logical *and* of *i_en_addr_counter*, coming from the SLD, and the control unit signal *ce_i* (entering G0). The counter performs a load whenever there is a LBUS system call (G1, G4, and G5), and the *load_i* control unit signal is asserted or at the end of each burst (G3). Moreover, the counter increments while one is performing a burst LBUS syscall with the control unit signal *inc_i* asserted. The counter stops when it reaches

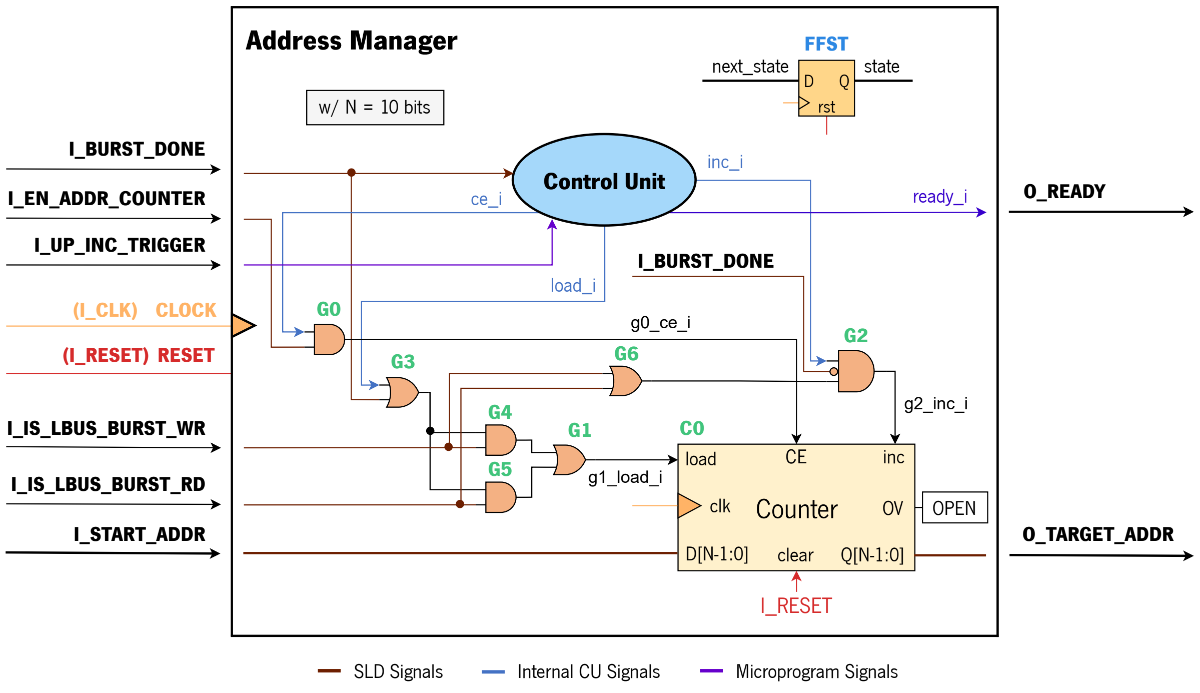


Figure 4.2.23: Address Manager RTL Design Internal Architecture.

the end of the burst, given by the logical *not* of *i_burst_done* in G2. The unit receives a trigger signal, *i_up_inc_trigger*, and outputs a ready signal, *o_ready*, in order to achieve synchronism with the microprogram. For more information on the LBUS syscalls refer to section 4.2.3. Finally, the *Q* output of the counter gives the target address, *o_target_addr*, to be fed to the LRAM.

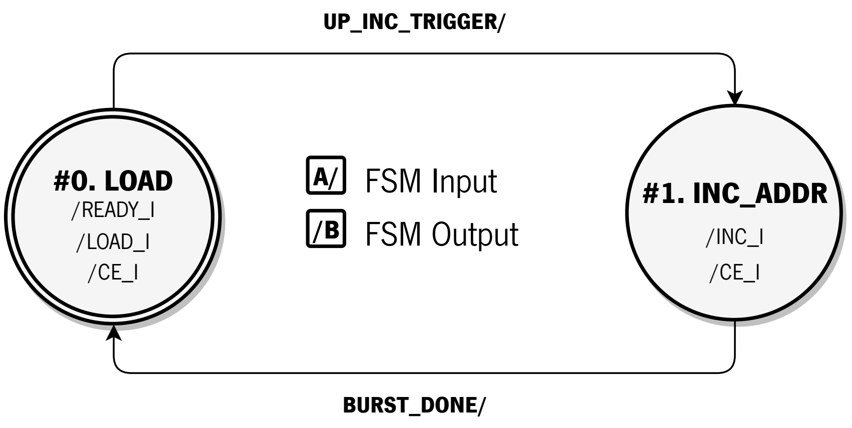


Figure 4.2.24: Address Manager Finite State Machine (FSM).

The address manager FSM is represented in fig. 4.2.24. As it is possible to see it only has two states and starts off in state #0, deemed as *load*. In the latter the FSM asserts the *ready_i* signal to inform the microprogram, and performs the load of the initial counter value with the assertion of the *load_i* and *ce_i* signals. The FSM then waits for the trigger of the microprogram to go onto the next state. In state #1, deemed as *inc_addr*, the FSM asserts the *inc_i* and maintains *ce_i* asserted to perform the increment of the address. As seen before in the index manager FSM, fig. 4.2.22, the address manager FSM also returns to the *load* state when the LRAM burst is finished, with *burst_done* asserted. The sequence diagrams of fig. B.14 and fig. B.15 depict the interaction between the entities involved in a single syscall procedure that writes/reads in an unsafe manner, i.e., without mutex protection, to the LRAM. The call is made from the hardware task entity, using a procedure that only calls the *lbus write* or *lbus read* syscalls in single-word or burst format.

4.3 Hardware Task

In HAL-ASOS the hardware task entity has a software application representative, as aforementioned in fig. 3.1.4, which extends hardware operation by executing system calls that cannot be transposed to programmable logic. This is the case of system calls related with descriptors, e.g., opening or closing a binary file, socket, or pipe. The hardware task implemented follows a very similar structure to the one existing in HAL-ASOS but it communicates with the host system via a new memory mechanism that makes use of the AXI master interface (*c.f* section 5.1). Additionally, the hardware task of fig. 4.3.1 also differs in the kernel call and response signals to execute the system calls because of module refactoring and the addition of certain functionally. Altogether, this increased the number of signals in the S00 Kernel and M00 Task interfaces. Moreover, it also does not exhibit a sleep feature regarding the state flip-flop FFST, as it was removed from the kernel. As it is possible to see in fig. 4.3.1, the hardware task behaves like most architectures with a control unit, represented in blue (🟦), and a datapath represented in orange (🟧), where the control unit asserts signals that activate certain datapath elements, e.g., the *run* signal activating the black box IP, and reacts from signals from the datapath to consider state switching, e.g., the *done* signal coming from the black box IP as well.
The hardware task's FSM is composed by a series of states dependent on the number of extra features needed, i.e., features that need Linux help to be carried out, and some other features that allow for the activation of microkernel services, like timing events or execute memory operations, represented in green (🟩).

As previously stated in section 4.2.2, the hardware task's control unit reacts to certain microkernel signals to start operation, *task_run*, or restart it, *task_reset*, and informs the microkernel when the processing is done with the *task_done* signal. This is possible to see at the top of fig. 4.3.1. While the

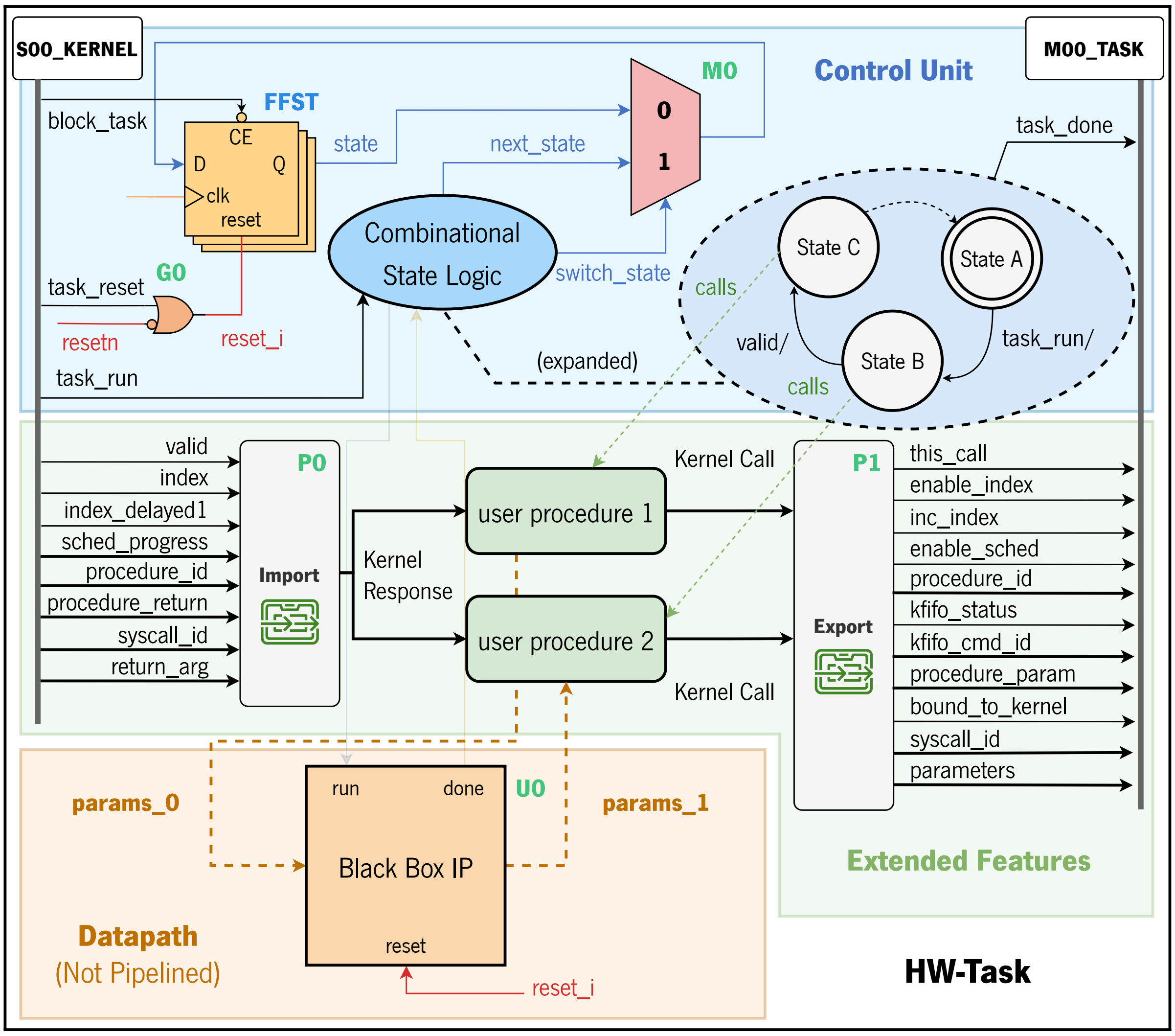


Figure 4.3.1: Hardware Task: Simplified Example Architecture, adapted [62].

task_reset signal clears the state flip-flop, labelled as FFST, the inverted *block_task* signal functions as *ce* for it. This way, the hw-task's state remains the same whilst the microkernel is executing a syscall. One major difference of a typical control unit and datapath architecture is that, in this case, the hardware task can make a user procedure call per state, which consequently dictates the active signals for the kernel call (P1). In a similar way, the results of syscall operation are reflected by the kernel through the signals of the kernel response (P2). These extended features will be explained in more detail in section 5.1. Usually, the parameters and returns of these extended procedures in each state are fed and retrieved directly onto/from the black box datapath elements (U0). Here this can be observed with the *params_0* and *params_1* vectors. Additionally, as laid out by HAL-ASOS, the hw-task's datapath, for most of the cases, represents the algorithm to be offloaded, e.g., a FAST algorithm [62]. The latter can then be implemented sequentially, as it is represented in fig. 4.3.1, or exploit Instruction Level Parallelism (ILP) by the use of various black box IPs in a pipelining configuration, as implemented in [62], if the need arises to achieve timing closure.

4.4 Resource Management

The resource management of the microkernel is done with the decoder entities present in fig. 4.1.1. Namely, the slave decoder, the page decoder, and the mutex decoder. As in [62], there are three main areas where resource management is applied: (1) the area of the S00 Control interface, where the AXI address must be properly decoded/translated into a kernel register, (2) the area of the S01 Data interface, where the address must be forwarded to the local memory, and, lastly, in the selection of the active mutex when using mutex-related system calls.

The most straightforward mapping happens in the S01 Data interface. This occurs since the local memory is coded in way that infers resources of the project's board after synthesis. Thus, the LRAM already has address decoding as it infers a true dual port block RAM. The forwarding of the address from the S01 Interface to the local memory only shrinks the interface's output address to fit the address size of the local memory, e.g., in this case, sixteen bits represent the address space of the AXI4 lite interfaces since they have 64K of size, which means that only the nine least significant bits of the address are fed onto the LRAM's address A port because the depth of the peripheral is stipulated at 1K of size, *c.f* fig. 4.6.3.

The mapping of the S00 Control interface is the most complex since it uses two-level address decoding as stipulated by HAL-ASOS. For this, it makes use of two combinational entities: the slave decoder, section 4.4.1, and the page decoder, section 4.4.2. The combination of these decoders allows for the organization of the S00 Control address space into pages. For more information about the actual register mapping of the pages refer to section 4.6.3. This type of organization follows a similar scheme to virtual addresses on Linux. So, it decomposes an address into multiple parts that, altogether, specify the physical location of a certain register by identifying the page and its word offset. The S00 addressing considering a twelve bit address is represented in fig. 4.4.1. In this case, the address' bits 7 and 6 specify the page of the register, e.g., "01" defines page one, and the bits 5 to 2 specify the word offset of the register, e.g., "0011" defines word three. As it is possible to perceive the amount of bits that specifies a certain page is two because the microkernel has four pages (from "00" to "11"), and the amount of bits for the word offset is four because each page can be mapped up to sixteen registers (from "0000" to "1111"). A detailed overview of the two-level address decoding of the S00 interface can be found in fig. 4.6.15.

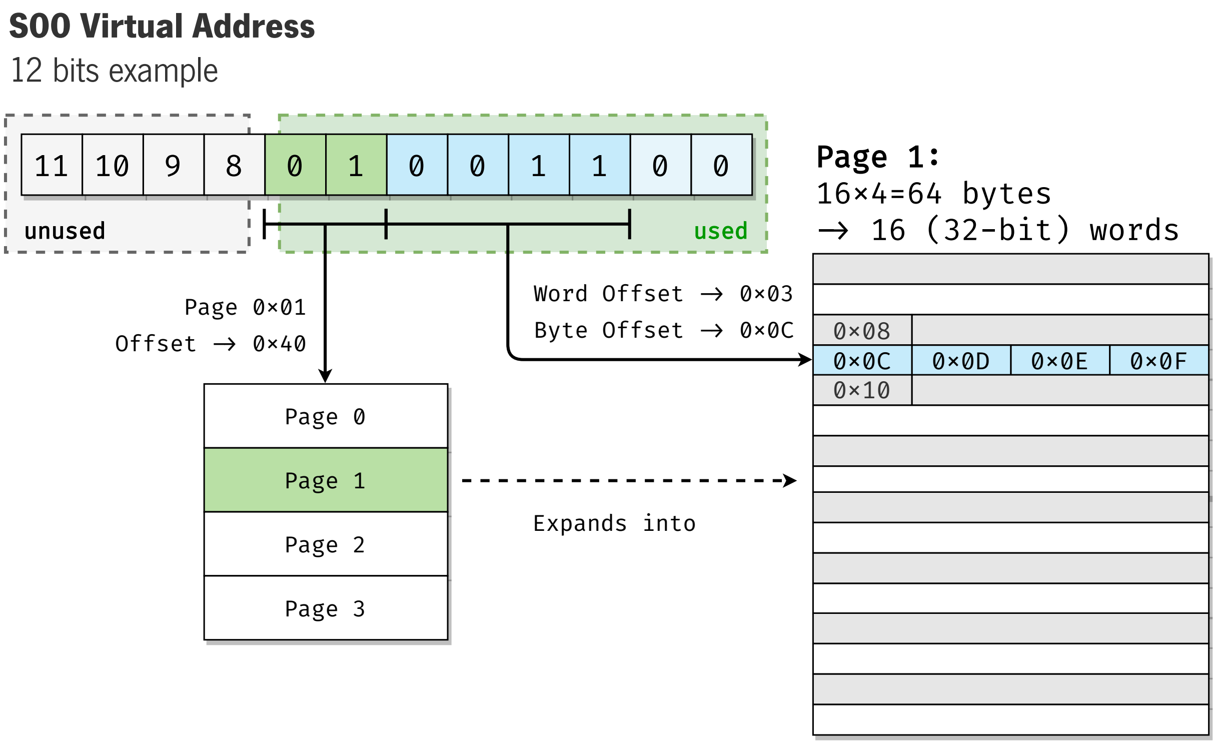


Figure 4.4.1: Virtual Register Addressing.

The third addressing area is the one concerning the mutexes. This area performs resource addressing by utilizing the mutex decoder combinational entity, section 4.4.3. The mutex resources are also mapped onto the S00 Control interface because of their dual port nature, fig. 4.4.2, having a specific region on the interface's address space, *c.f* section 4.6.3. Although, here one intends to portray the resource management done regarding the B ports of both mutexes, i.e., how the microkernel handles mutex selection when a mutex-related syscall is called, e.g., mutex lock, try-lock or unlock (*c.f* section 4.2.3). Regarding this, the mutex decoder helps in that sense since it establishes the currently active mutex, and consequently dictates which mutex data is used and forwarded from the SLD to the mutex and vice-versa. Further information on the mutex decoder is provided in section 4.4.3.

4.4.1 Slave Decoder

The slave decoder is a fully combinational entity of the microkernel that divides the S00 Control AXI4 lite slave interface into multiple pages, hence the name. The unit was introduced in [62] as a way to save target resources by multiplexing output data to various units in the same page and demultiplexing input data coming in vector format from the same units. The RTL design of the slave decoder element is represented in fig. 4.4.3. As it is possible to see, its primary logic element is the LUT L0, which takes some

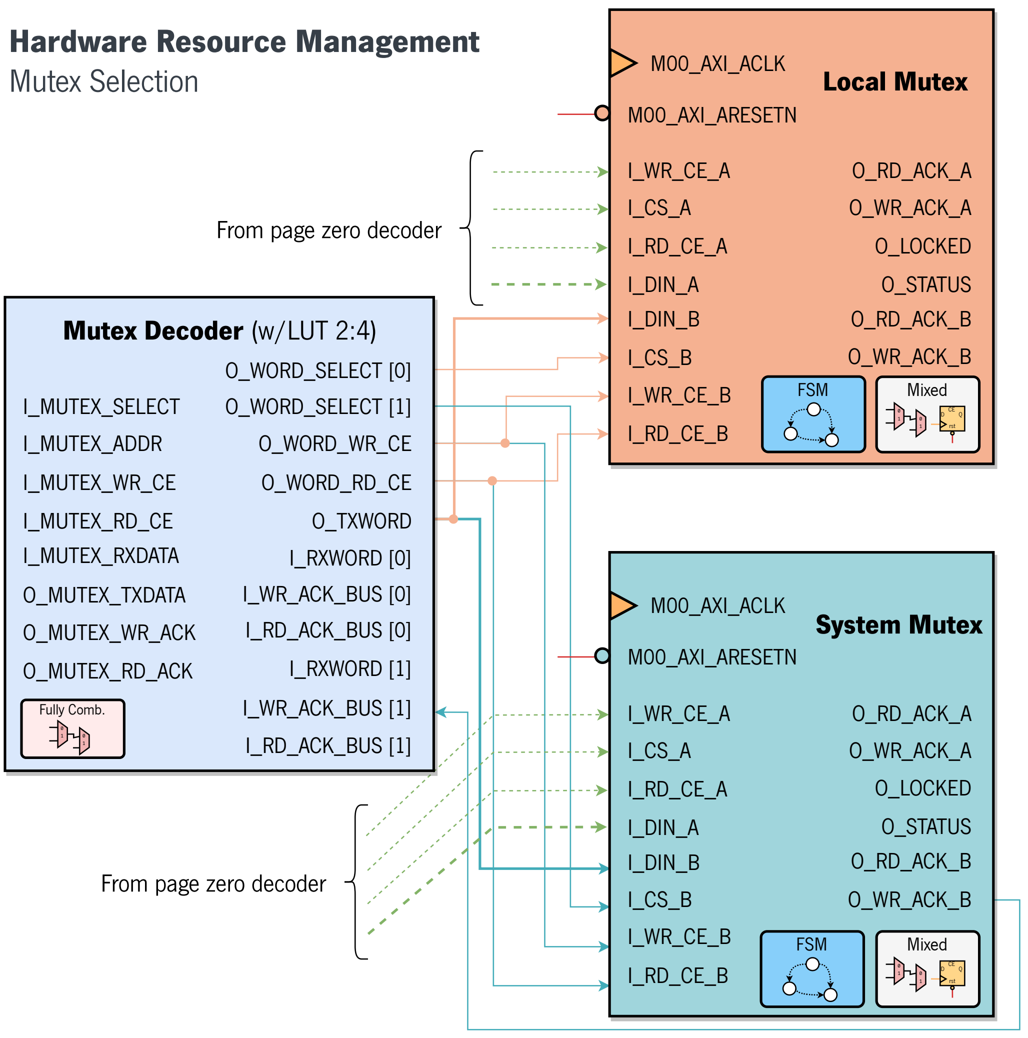


Figure 4.4.2: Mutex Resource Management Overview.

bits of the input address and uses them to specify the active pages. These select bits for the pages are concatenated into a *page_select* array with size equal to the number of pages. The example of fig. 4.4.3 is made for a generic scenario with *N* kernel pages and *W* words in each page, considering a 32-bit machine word. The aforementioned LUT is enabled by the chip select provided by the AXI4 lite interface (S00 Control), *c.f* fig. 4.6.15. The data received by the slave decoder, *rxdata*, and the *wr_ce* and *rd_ce* signals are forwarded to the page decoder entity, *c.f* section 4.4.2, to ultimately reach the slave modules. From a post-synthesis standpoint this is possible to do if one does not surpass the fan-out threshold, which explains why no decoding logic is used in these signals. Regarding data coming from multiple slaves, e.g., received data, *pagen_rxdata*, read acknowledge, *page_rd_ack*, and write acknowledge, *page_wr_ack*, there is decoding logic to generate only one word of data, *txdata*, one write acknowledge, *wr_ack*, and one read acknowledge, *rd_ack*. The slave data is demultiplexed in M0, according to the same bits that identify the currently active page, while the acknowledges are generated by an unary *OR* of all the of the corresponding page acknowledges, i.e., a logical *OR* of with all the write or read acknowledges (UG0 and UG1). The ports marked as output, e.g., *O_**, are directly connected to inputs of the AXI4 lite interface, *c.f* fig. 4.6.15 and fig. 4.6.1. The HDL representation of the slave decoder is in listing 4.13. Future implementations of the slave decoder element could improve the code scalability by using HDL *generate* statements.

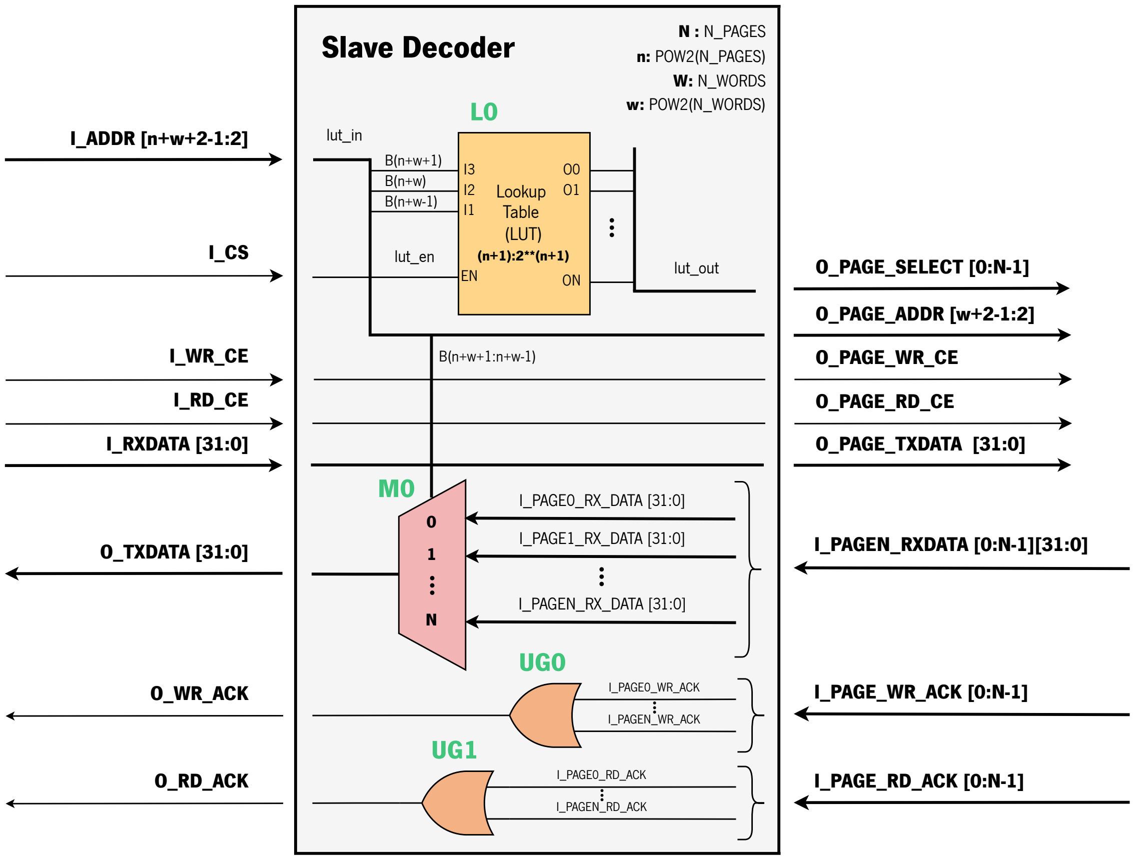


Figure 4.4.3: Slave Decoder RTL Design Internal Architecture, adapted [62].

Listing 4.13: Slave Decoder (HDL).

```vhdl
-- Direct Connections
O_PAGE_ADDR    <= I_ADDR(POW2(N_WORDS)+2-1 downto 2);
O_PAGE_WR_CE   <= I_WR_CE;
O_PAGE_RD_CE   <= I_RD_CE;
O_PAGE_TXDATA <= I_RXDATA;

-- Other Connections
O_WR_ACK <= unary_page_or(I_PAGE_WR_ACK);
O_RD_ACK <= unary_page_or(I_PAGE_RD_ACK);

-- Stage 1 address decoding
O_PAGE_SELECT <= lut_out;
lut_en <= I_CS;
---------------------------------------------------------------------------------------------
ADDR_DECODE_STAGE1_L0 : process (lut_en,I_ADDR,lut_in)
---------------------------------------------------------------------------------------------
begin
    lut_in  <= I_ADDR(POW2(N_PAGES)+POW2(N_WORDS)+2-1 downto POW2(N_WORDS)+2);
    lut_out <= (others => '0');
        case to_integer(unsigned(lut_in)) is
            when W_PAGE0_OFFSET => lut_out <= (C_PAGE0_BIT => lut_en, others => '0');
            when W_PAGE1_OFFSET => lut_out <= (C_PAGE1_BIT => lut_en, others => '0');
            when W_PAGE2_OFFSET => lut_out <= (C_PAGE2_BIT => lut_en, others => '0');
            when W_PAGE3_OFFSET => lut_out <= (C_PAGE3_BIT => lut_en, others => '0');
            when others         => null;
        end case;
end process ADDR_DECODE_STAGE1_L0;
-------------------------------------------------------------------------------------------

-- Slave RX data demultiplexing
-------------------------------------------------------------------------------------------
RX_DATA_DEMUX_M0 : process (I_ADDR, I_PAGEN_RXDATA)
-------------------------------------------------------------------------------------------
   variable select_M0 : integer range 0 TO (N_PAGES-1);
begin
    select_M0 :=  to_integer(unsigned(I_ADDR(POW2(N_PAGES)+POW2(N_WORDS)+2-1 downto POW2(
        N_WORDS)+2)));
    O_TXDATA <= (others => '0');
    case select_M0 is
        when W_PAGE0_OFFSET => O_TXDATA <= I_PAGEN_RXDATA(C_PAGE0_BIT);
        when W_PAGE1_OFFSET => O_TXDATA <= I_PAGEN_RXDATA(C_PAGE1_BIT);
        when W_PAGE2_OFFSET => O_TXDATA <= I_PAGEN_RXDATA(C_PAGE2_BIT);
        when W_PAGE3_OFFSET => O_TXDATA <= I_PAGEN_RXDATA(C_PAGE3_BIT);
        when others         => null;
    end case;
end process RX_DATA_DEMUX_M0;
-------------------------------------------------------------------------------------------
```

4.4.2 Page Decoder

The page decoder element, along the lines of the slave decoder, is also fully combinational and divides a certain microkernel page into multiple words. Each one of these words then represents the data stored in kernel registers with fixed positions in the S00 Control address space, *c.f* section 4.6.3. The page decoder dictates the flow of data within a certain page, specifying the currently-active register word at any given time. For this, its design also is based around a LUT element that specifies the active words, *word_select*, activated by the page select signal, *page_select*, coming from the slave decoder. It is important to note that every page has a slave decoder with direct connection to its registers, *c.f* fig. 4.6.15. These registers are usually mapped in a level lower than the microkernel's top-level inside specialized hardware resources, e.g., mutex, LINTC, but can also trace back to a certain register mapped in the top-level associated with unmapped register words, *c.f* fig. B.10. For simplification of the slave interfaces' address spaces consider that all unmapped registers words return "1facefff", i.e., the unmapped-interface status. The rest of the page decoder's design complies with what was established for the slave decoder, but to feed multiple registers, and decode their output data and acknowledges, instead of page-referrent data. The example of fig. 4.4.4, as fig. 4.4.3, also refers to a kernel containing *N* pages each one with *W* words. The HDL representation of the page decoder is in listing 4.14. As forementioned for the slave decoder, future implementations of the page decoder could also focus on improving code scalability by using HDL *generate* statements.

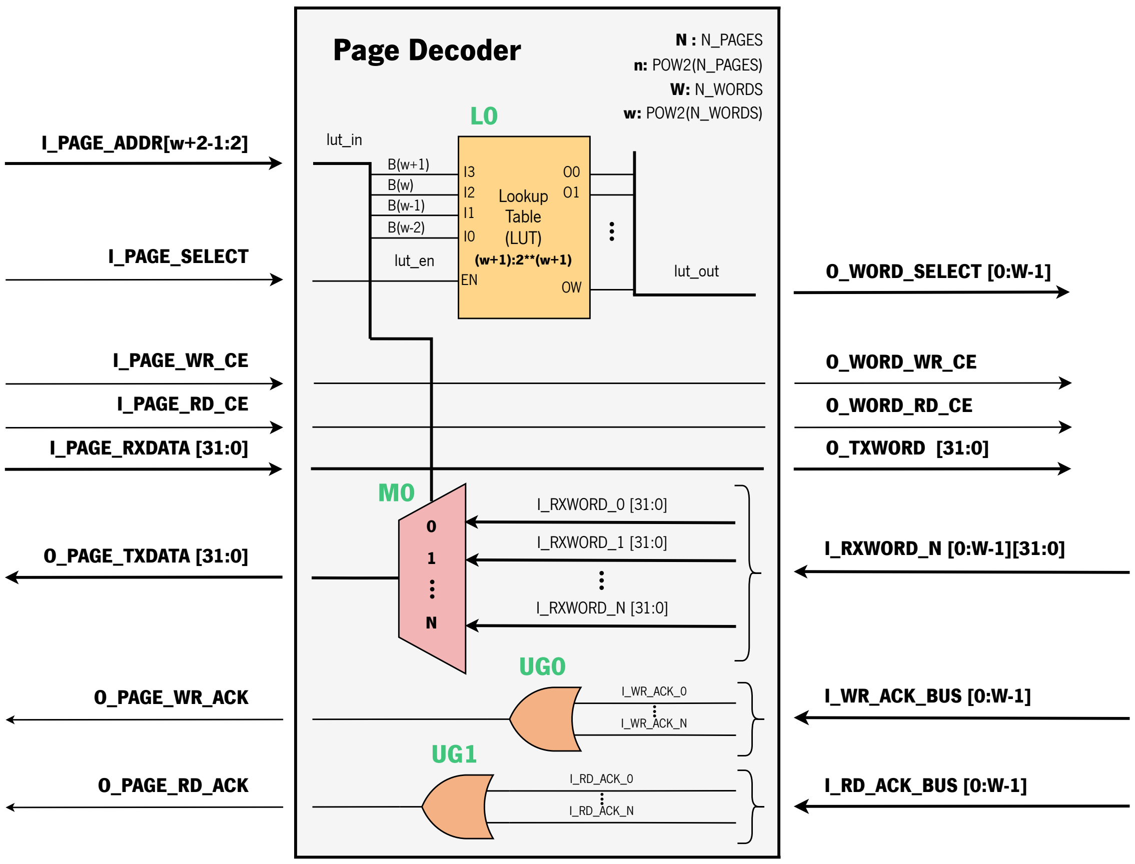


Figure 4.4.4: Page Decoder RTL Design Internal Architecture, adapted [62].

Listing 4.14: Page Decoder (HDL).

```
-- Direct Connections
O_WORD_WR_CE <= I_PAGE_WR_CE;
O_WORD_RD_CE <= I_PAGE_RD_CE;
O_TXWORD <= I_PAGE_RXDATA;

-- Other Connections
O_PAGE_WR_ACK <= unary_word_or(I_WR_ACK_BUS);
O_PAGE_RD_ACK <= unary_word_or(I_RD_ACK_BUS);

-- Stage 2 address decoding
O_WORD_SELECT <= lut_out;
lut_en <= I_PAGE_SELECT;
---------------------------------------------------------------------------------------------
ADDR_DECODE_STAGE2_L0 : process (lut_en,I_PAGE_ADDR,lut_in)
---------------------------------------------------------------------------------------------
   variable select_L0 : integer range 0 TO (N_WORDS-1);
begin
    lut_in <= I_PAGE_ADDR(POW2(N_WORDS)+2-1 downto 2);
    select_L0 := to_integer(unsigned(lut_in));
    lut_out <= (others => '0');
        case select_L0 is
            when W_WORD0_OFFSET => lut_out <= (C_WORD0_BIT => lut_en, others => '0');
            -- MORE (From word 1 to E)
            when W_WORDF_OFFSET => lut_out <= (C_WORDF_BIT => lut_en, others => '0');
            when others         => null;
        end case;
end process ADDR_DECODE_STAGE2_L0;
```

```
--------------------------------------------------------------------------------

-- RX word demultiplexing
--------------------------------------------------------------------------------
RX_WORD_DEMUX_M0 : process (I_PAGE_ADDR, I_RXWORD_N)
--------------------------------------------------------------------------------
   variable select_M0 : integer range 0 TO (N_WORDS-1);
begin
    select_M0 :=  to_integer(unsigned(I_PAGE_ADDR(POW2(N_WORDS)+2-1 downto 2)));
    O_PAGE_TXDATA <= (others => '0');
    case select_M0 is
        when W_WORD0_OFFSET => O_PAGE_TXDATA <= I_RXWORD_N(C_WORD0_BIT);
        -- MORE (From word 1 to E)
        when W_WORDF_OFFSET => O_PAGE_TXDATA <= I_RXWORD_N(C_WORDF_BIT);
        when others         => null;
    end case;
end process RX_WORD_DEMUX_M0;
--------------------------------------------------------------------------------
```

4.4.3 Mutex Decoder

The mutex decoder follows the same architecture stipulated for the page decoder, section 4.4.2, but assuming a single-bit address. This happens because, in this case, one only has two mutex entities to select from, the local mutex and the system mutex, *c.f* fig. 4.4.2.

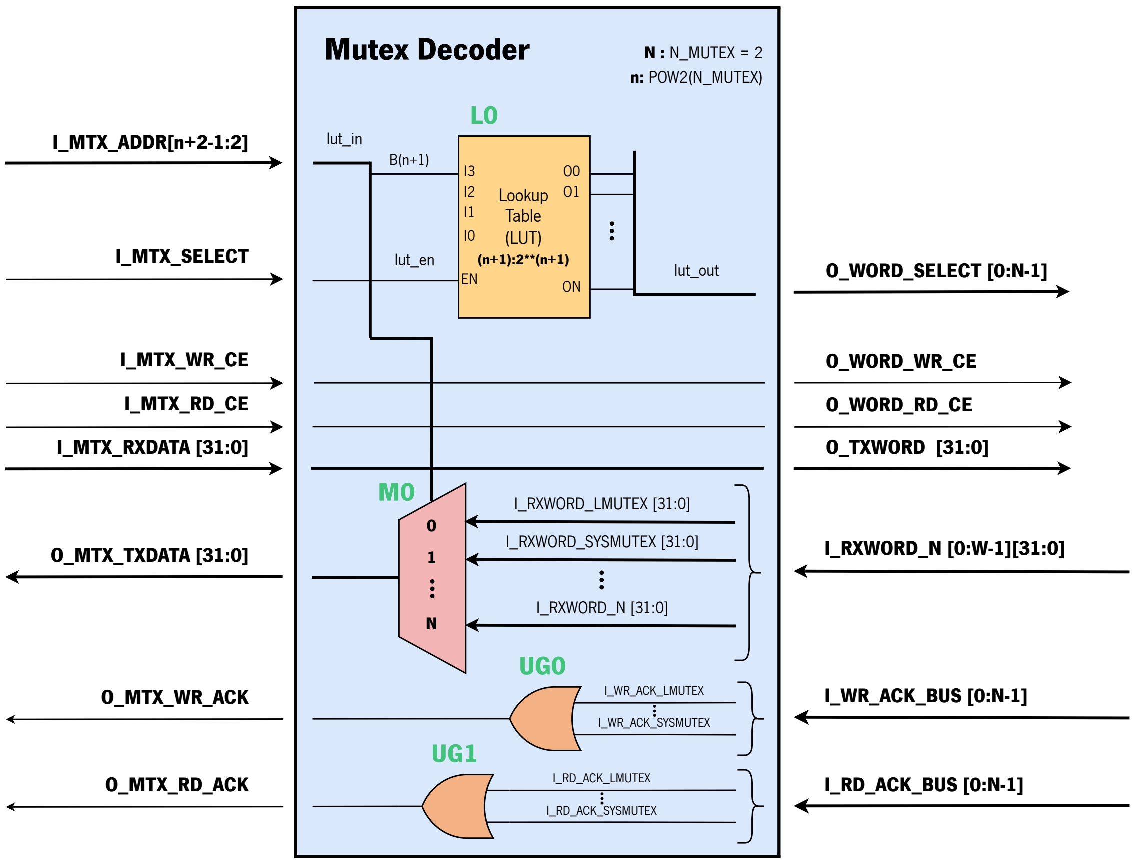


Figure 4.4.5: Mutex Decoder RTL Design Internal Architecture.

Each one of these mutexes respectively prevents race conditions related with the local memory, section 4.5.3, and the main memory, section 4.6.5. The system calls specified for handling the mutex resource, section 4.5.1, comply with the establishment of this bit's logical state in the SLD to indicate the currently active mutex.

The mutex decoder entity is the equivalent to the HAL-ASOS' local bus decoder, but in this case it only decodes the mutex resources, instead of including other hardware resources, like the LRAM, or the ZCU. As forementioned in section 4.2.3, the dissertation work directly maps the SLD to the hardware resources except for the mutexes, which use the mutex decoder logic element, fig. 4.4.5.

4.5 Hardware Resources

With the resource management presented in section 4.4 it is possible to access multiple services of the accelerator through the S00 Control and S01 Data interfaces. This section will present all of these host- and accelerator-available services, specifically, related to synchronization and exclusivity, interrupt management, memory management, event management related with the aforementioned interfaces, and, lastly, parameter configuration concerning the shared portion of main memory.

4.5.1 Hardware Mutex

The microkernel's exclusivity mechanism follows the premises of HAL-ASOS, and, as so, it makes use of two mutexes to implement mutual exclusion regarding the LRAM, with the LMUTEX, and the SYSRAM, with the SYSMUTEX. The term SYSRAM refers to the section of the host's main memory dedicated to the accelerator. Despite protecting different memory segments they are both based on the same hardware module with the internal architecture depicted in fig. 4.5.1. Since one did not have access to the source code of HAL-ASOS, the development of the hardware mutex followed its available RTL schematic. Figure 4.5.1 represents this schematic, but with some changes to fit the microkernel needs. One then interpreted the available design and coded the hardware mutex similar to other modules of the microkernel. Thus, as a whole, the functioning of the mutex is similar to the one in HAL-ASOS.

The first thing to note is that the hardware mutex is a dual port module. This means that it needs to have logic that handles lock and unlock requests from the A port, regarding the host system, and also the B port, concerning the hardware microkernel. For this, each port is associated with its own datapath. As the module must be written and read by the host system, it needs to implement datapath logic to comply

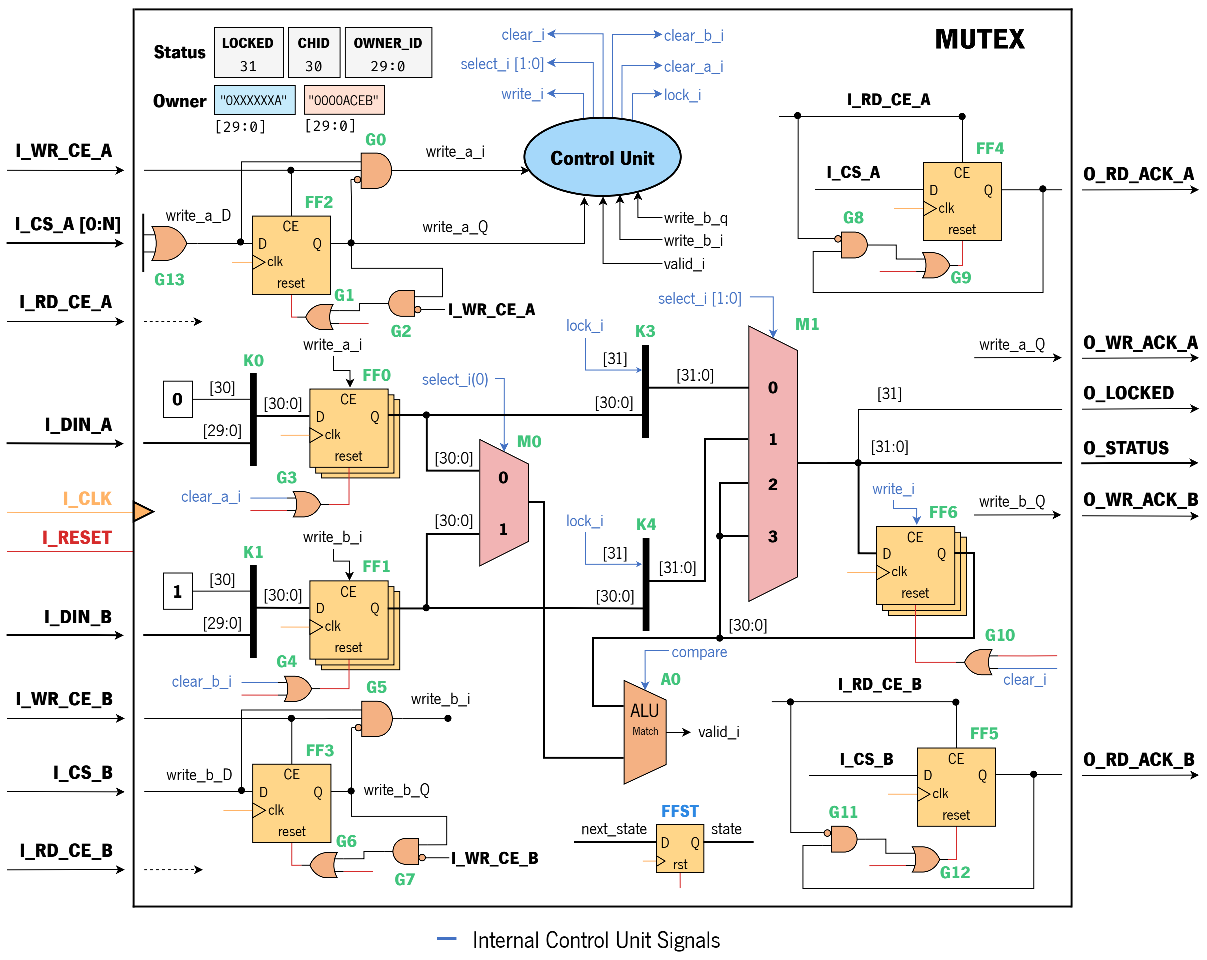


Figure 4.5.1: Hardware Mutex RTL Design Internal Architecture, adapted [62].

with the generic bus model, i.e., the aforementioned model applied in the dissertation based on slave module write and read acknowledges. For this effect, one has FF2 and FF4, which, respectively, generate the write and read acknowledges for the port A (host port), and FF3 and FF5, which, respectively, generate the write and read acknowledges for port B (microkernel port). This logic allows for the generation of the acknowledges one clock cycle later after the assertion of the respective clock enable and chip select. This can be seen in listing A.6. To perform a lock of the mutex it is necessary to write the owner ID of a certain entity (host or kernel) to the data registers of the mutex. The host uses the flip-flop FF0, whilst the kernel uses the flip-flop FF1, listing 4.15.

Listing 4.15: Hardware Mutex Data Registers (HDL).

```
-------------------------------------------------------------------------------
---- Datapath
-------------------------------------------------------------------------------
cs_a_i <= I_CS_A(0) or I_CS_A(1);

-- Channel A data (OWNER_ID) latch
-------------------------------------------------------------------------------
```

```
data_a_D <= C_HW_CHANNEL_A_ID & I_DIN_A(C_OWNER_ID_OFFSET downto 0);
-------------------------------------------------------------------------------------------
DATA_A_FF0 : process (I_CLK)
-------------------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if (I_RESET = '1') or (clear_a_i = '1') then
            data_a_Q <= (others => '0');
        elsif (I_WR_CE_A = '1') then
            data_a_Q <= data_a_D;
        end if;
    end if;
end process DATA_A_FF0;
-------------------------------------------------------------------------------------------

-- Channel B data (OWNER_ID) latch
-------------------------------------------------------------------------------------------
data_b_D <=  C_HW_CHANNEL_B_ID & I_DIN_B(C_OWNER_ID_OFFSET downto 0);
-------------------------------------------------------------------------------------------
DATA_B_FF1 : process (I_CLK)
-------------------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if (I_RESET = '1') or (clear_b_i = '1') then
            data_b_Q <= (others => '0');
        elsif (I_WR_CE_B = '1') then
            data_b_Q <= data_b_D;
        end if;
    end if;
end process DATA_B_FF1;
-------------------------------------------------------------------------------------------
```

The owner ID of the hardware is always “0000ACEB”, while the owner ID of the host is “0XXXXXXA”. The host’s owner ID was established this way because the software application might want to lock or unlock the mutex with multiple threads, which means that no other thread other than the one that locked the mutex can unlock it. This mimics the way a typical mutex object functions in software. The hardware mutex functions in the following way: the first entity to write its owner ID to the data register of the mutex acquires it; a successive write of another owner ID that is different from the one that locked the mutex in first place does not unlock it, so it must wait for the release of the resource; the mutex is unlocked by writing the same owner ID that locked the mutex to the data register; since one is talking about mutual exclusion, only A or B can have the mutex resource at a certain time, this being favored as well by the module’s control unit, *c.f* fig. 4.5.2. Internally, the writing of the owner ID to the data register, FF0 or FF1, depending on the entity, concatenates the 30-bit owner ID word with a bit indicating its channel ID, at K0/K1. A ’0’ represents the A channel, i.e., the host system, and ’1’ represents channel B, i.e., the hardware kernel. The FF0 and FF1 registers actually store the 30-bit words after concatenation. The assertion of the *write_a_i* or the *write_b_i* are fed onto the mutex FSM but also promote the data in FF0 and FF1 to the register’s Q output, respectively.

Listing 4.16: Hardware Mutex Multiplexers (HDL).

```vhdl
-- Channel data decoding
-----------------------------------------------------------------------------------------
DATA_DECODING_M0 : process (select_i(0),data_a_Q,data_b_Q)
-----------------------------------------------------------------------------------------
begin
    data_M0 <= (others => '0');
    case select_i(0) is
        when '0' => data_M0 <= data_a_Q;
        when '1' => data_M0 <= data_b_Q;
        when others => null;
    end case;
end process DATA_DECODING_M0;
-----------------------------------------------------------------------------------------

-- Status decoding
-----------------------------------------------------------------------------------------
STATUS_DECODING_M1 : process (select_i,lock_i,data_a_Q,data_b_Q,status_Q)
-----------------------------------------------------------------------------------------
    variable select_M1 : INTEGER RANGE 0 TO 3;
begin
    select_M1 := to_integer(unsigned(select_i));
    data_M1 <= (others => '0');
    case select_M1 is
        when 0      => data_M1 <= lock_i & data_a_Q;
        when 1      => data_M1 <= lock_i & data_b_Q;
        when 2 | 3  => data_M1 <= status_Q;
        when others => null;
    end case;
end process STATUS_DECODING_M1;
-----------------------------------------------------------------------------------------
```

In the FSM, one expects the assertion of these *write_a_i* and *write_b_i* signals. The FSM then passes from state #0, deemed as *free*, to #4 *accept_a* or #1 *accept_b* depending on the latter. The channel considered as priority is the B channel, pertaining to the kernel. This observed in fig. 4.5.2 since one can only enter *accept_a* if *write_b_i* is not asserted. Following the FSM flow, when reaching *accept_b* or *accept_a*, the mutex clears the corresponding data register (FF0 or FF1), and selects the corresponding word in the M0 multiplexer. The *select_i* signal with "01" selects the channel B word containing the owner ID and the channel ID, and the same signal with "00" selects the channel A word. Note that only the least significant bit of the select is used in M0. The HDL referent to the M0 and M1 multiplexers is in listing 4.16.

These words, FF1_Q and FF0_Q are also concatenated in K3/K4 with their respective lock bit. If the flow of the mutex's FSM follows the channel B then the bit will be asserted on channel B, and the same happens for channel A if the FSM performs a channel A lock. This is dictated by the *lock_i* output portrayed in the *accept_b*/*owned_b* and *accept_a*/*owned_a* states. The *owned_b* and *owned_a* states change the select output accordingly. While the *owned_a* state selects "10" the *owned_b* state selects "11", both in multiplexer M1. This is done to keep the mutex status register data the same, i.e., indicating a lock, until the arrival of a new write on one of the data registers that changes the *valid_i* signal from '0' to '1'.

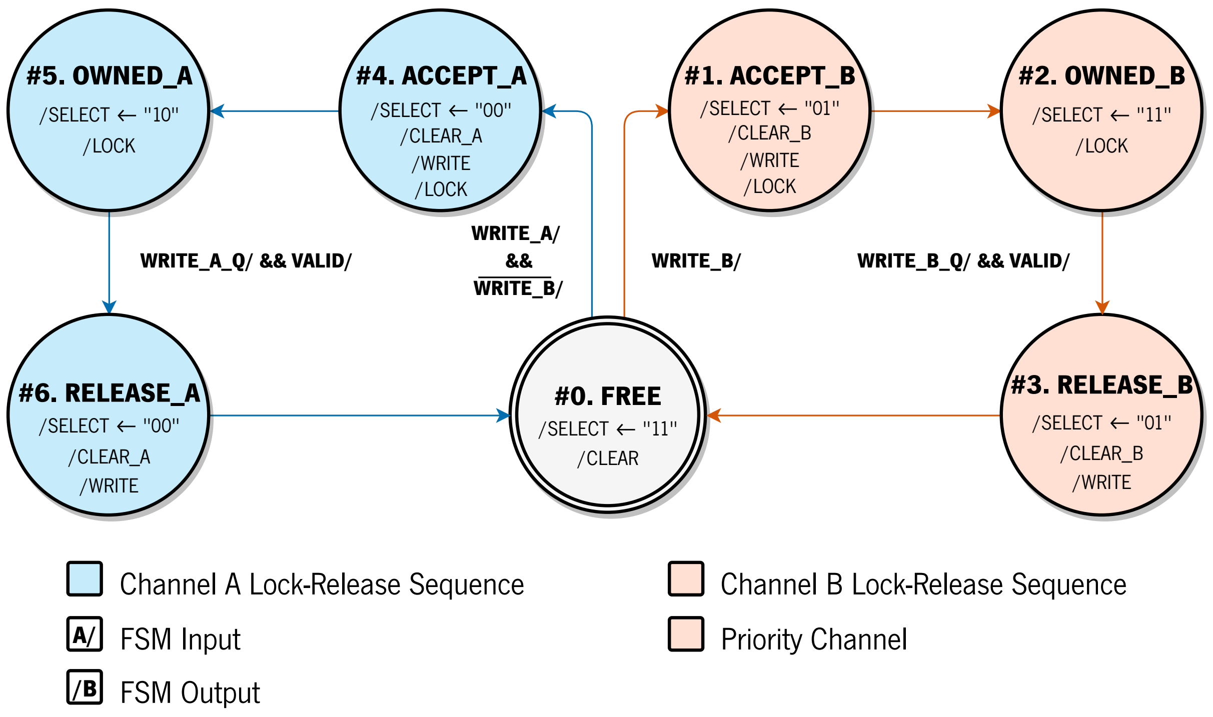


Figure 4.5.2: Hardware Mutex Finite State Machine (FSM), adapted [62], HDL in listing A.7.

As stated, the mutex is only unlocked if the owner ID and channel ID written to the respective date register are the same. This comparison is done in A0, which asserts the *valid_i* signal, listing 4.17, that is then fed onto the FSM. The combination of the *valid_i* signal and the respective indicator that the data register was written again, with the assertion of *write_b_q* or *write_a_q*, makes the FSM go to *release_b* or *release_a* in the next clock cycle, respectively.

Listing 4.17: Hardware Mutex Valid Generation (HDL).

```
-- Owner comparison (ALU)
-----------------------------------------------------------------------------------------
valid_i <= '1' when status_Q(MUTEX_DATA_WIDTH-1-1 downto 0) = data_M0 else '0';
-----------------------------------------------------------------------------------------
-- MORE (Control Unit)
```

In these states, #3 or #6, the select output is then changed again to save the unlock word, FF0_Q or FF1_Q with the lock bit not asserted, onto the status register with the *write_i* signal assertion. The state *release_b* uses “01” to select the word coming from the channel B datapath, while the state *release_a* uses “00” to select the word from the channel A datapath portion. Both states clear their respective data register with *clear_b* or *clear_a*. On the next clock cycle the FSM goes to #0 again where it keeps selecting “11” to use what is on the status register, and also clears it with *clear_i*. The mutex’s status word (coming from FF6) is then decoded in the microprogram to perform the mutex-related system calls, and the locked output, *o_locked* is used specifically in the LMUTEX to block access to the LRAM internally. The HDL representing the status register is represented in listing 4.18.

Listing 4.18: Hardware Mutex Status Register (HDL).

```
-- Status latch
--------------------------------------------------------------------------------
status_D <= data_M1;
--------------------------------------------------------------------------------
STATUS_FF6 : process (I_CLK)
--------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if (I_RESET = '1') or clear_i = '1' then
            status_Q <= (others => '0');
        elsif (write_i = '1') then
            status_Q <= status_D;
        end if;
    end if;
end process STATUS_FF6;
--------------------------------------------------------------------------------
O_STATUS <= status_D;
O_LOCKED <= status_D(C_LOCKED_BIT);
```

In the top left corner of HAL-ASOS is depicted the structure of the mutex status word and the owner IDs from channel A and B. The hardware mutex control unit's representation in HDL is represented in listing A.7. The signals marked in blue (🟦), represent signals coming from the control unit.

4.5.2 Local Interrupt Controller

As it happened in HAL-ASOS, the synchronization with the host system side, in this case containing KIVIO, *c.f* section 3.2, is performed with the utilization of the Local Interrupt Controller (LINTC) resource, fig. 4.5.3. Specifically, this module helps with the notification of the host via Interrupt Request (IRQ), pertaining actions that must be promptly executed, e.g., checking the allocated portion of main memory in the case of important data exchange, or executing specific actions when the hardware microkernel identifies a fault.

For this, the resource implements two registers available to the host system through the S00 Control interface, with FF2 representing the LINTC's control register (writable/readable) and with FF3 concerning the LINTC's status register (read-only), both represented in yellow boxes (🟨), listing 4.19. The structure of the control register word is represented in the top-right corner of fig. 4.5.3 alongside with the structure of the status register word.

The module receives an interrupt source bit vector with the logical state of multiple interrupt sources. Each source then propagates the first stage of IRQ generation, which is the two cycle synchronizer/filter, represented by the FF4 and FF5 flip-flops. As it is considered a sensitive signal, the interrupt source needs this stage to assure protection against metastability, despite, higher clock frequencies might demand for

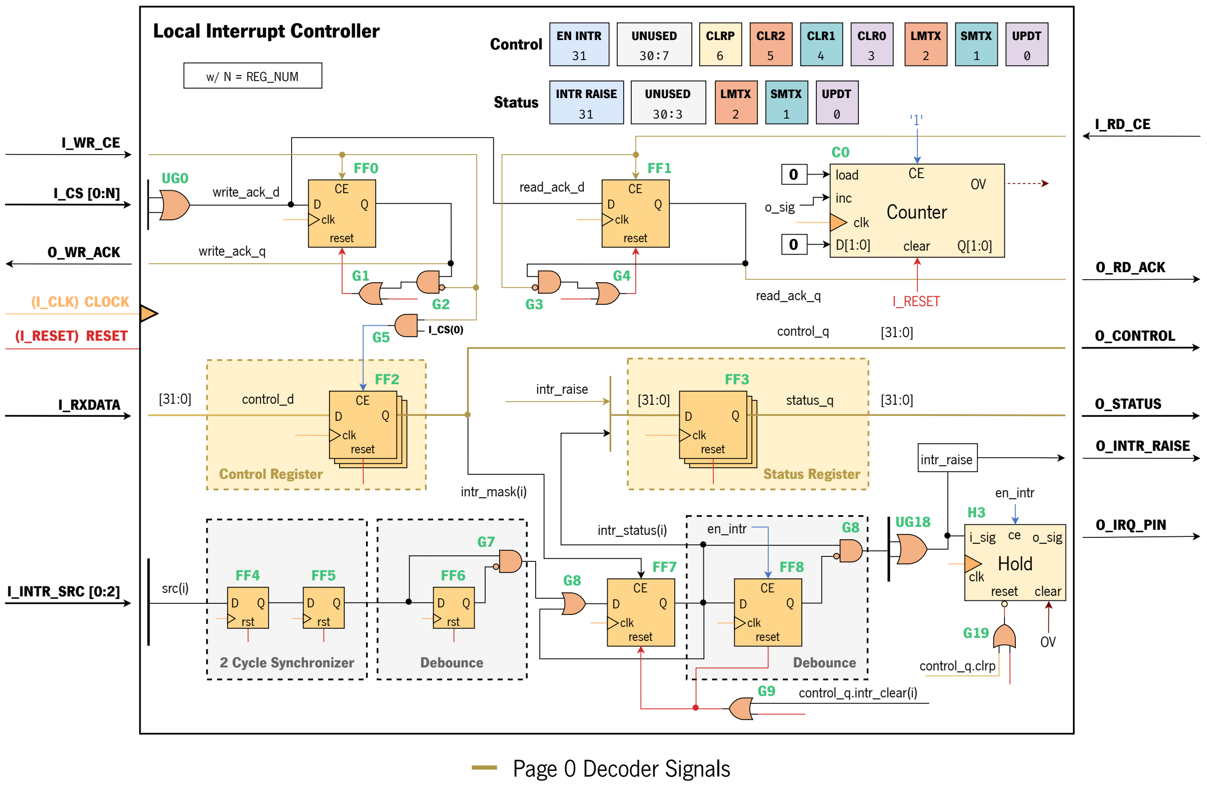


Figure 4.5.3: Local Interrupt Controller RTL Design Internal Architecture (Simplified), expanded in fig. B.2.

a three-stage synchronizer/filter. The signal then propagates to FF6 where a debounce stage is applied (also associated with the G7 gate), this avoids any glitches that might occur in the span of one clock cycle. The next flip-flop is the one responsible for storing the status of a certain interrupt, and thus, it is clock enabled by the interrupt mask, i.e., an interrupt must be masked to generate an IRQ.

Listing 4.19: LINTC Control and Status Registers (HDL).

```vhdl
-- MORE (Acknowledge Generation Logic)
-----------------------------------------------------------------------------------------
CONTROL_FF2 : process (I_CLK)
-----------------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if(I_RESET = '1') then
            control_q <= intc_vector_to_control_data((others => '0'));
        elsif I_WR_CE = '1' and I_CS(0) = '1' then
            control_q <= intc_vector_to_control_data(I_RXDATA);
        end if;
    end if;
end process CONTROL_FF2;
-----------------------------------------------------------------------------------------
O_CONTROL <= intc_control_data_to_vector(control_q);

-----------------------------------------------------------------------------------------
STATUS_FF3 : process (I_CLK)
-----------------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if(I_RESET = '1') then
            status_q <= intc_vector_to_status_data((others => '0'));
        else
```

```
            status_q <= status_d;
        end if;
    end if;
end process STATUS_FF3;
------------------------------------------------------------------------------------------
O_STATUS <= intc_status_data_to_vector(status_q);

-- MORE (IRQ Logic)
```

The areas represented in dashed gray (▨) represent logic that is replicated for each one of the interrupt sources through a *generate* statement in the HDL. Concerning this, the control register has three clear bits and three mask bits, while the status register has three status bits, all associated with the three interrupt sources provided, i.e., local mutex, system mutex, and update. It is then the responsibility of KIVIO, to have an handler for the interrupt and act depending on the contents of the status register, i.e., according to the interrupts' status bits. The HDL for the IRQ logic is represented in listing 4.20

Listing 4.20: LINTC IRQ Logic (HDL).

```
------------------------------------------------------------------------------------------
INTR_GEN:
for i in 0 to C_INTR_NUM-1 generate
------------------------------------------------------------------------------------------
begin
------------------------------------------------------------------------------------------
SYNC_FFI4_FFI5 : process (I_CLK)
------------------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if(I_RESET = '1') then
            src_q(i) <= '0';
            src_q2(i) <= '0';
        else
            src_q(i)  <= I_INTR_SRC(i);
            src_q2(i) <= src_q(i);
        end if;
    end if;
end process SYNC_FFI4_FFI5;
------------------------------------------------------------------------------------------

deb_src_i(i) <= src_q2(i) and not(deb_src_q(i));
------------------------------------------------------------------------------------------
DEBOUNCE_FFI6 : process (I_CLK)
------------------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if(I_RESET = '1') then
            deb_src_q(i)  <= '0';
        else
            deb_src_q(i)  <= src_q2(i);
        end if;
    end if;
end process DEBOUNCE_FFI6;
------------------------------------------------------------------------------------------

mask_src_d(i)  <= deb_src_q(i) or mask_src_q(i);
------------------------------------------------------------------------------------------
STATUS_FFI7 : process (I_CLK)
------------------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if(I_RESET = '1') or (control_q.clear(i) = '1') then
            mask_src_q(i)  <= '0';
        elsif control_q.mask(i) = '1' then
            mask_src_q(i)  <= mask_src_d(i);
        end if;
    end if;
end process STATUS_FFI7;
------------------------------------------------------------------------------------------

status_d.unused <= (others => '0');
------------------------------------------------------------------------------------------
STATUS_D_UPDT_0 : process(mask_src_q(i))
------------------------------------------------------------------------------------------
begin
    status_d.status(i) <= mask_src_q(i);
end process STATUS_D_UPDT_0;
------------------------------------------------------------------------------------------

intr_i(i)  <= mask_src_q(i) and not(deb2_src_q(i));
------------------------------------------------------------------------------------------
DEBOUNCE2_FFI8 : process (I_CLK)
------------------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if(I_RESET = '1') or control_q.clear(i) = '1' then
            deb2_src_q(i)  <= '0';
        elsif control_q.enable_intr = '1' then
            deb2_src_q(i)  <= mask_src_q(i);
        end if;
    end if;
end process DEBOUNCE2_FFI8;
------------------------------------------------------------------------------------------
end generate INTR_GEN;
------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------
SYNC_FF19 : process (I_CLK)
------------------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if(I_RESET = '1') then
            irq_sig_q  <= '0';
        else
            irq_sig_q  <= intr_i(0) or intr_i(1) or intr_i(2);
        end if;
    end if;
end process SYNC_FF19;
------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------
STATUS_D_UPDT_1 : process(irq_sig_q)
------------------------------------------------------------------------------------------
begin
    status_d.intr_raise <= irq_sig_q;
    O_INTR_RAISE <= irq_sig_q;
end process STATUS_D_UPDT_1;
------------------------------------------------------------------------------------------
-- MORE (Counter and Hold Units)
```

Each one of the clear bits of the control register word reset their respective status register, in this generic case represented as FF7, and the control bits 2 to 0 serve as the interrupt masking signals, also generically represented in FF7. After, the flip-flop FF8 applies another layer of debounce to guarantee signal stability. This particular register is clock enabled by the global interrupt enabling/masking, which means that if the *en_intr* signal is not asserted the IRQ will not be generated.

Lastly, the signal enters the hold entity H3 where the signal is held for four clock cycles to assure the host can perceive it. The establishment of four clock cycles is determined by the counter unit C0, since one used a 2-bit counter with the overflow signal as clear input of H3. This could also be done with four flip-flops in cascade, but this approach is more scalable since it allows rapid extension of the hold cycles by adjusting the counter. Additionally, the interrupt raised signal, deemed as *intr_raise* is exposed as an output of the module to allow synchronization with the microprogram in what relates to the LINTC write system call, *c.f* section 4.2.3. These two units are represented in listing 4.21

Listing 4.21: LINTC Counter and Hold Units (HDL).

```
-- Interrupt pin hold
hold_reset_i <= I_RESET or control_q.clear_pin;
-----------------------------------------------------------------------------------------
IRQ_PIN_HOLD_H3 : entity HOLD
-----------------------------------------------------------------------------------------
port map (
I_CLK   => I_CLK,
I_RESET => hold_reset_i,
I_CE    => control_q.enable_intr,
I_CLEAR => counter_ov_i,
I_SIG   => irq_sig_q,
O_SIG   => irq_pulse_i
);
-----------------------------------------------------------------------------------------
O_IRQ_PIN <= irq_pulse_i;

-----------------------------------------------------------------------------------------
IRQ_COUNTER_C0 : entity COUNTER_LOAD
-----------------------------------------------------------------------------------------
generic map(
COUNT_WIDTH => 2)
port map(
I_CLK => I_CLK,
I_CE => '1',
I_RESET => I_RESET,
I_INC => irq_pulse_i,
I_LOAD => '0',
I_D => (others => '0'),
O_Q => OPEN,
O_COUNTER_OV => counter_ov_i
);
-----------------------------------------------------------------------------------------
```

4.5.3 Local Memory

The introduction of a kernel local memory was proposed by [62] in its research and its main function was to store variables related to the hardware task, whilst, paired with the ZCU, acted similarly to a cache. With this one means that the memory was the temporary storage for data that was intended to go to the host's main memory. As the HAL-ASOS' core with the microprogram functions similarly to a CPU, the ZCU allowed data transfers from the local to main memory (or vice-versa) without the intervention of the core or central processing unit of the kernel in a DMA-like fashion, hence the term zero copy.

In the dissertation's scenario, since one did not implement a ZCU, the main purpose of the LRAM is solely to store hw-task variables internally with the use of appropriate system calls that operate over it, or other data pertaining to the hw-task but coming from the host system through the AXI4 lite interface S01

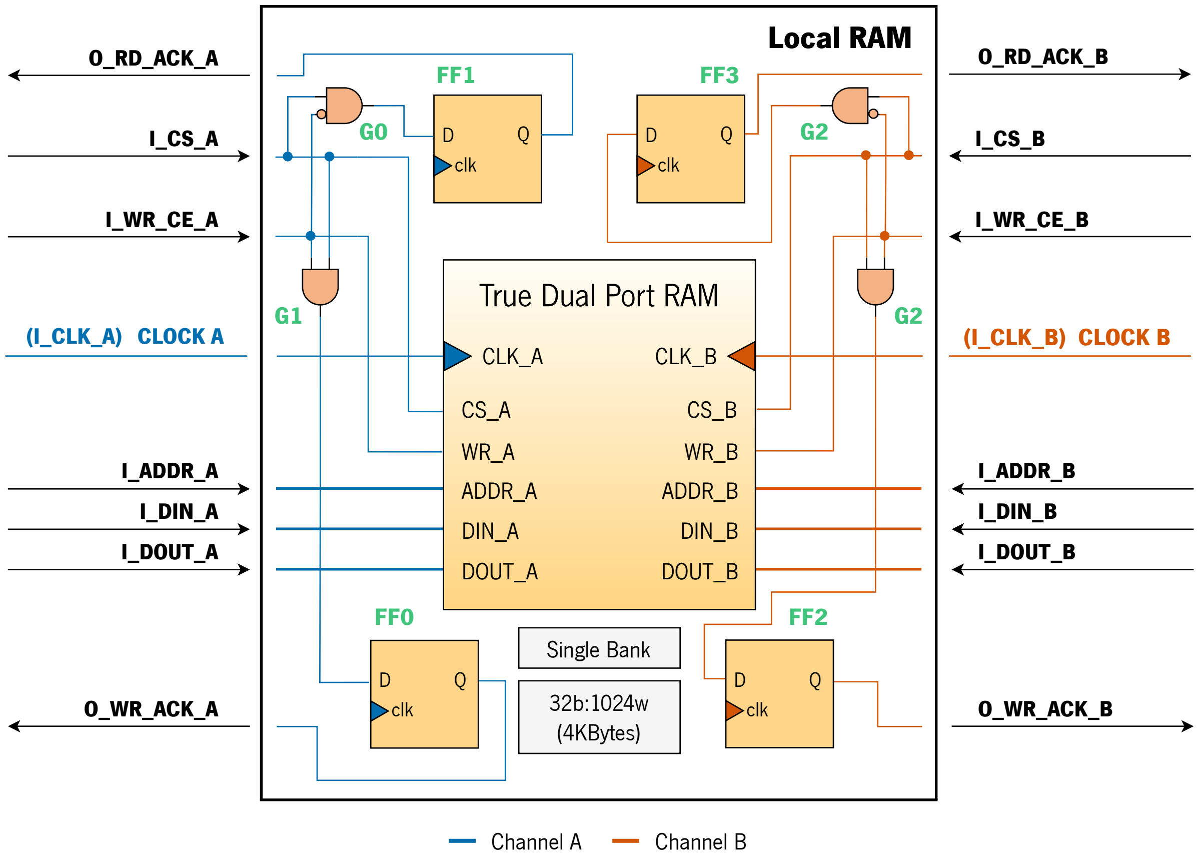


Figure 4.5.4: Local Random-Access Memory (LRAM) RTL Design Internal Architecture.

Data. In this case, the transfers of data from the hardware to the main memory are done directly, utilizing an AXI4 master interface that supports burst transactions. More on that later in section 4.6.5.

HAL-ASOS implemented this memory element as a true dual-port synchronous RAM. The same approach was followed in the dissertation's work, fig. 4.5.4, implementing a RAM capable of being simultaneously written or read by the host system through the A port, and by the kernel/task through the B port. Since the RAM allows this type of interface it needs to be protected with some sort of synchronization mechanism that implements mutual exclusion and, thus, prevents race conditions. That mechanism is the local mutex. The latter is represented in fig. 4.1.1 with a connection to the local memory passing through a blue lock icon to symbolize the protection enforced by this entity. Refer to section 4.5.1 for more information on the design of the hardware mutex. The logical combination of the mutex's locked signal and the LRAM's write enable gives origin to the memory's write clock enable. A write to the memory is only possible when both of them are asserted. The definition of the logical expression that generates the LRAM's write clock enable is given by the local RAM protection entity (*c.f* fig. B.10). This entity is very simple, but it was used to avoid conflicts when inferring the target board's block RAM, by assuring that the

RAM only has one write clock enable for each one of the ports, listing 4.22.

Listing 4.22: Local Memory (TDP RAM) Port Logic (HDL).

```
-- MORE
type ram_t is array (RAM_DEPTH-1 downto 0) of std_logic_vector (RAM_W_WIDTH-1 downto 0);
shared variable ram_name : ram_t;

begin
------------------------------------------------------------------------------------------
PORT_A : process(I_CLK_A)
------------------------------------------------------------------------------------------
begin
    if(I_CLK_A'event and I_CLK_A = '1') then
        if(I_CS_A = '1') then
                if(I_WR_CE_A = '1') then
                    ram_name(to_integer(unsigned(I_ADDR_A))) := I_DIN_A;
                end if;
            O_DOUT_A <= ram_name(to_integer(unsigned(I_ADDR_A)));
        end if;
    end if;
end process PORT_A;
------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------
PORT_B : process(I_CLK_B)
------------------------------------------------------------------------------------------
begin
    if(I_CLK_B'event and I_CLK_B = '1') then
        if(I_CS_B = '1') then
                if(I_WR_CE_B = '1') then
                    ram_name(to_integer(unsigned(I_ADDR_B))) := I_DIN_B;
                end if;
            O_DOUT_B <= ram_name(to_integer(unsigned(I_ADDR_B)));
        end if;
    end if;
end process PORT_B;
------------------------------------------------------------------------------------------
-- MORE (Acknowledge Generation)
```

Another thing to note is that HAL-ASOS uses four 8-bit memories with 1K of depth (10-bit address) leveraging the simplicity of typical microprocessor memory banking. Despite, the local memory of the dissertation uses one 32-bit memory with 1K of depth since it provided the same functionality, i.e., the output of a 32-bit word with one cycle of latency, whilst having reduced design effort for this particular case, also in listing 4.22.

Moreover, since the memory segment needs to comply with the generic bus architecture aforementioned in other sections, i.e., based on slave acknowledges, to make use of the AXI4 lite interface, section 4.6.1, it was imperative to add read and write acknowledge generation logic to the design (FF0 to FF3). When both the chip select and the write clock enable of the RAM are asserted one has the write acknowledge on the next clock cycle. Although, since there is not a read clock enable, the read acknowledge is generated one cycle after the assertion of the chip select with the write clock enable not asserted. This applies for both memory ports. The LRAM's acknowledge logic implementation in HDL can be seen in listing 4.23.

Listing 4.23: Local Memory (TDP RAM) Acknowledge Generation (HDL).

```
-- MORE (Port Logic)
------------------------------------------------------------------------------------------
ACK_REGS_A : process(I_CLK_A)
------------------------------------------------------------------------------------------
begin
    if(I_CLK_A'event and I_CLK_A = '1') then
    O_WR_ACK_A <= i_wr_ce_a and i_cs_a;
    O_RD_ACK_A <= not(i_wr_ce_a) and i_cs_a;
    end if;
end process ACK_REGS_A;
------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------
ACK_REGS_B : process(I_CLK_B)
------------------------------------------------------------------------------------------
begin
    if(I_CLK_B'event and I_CLK_B = '1') then
        O_WR_ACK_B <= i_wr_ce_b and i_cs_b;
        O_RD_ACK_B <= not(i_wr_ce_b) and i_cs_b;
    end if;
end process ACK_REGS_B;
------------------------------------------------------------------------------------------
end architecture rtl;
```

4.5.4 Slave Interface Event Manager

The slave interface event manager, fig. 4.5.5, is the module that handles the timing of the events for both AXI4 lite interfaces, S00 Control and S01 Data. Regarding this, it uses an event manager that reuses the event's module of the SLD, *c.f* section 4.2.5.1, but adds functionality in terms of compliance with the generic bus model, implements an interface timeout counter, and adds specific registers to hold the interfaces' status.

Firstly, as seen in other dissertation modules, the interface event manager adds write and read acknowledge generation logic to comply with the established protocol. For this the flip-flops FF2 and FF3 generate the write and read acknowledges one cycle after the assertion of the write clock enable, *wr_ce*, and the chip select, *cs*. Both flip-flops have logic dedicated to reset their state when the *wr_ce* is not asserted anymore and the *wr_ack* was emitted (G7/G8 and G9/G10).

This timing entity is deeply related with the AXI4 lite interface state machine, *c.f* fig. 4.6.6. Thus, the FSM controls the module by specifying its input triggers, enabling the timeout counter C0, and determining the events to wait on. The aforementioned counter starts at zero timeouts and increments every time a timeout occurs until there is an overflow. With a counter overflow, the gate G2 gives origin to the assertion of the counter clear input, and the latter is reset to its initial state. For the waiting on events, this unit specifies a fixed timeout value of sixteen cycles. Moreover, the module has two explicit read-only registers (FF0 and FF1), which hold the current status of both slave interfaces (S00 and S01), and two implicit

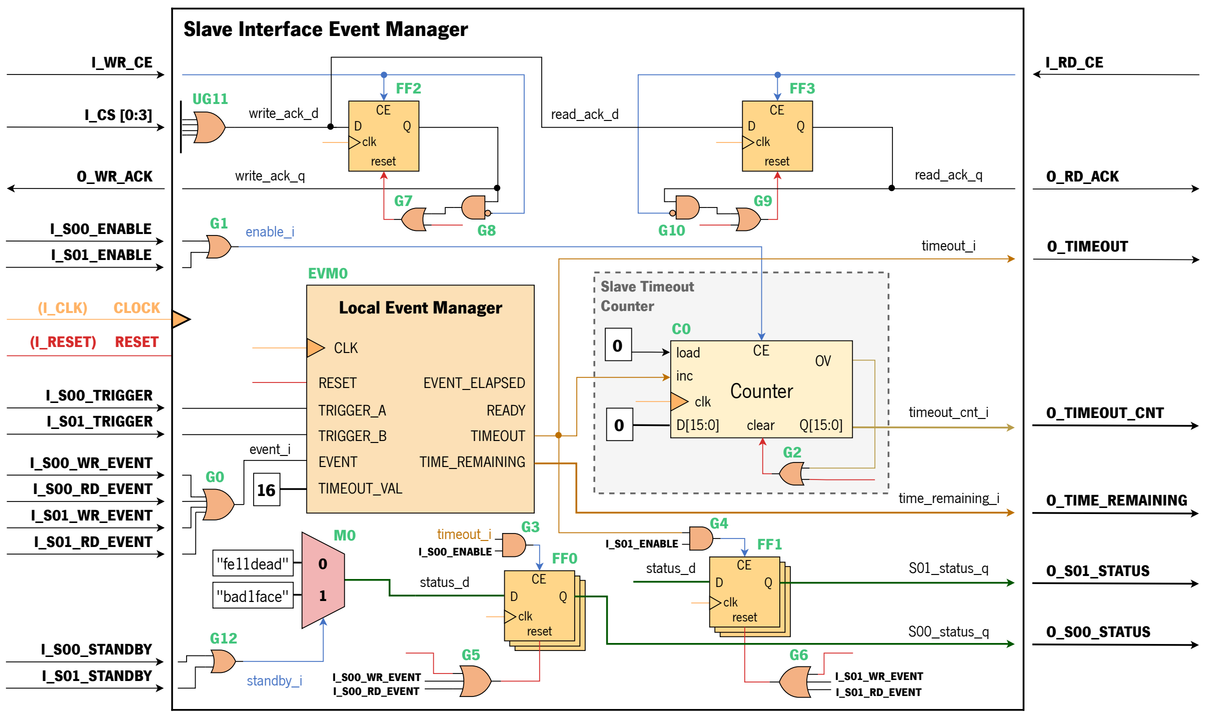


Figure 4.5.5: Slave Interface (S00 and S01) Event Manager RTL Design Internal Architecture.

read-only registers inside the local event manager and the timeout counter, which hold the remaining time until timeout and the number of timeouts since the kernel started running, respectively. With specific logic dependent on AXI4 lite interface FSM triggering, the module is capable of defining three status words for the slave interfaces: (1) “00000000”, which indicates correct operation, (2) “fe11dead”, which indicates AXI signal failure, and (3) “bad1face”, which indicates a failure of slave acknowledge signals. For more information on the triggering of this module by the AXI4 lite interface refer to section 4.6.1. The HDL representation of the slave interface event manager is portrayed by listing 4.24.

Listing 4.24: Slave Interface Event Manager (HDL).

```
cs_i <= I_CS(0) or I_CS(1) or I_CS(2) or I_CS(3);
event_i <= I_S00_WR_EVENT or I_S00_RD_EVENT or I_S01_WR_EVENT or I_S01_RD_EVENT;
timeout_value_i <= std_logic_vector(to_unsigned(C_TIMEOUT_CYCLES,timeout_value_i'LENGTH));
enable_i <= I_S00_ENABLE or I_S01_ENABLE;
standby_i <= I_S00_STANDBY or I_S01_STANDBY;
reset_i <= I_RESET or timeout_cnt_ov_i;

-------------------------------------------------------------------------
EVENT_MANAGER_S0 : entity EVENT_MANAGER
-------------------------------------------------------------------------
port map(
I_CLK                => I_CLK,
I_RESET              => I_RESET,
I_TRIGGER_B          => I_S00_TRIGGER,
I_TRIGGER_A          => I_S01_TRIGGER,
I_EVENT              => event_i,
I_TIMEOUT_VALUE      => timeout_value_i,
O_EVENT_ELAPSED      => OPEN,
O_READY              => OPEN,
O_TIMEOUT            => timeout_i,
O_TIME_REMAINING     => time_remaining_i
);
----------------------------------------------------------------------
O_TIMEOUT <= timeout_i;
O_TIME_REMAINING <= std_logic_vector(to_unsigned(to_integer(unsigned(time_remaining_i)),
    O_TIME_REMAINING'LENGTH));

----------------------------------------------------------------------
TIMEOUT_COUNTER_C0 : entity COUNTER_LOAD
----------------------------------------------------------------------
generic map(
COUNT_WIDTH => C_SLV_IF_EVENT_MANAGER_WIDTH)
port map(
I_CLK => I_CLK,
I_CE => enable_i,
I_RESET => reset_i,
I_INC => timeout_i,
I_LOAD => '0',
I_D => (others => '0'),
O_Q => timeout_count_i,
O_COUNTER_OV => timeout_cnt_ov_i
);
----------------------------------------------------------------------
O_TIMEOUT_CNT <= std_logic_vector(to_unsigned(to_integer(unsigned(timeout_count_i)),
    O_TIMEOUT_CNT'LENGTH));

----------------------------------------------------------------------
STATUS_M0 : process (standby_i)
----------------------------------------------------------------------
begin
    status_d <= (others => '0');
    case standby_i is
        when '0' =>
            status_d <= x"fe11dead";
        when '1' =>
            status_d <= x"bad1face";
        when others =>
            null;
    end case;
end process STATUS_M0;
----------------------------------------------------------------------

----------------------------------------------------------------------
STATUS_FF0 : process (I_CLK)
----------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if(I_RESET = '1') or I_S00_WR_EVENT = '1' or I_S00_RD_EVENT = '1' then
            s00_status_q <= (others => '0');
        elsif I_S00_ENABLE = '1' and (timeout_i = '1') then
            s00_status_q <= status_d;
        end if;
    end if;
end process STATUS_FF0;
----------------------------------------------------------------------
O_S00_STATUS <= s00_status_q;

----------------------------------------------------------------------
STATUS_FF1 : process (I_CLK)
----------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if(I_RESET = '1') or I_S01_WR_EVENT = '1' or I_S01_RD_EVENT = '1' then
            s01_status_q <= (others => '0');
        elsif I_S01_ENABLE = '1' and timeout_i = '1' then
            s01_status_q <= status_d;
        end if;
    end if;
end process STATUS_FF1;
----------------------------------------------------------------------
O_S01_STATUS <= s01_status_q;

-- MORE (Write and Read Acknowledgement Logic)
```

4.5.5 Interface Configuration Registers

The interface registers module only serves the purpose of storing important information related to the shared memory segments, namely the SYSRAM and the LRAM. For this unit implements three writable and readable registers associated with the SYSRAM and one read-only register associated with the LRAM. The register FF2 stores the base address where the allocated memory for the accelerator starts. The contents of this register are then used later in the SLD to add the base to the intended address, *c.f* section 4.2.5. Similarly, the register FF3 stores the initial position of the memory mapped KFIFO, being used in extended-feature procedures to once again calculate the correct address, *c.f* section 5.1. The register FF4 stores the beginning position of the allocated memory designated for data exchange with the host system and that does not include the KFIFO. Additionally, this unit complies with the use of the generic bus interface based on slave acknowledges (FF0 and FF1). The module's HDL representation can be seen in listing 4.25.

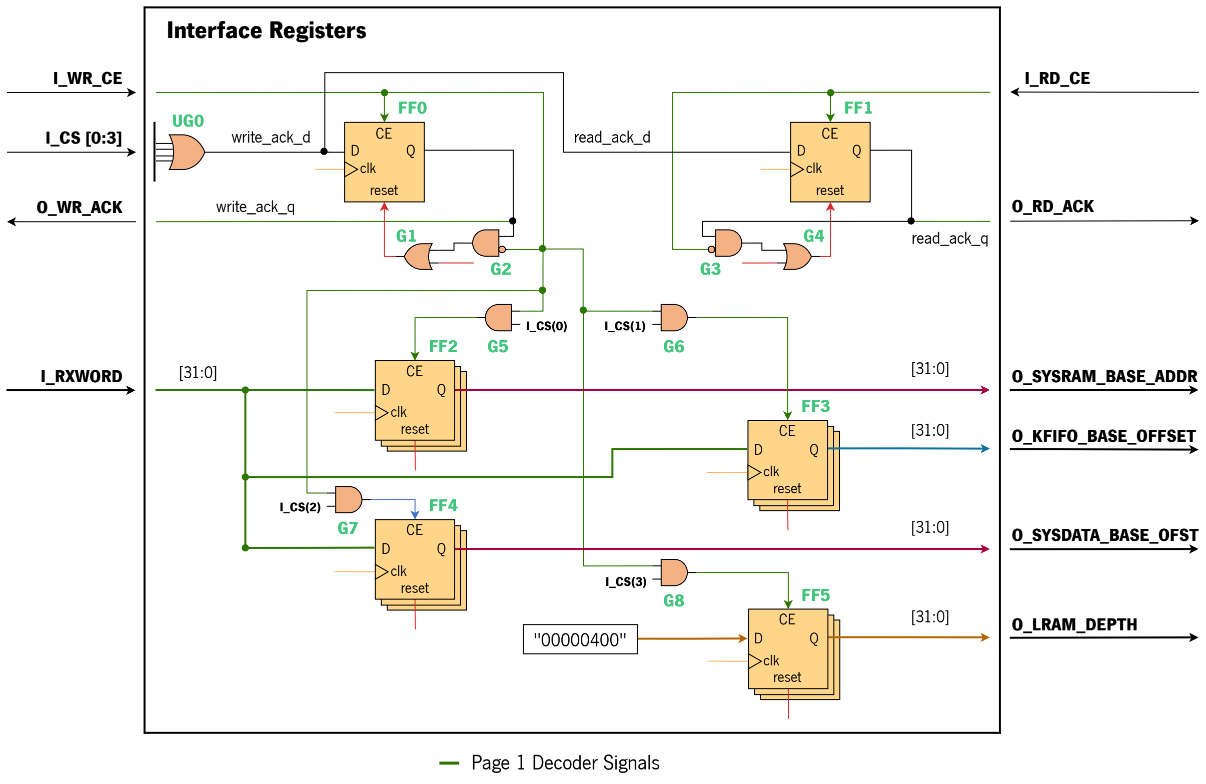


Figure 4.5.6: Interface Registers RTL Design Internal Architecture.

Listing 4.25: Interface Registers (HDL).

```
cs_i <= I_CS(0) or I_CS(1) or I_CS(2) or I_CS(3);

-- MORE (Write and Read Acknowledgement Logic)
-------------------------------------------------------------------------
SYSRAM_BASE_ADDR_FF2 : process (I_CLK)
```

```
----------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if(I_RESET = '1') then
            O_SYSRAM_BASE_ADDR <= (others => '0');
        elsif I_WR_CE = '1' and I_CS(0) = '1' then
            O_SYSRAM_BASE_ADDR <= I_RXWORD;
        end if;
    end if;
end process SYSRAM_BASE_ADDR_FF2;
----------------------------------------------------------------------

----------------------------------------------------------------------
KFIFO_BASE_OFFSET_FF3 : process (I_CLK)
----------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if(I_RESET = '1') then
            O_KFIFO_BASE_OFFSET <= (others => '0');
        elsif I_WR_CE = '1' and I_CS(1) = '1' then
            O_KFIFO_BASE_OFFSET <= I_RXWORD;
        end if;
    end if;
end process KFIFO_BASE_OFFSET_FF3;
----------------------------------------------------------------------

----------------------------------------------------------------------
SYSDATA_BASE_OFFSET_FF4 : process (I_CLK)
----------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if(I_RESET = '1') then
            O_SYSDATA_BASE_OFFSET <= x"00000080"; -- offset 128
        elsif I_WR_CE = '1' and I_CS(2) = '1' then
            O_SYSDATA_BASE_OFFSET <= I_RXWORD;
        end if;
    end if;
end process SYSDATA_BASE_OFFSET_FF4;
----------------------------------------------------------------------

----------------------------------------------------------------------
LRAM_DEPTH_FF5 : process (I_CLK)
----------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if(I_RESET = '1') then
           O_LRAM_DEPTH <= std_logic_vector(to_unsigned(C_LRAM_DEPTH,O_LRAM_DEPTH'LENGTH));
        else
           O_LRAM_DEPTH <= std_logic_vector(to_unsigned(C_LRAM_DEPTH,O_LRAM_DEPTH'LENGTH));
        end if;
    end if;
end process LRAM_DEPTH_FF5;
----------------------------------------------------------------------
```

4.6 External Interfaces

This section will describe the microkernel interfaces with the host system. For this, firstly one describes the AXI4 lite slave interface developed, *c.f* section 4.6.1, utilized by S00 Control and S01 Data. Afterwards, one characterizes both interfaces and their register mappings, and ultimately the M00 System interface is portrayed.

4.6.1 AXI Lite Interface

The AXI4 Lite Interface module is used by both the S00 control-oriented interface and the S01 data-oriented interface. The module itself has to comply with the register-based AXI4 lite protocol, responding to the appropriate signals at the appropriate time. For this, the module has a control unit, marked in bright blue (🟦) in fig. 4.6.1, that receives as inputs AXI signals and also outputs specific AXI signals back. This section will explain all the datapath of the interface in detail, as well as its control unit, i.e., the main core of the module. This explanation will cover not only the RTL design of fig. 4.6.1, but also, the module's implementation in VHDL.

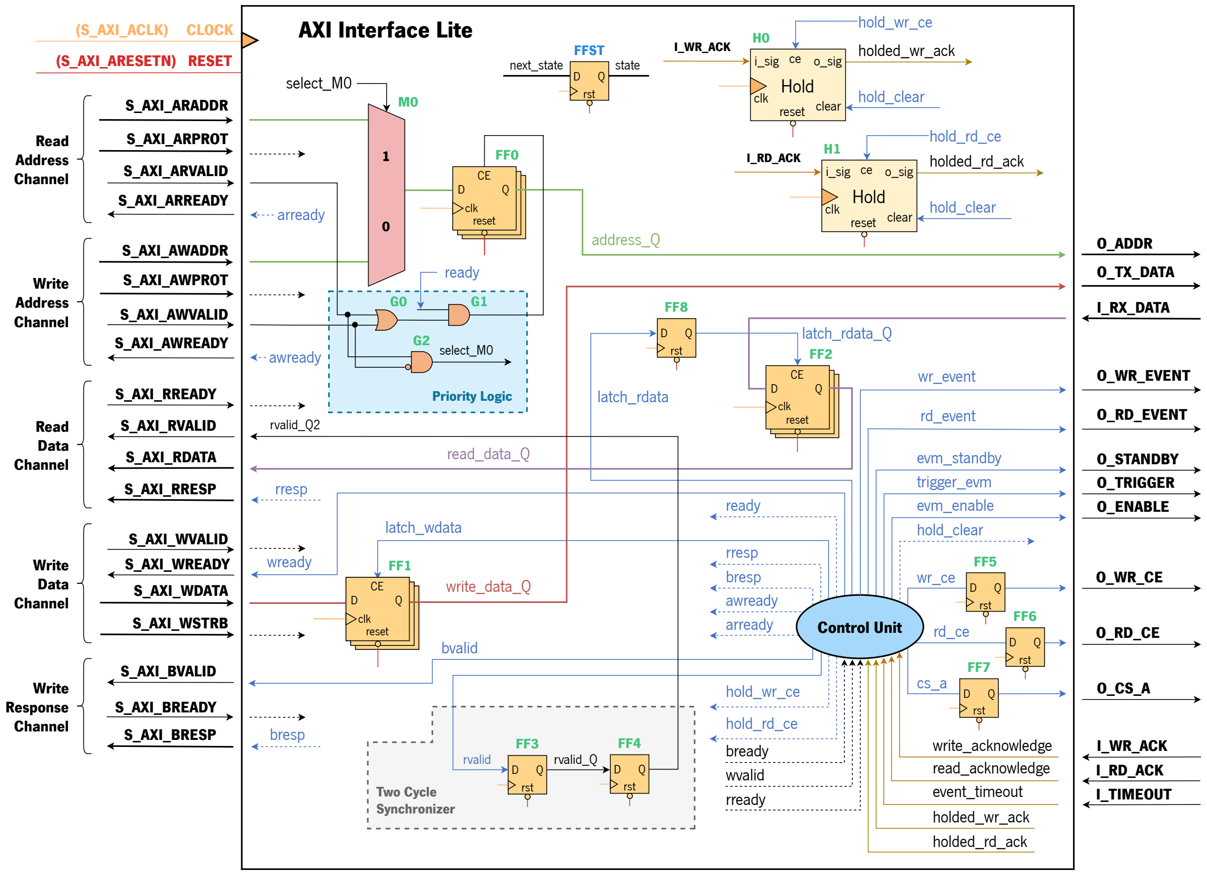


Figure 4.6.1: AXI4 Lite Interface RTL Design Internal Architecture.

Starting with the datapath, one of the main choices made in the design of the interface's datapath was that one would try to save as many resources as possible, in terms of logic gates, flip-flops, and multiplexers, just to name a few. Another choice made was that all the important multi-bit signals coming as interface inputs would end up being assured by a flip-flop, to ensure data correctness, even if that meant that one would increase the overall latency of the interface by a few cycles. The signals submitted to flopping were the AXI address signal, represented in green (🟩), the AXI write data signal, represented in red (🟥), and the AXI read data signal, represented in purple (🟪). In this case, that choice resulted in some more flip-flops that function as synchronizers (FF3 to FF8) for some AXI signals, but that was a trade-off one was willing to make once again to ensure data correctness, i.e., making sure one was writing to, or reading from the right place, and that what was read or written was in fact the actual value to read or write.

The AXI lite protocol presents five main channels of communication, the read address channel, the write address channel, the read data channel, the write data channel, and ultimately, the write response channel. As it is possible to perceive, the write transaction has three channels while the read transaction only has two channels. That is justified by the fact that the read transaction does not require an acknowledgment stage, and that is a factor to have in mind when designing the module's state machine, fig. 4.6.6. Note that some signals of multiple channels are not connected to anything because they really do not contribute to state transition nor represent useful information, like an address or a multi-bit data signal would represent. In this case, the signals left out were the ones pertaining to the write data strobe (*s_axi_wstrb*), which indicates what bits of *s_axi_wdata* must be written to the slave, and the address read and write protection of the master, namely *s_axi_arprot* and *s_axi_awprot*, respectively.

In the address datapath, one encounters logic to define the priority channel in the transaction, denoted in the dashed blue rectangle (⬜), a multiplexer (M0) to select which address is going to be the combinational input of the address flip-flop (FF0), and the FF0 flip-flop itself. A choice was made to define the write transaction as the priority, since effectively saving the data is viewed as more important than reading it, i.e., if the data was correctly written to the slave, the read transaction can wait and still perform a correct read. On the contrary, a read made to a certain address might fail to obtain the proper data if the write is not executed first, when talking about a read and write on the same address. The priority logic block defines that if both of the protocol initiating signals *s_axi_arvalid* and *s_axi_awvalid* are set to the logic value '1', the one selected in the M0 multiplexer will be the write address (*s_axi_awaddr*), according to the G2 gate. For the clock enable of the FF0 flip-flop, one uses a logic *and* (G1) of the ready signal coming from the FSM, marked in blue (🟦) in listing 4.26, and one of the protocol initiating signals (*s_axi_arvalid* or *s_axi_awvalid*), assuring that, in the next clock cycle, one has the address at the module's output *o_addr*.

Listing 4.26: AXI4 Lite Interface Datapath: Address (HDL).

```
-- Address latching
-----------------------------------------------------------------------
ADDR_LATCH_FF0 : process (S_AXI_ACLK)
-----------------------------------------------------------------------
```

```
    begin
        if rising_edge(S_AXI_ACLK) then
            if reset_i = '1' then
                axi_addr_Q <= (others => '0');
            elsif (ready_i = '1') and (S_AXI_AWVALID = '1' or S_AXI_ARVALID = '1') then
                axi_addr_Q <= axi_addr_D;
            end if;
        end if;
    end process ADDR_LATCH_FF0;
    ----------------------------------------------------------------------
    O_ADDR <= axi_addr_Q;

    -- Address select
    ----------------------------------------------------------------------
    MUX_ADDR_M0 : process (S_AXI_AWVALID, S_AXI_ARVALID, S_AXI_AWADDR,
    S_AXI_ARADDR)
    ----------------------------------------------------------------------
        variable select_M0 : std_logic;
    begin
        select_M0 := ((not S_AXI_AWVALID) and S_AXI_ARVALID);
        axi_addr_D <= (others => '0');
        case select_M0 is
            when '0' =>
                axi_addr_D <= S_AXI_AWADDR;
            when '1' =>
                axi_addr_D <= S_AXI_ARADDR;
            when others =>
                null;
        end case;
    end process MUX_ADDR_M0;
    ----------------------------------------------------------------------
```

The datapath referring to the write and read data also assures valid data at the *o_tx_data* and *s_axi_rdata* outputs when the signals *latch_wdata* and *latch_rdata* reach the flip-flops FF1 and FF2, respectively. As seen in fig. 4.6.1, the signals *latch_wdata* and *latch_rdata* are outputs of the module's control unit, and are represented in listing 4.27 in blue (🟦).

Listing 4.27: AXI4 Lite Interface Datapath: Data (HDL).

```
    -- Write data latching
    ----------------------------------------------------------------------
    TX_DATA_FF1 : process (S_AXI_ACLK)
    ----------------------------------------------------------------------
    begin
        if rising_edge(S_AXI_ACLK) then
            if reset_i = '1' then
                txdata_Q <= (others => '0');
            elsif latch_wdata_i = '1' then
                txdata_Q <= S_AXI_WDATA;
            end if;
        end if;
    end process TX_DATA_FF1;
    ----------------------------------------------------------------------
    O_TX_DATA <= txdata_Q;

    -- Read data latching
    ----------------------------------------------------------------------
    RX_DATA_FF2 : process (S_AXI_ACLK)
    ----------------------------------------------------------------------
    begin
        if rising_edge(S_AXI_ACLK) then
            if reset_i = '1' then
                rxdata_Q <= (others => '0');
            elsif latch_rdata_Q = '1' then
                rxdata_Q <= I_RX_DATA;
            end if;
        end if;
    end process RX_DATA_FF2;
    ----------------------------------------------------------------------------
    S_AXI_RDATA <= rxdata_Q;
```

Note that other datapath elements also include the write and read acknowledge *hold* entities (H0 and H1), the state register (FFST), and the forementioned flip-flop synchronizers (FF3 to FF8). For more information about these datapath elements, please refer to section section 4.6.1.1.

4.6.1.1 Optimized Control Unit and Associated Dapapath

As stated earlier in section 4.6.1, the AXI lite interface FSM complies with the AXI protocol by responding to certain AXI signals. This control unit is deemed as optimized for two main reasons. Firstly, it deals with the failure of AXI signals that specify the end of a transfer, i.e., *s_axi_bready* and *s_axi_rready*, for the write and read, respectively, and the failure of generic bus signals coming from the slave, i.e., the write and read acknowledges. Secondly, it saves a clock cycle after a read transfer by skipping over the *idle* state, if both of the signals that initiate a write transfer are asserted (*s_axi_awvalid* and *s_axi_wvalid*), fig. 4.6.6.

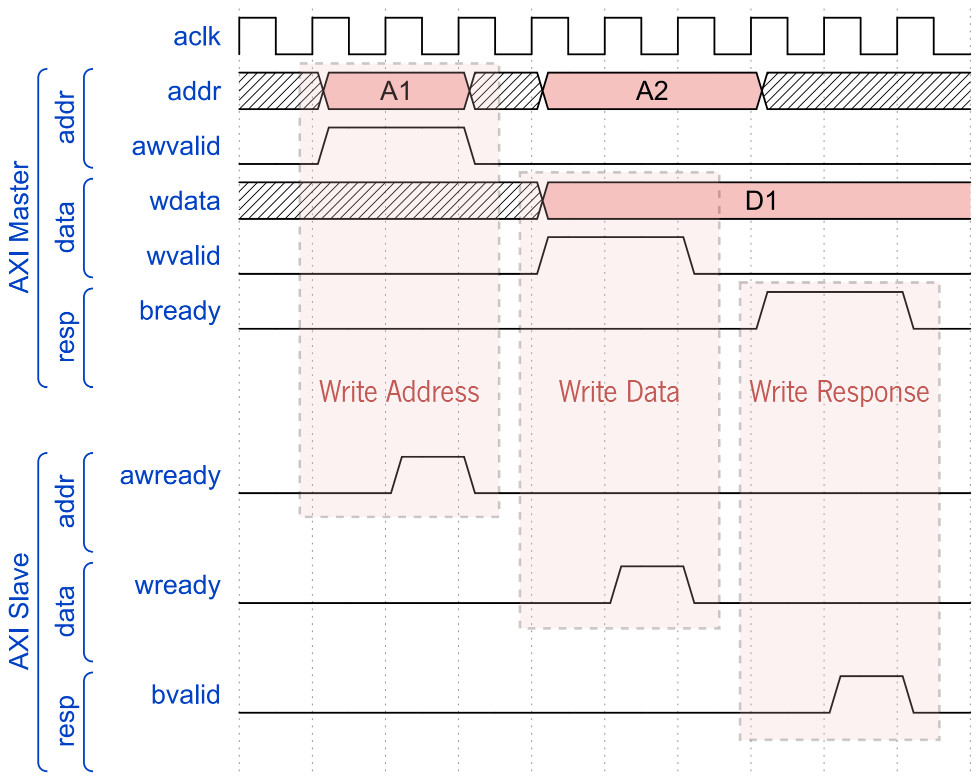


Figure 4.6.2: AXI4 Lite Protocol Waveform: Write Transfer.

Figure 4.6.2 depicts the protocol waveform for a AXI4 lite write transfer. In this case, the master sends an address accompanied by the address valid signal to identify a valid address, followed by the data and its data valid signal, and ultimately asserts its bus ready signal waiting for the slave's response. The slave has to assert its address ready signal and its data ready signal to accept the address and the data, respectively. Finally, the slave must assert the bus valid signal to send an acknowledgment to the master and end the transfer. Even though the example of fig. 4.6.2 refers to a AXI4 lite write transfer, the same principles of valid-ready handshake expand across the AXI protocol, i.e., even the AXI4 lite read transfer, fig. 4.6.5, and the AXI4 read and write transfers utilize these handshake signals.

The state machine of section 4.6.1.1 starts with the *idle* state, seen by the reset state of FFST in listing 4.28, where the *ready* signal is asserted to latch the selected address onto the address flip-flop FF0 (paragraph four of section 4.6.1), and also where both bit signal-holding entities H0 and H1 are cleared, listing 4.28.

Listing 4.28: AXI4 Lite Interface Datapath: State Flip-Flop and Hold Units (HDL).

```
-- State register
-----------------------------------------------------------------------
FFST : process (S_AXI_ACLK)
-----------------------------------------------------------------------
begin
    if rising_edge(S_AXI_ACLK) then
        if reset_i = '1' then
            state <= S0_IDLE;
        else
            state <= next_state;
        end if;
    end if;
end process FFST;
-----------------------------------------------------------------------
-- Write ack hold
-----------------------------------------------------------------------
WR_ACK_HOLD_H0 : entity HOLD
-----------------------------------------------------------------------
port map (
I_CLK   => S_AXI_ACLK,
I_RESET => reset_i,
I_CE    => hold_wr_ce_i,
I_CLEAR => hold_clear_i,
I_SIG   => I_WR_ACK,
O_SIG   => holded_wr_ack_i);
-----------------------------------------------------------------------
-- Read ack hold
-----------------------------------------------------------------------
RD_ACK_HOLD_H1 : entity HOLD
-----------------------------------------------------------------------
port map (
I_CLK   => S_AXI_ACLK,
I_RESET => reset_i,
I_CE    => hold_rd_ce_i,
I_CLEAR => hold_clear_i,
I_SIG   => I_RD_ACK,
O_SIG   => holded_rd_ack_i);
-----------------------------------------------------------------------
```

Starting with the write transfer path, one might notice that it has one extra state than its read counterpart. As stated earlier in paragraph three of section 4.6.1, this is due to the fact that the read transfer does not need an acknowledgment channel, and therefore, does not need an acknowledgement state either. The write address state, named as *wr_addr*, performs the first stage of the AXI protocol presented in fig. 4.6.2 by confirming the address with the assertion of the *s_axi_awready* signal, whilst it asserts the chip select of the intended slave. For clarification, the latter was named *cs_a* because some slaves that use the S00 and S01 interface are dual port, meaning that they can be written to, or read from, by the master using this AXI interface and also by another master entity on their B port.

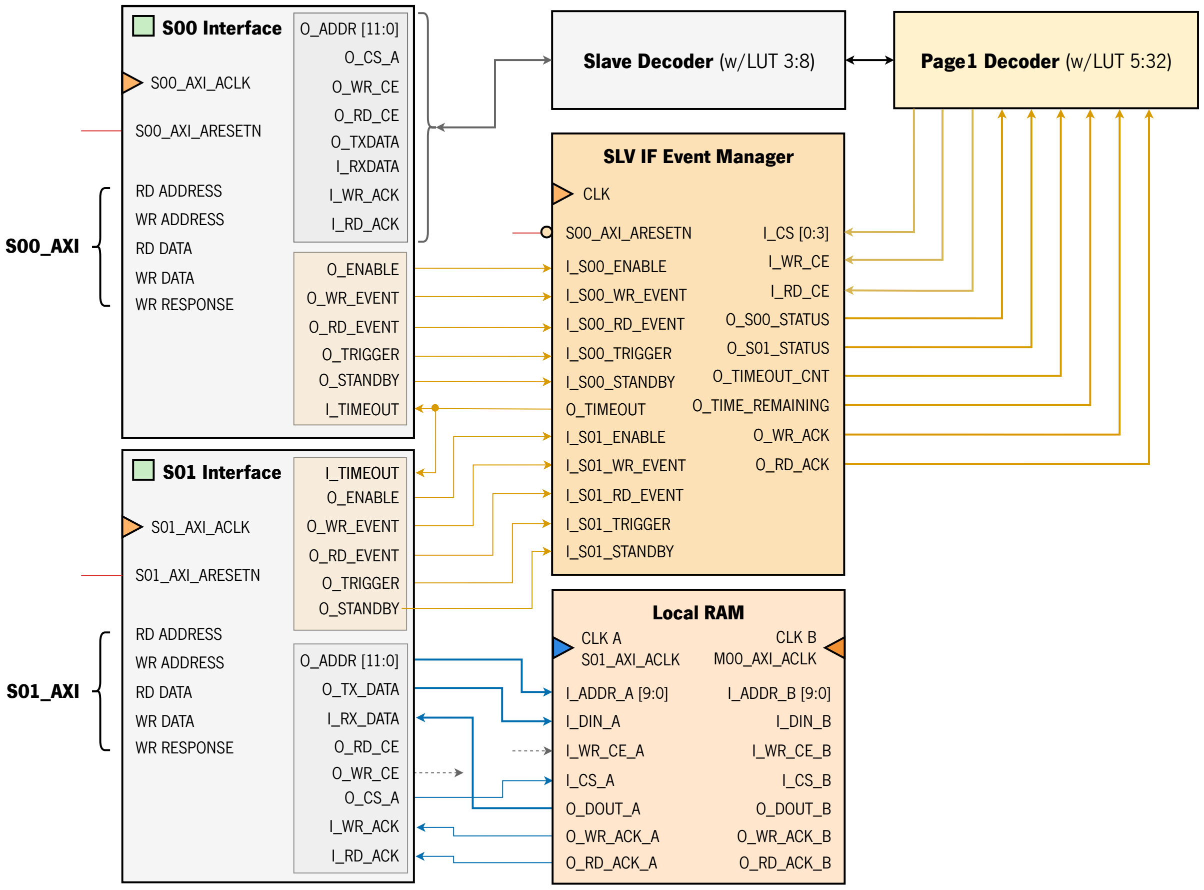


Figure 4.6.3: Connection of the S00 Control and S01 Data Interfaces with the Slave Interface Event Manager and the LRAM.

When the *wvalid* signal is asserted, the state machine moves to the write data state, named *wr_data*. This state corresponds to the write data stage of the AXI protocol in fig. 4.6.2. Concerning this, the state activates the *s_axi_wready* signal to accept the data stipulated by the master, keeps the *cs_a* signal asserted, activates the slave write clock enable, *wr_ce*, to enable the slave writing, and latches the write data in the FF1 flip-flop with the *latch_wdata* signal (refer to fig. 4.6.1 or paragraph five of section 4.6.1). The transition to the next state depends on the logic level of the *bready* signal, i.e., if *bready* is '1' then the state machine proceeds to the *ack* state as normal, but if *bready* is '0' the next state is the write wait one, named *wr_wait*. In the wait state, the FSM enables and triggers the slave interface event manager, that serves both the S00 and the S01 interfaces, to wait a specified number of cycles, sixteen in this case, for the *bready* signal to be asserted. One transitions to the *ack* state if *bready* comes within the specified cycles or if the event manager unit times out, dictated by the *event_timeout* signal. For more details on the slave interface event manager, please refer to section 4.5.4, and for an overview of the S00 Control and S01 Data interfaces' connection with the slave interface event manager, and the LRAM, refer to fig. 4.6.3.

The modules used in the dissertation all have *wr_ce* and *cs* inputs and generate write and read acknowledges with the logic *AND* of *wr_ce* and *cs* after one clock cycle (generic bus). Hence, when one has both the *cs_a* and *wr_ce* signals asserted in the *wr_data* state, the *wr_ack* should come after one clock cycle in the *ack* state or the *wr_wait* state. If the state machine does not need to wait on *bready* one proceeds normally and the *wr_ack* is verified for the transition of *ack* state into the *idle* state. When the FSM has to wait, and one takes in fact the path explained earlier that leads to the *wr_wait* state, the *wr_ack* coming in from the slave must be postponed for the same number of cycles as the wait. This calls for the activation of the *hold_wr_ce* signal in the *wr_wait* state. The signal is fed directly onto the hold logic H0, fig. 4.6.1, that holds *i_wr_ack* until cleared. The HDL of the hold entity can be seen in listing 4.29 and the module's internal architecture is depicted in fig. 4.6.4. The state machine then passes through the *ack* state, where the *bresp* signal is assigned with "00" to indicate an *okay* response, the *cs_a* is maintained active for the final clock cycle, and the *bvalid* signal is asserted as specified by the AXI protocol seen in fig. 4.6.2.

Listing 4.29: Bit Signal Hold Entity (HDL).

```vhdl
sig_d <= I_SIG or sig_q;
---------------------------------------------
SIG_HOLD_FF0 : process (I_CLK)
---------------------------------------------
begin
    if rising_edge(I_CLK) then
        if I_RESET = '1' or I_CLEAR = '1'
            then
            sig_q <= '0';
        elsif I_CE = '1' then
            sig_q <= sig_d;
        end if;
    end if;
end process SIG_HOLD_FF0;
---------------------------------------------
O_SIG <= sig_d;
```

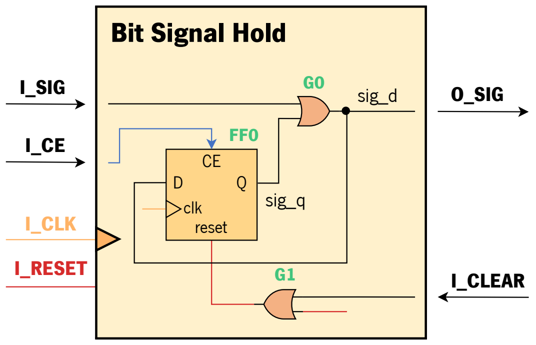


Figure 4.6.4: Bit Signal Hold Entity RTL Design Internal Architecture.

Finally, as forementioned, following the best considered path, i.e., the path that has less latency in clock cycles, the *wr_ack* signal should be active, and the system moves onto the *idle* state. Otherwise, if there was a wait, one should expect the *holded_wr_ack* signal to be '1' and the system still proceeds to *idle*.

Another important part of the state machine design is accounting for the case where the slave fails to send the *wr_ack* or *rd_ack* for some particular reason. Referring to the write transfer, this is resolved by adding another state as an alternative next state for the *ack* state. This state, named *wr_standby*, implies a standby, similar to the one used for waiting on *bready*, but this time for waiting on the *wr_ack*. The only change needed is the specification of the *wr_ack* as the new event. The interface then only gets out of the *standby* state to the *idle* state by the assertion of *wr_ack* within sixteen cycles or the event manager timeout, once again specified by the *event_timeout* signal. Note that the extra state, i.e., the *standby* state, was added because *bvalid* could not be asserted for more than one cycle while waiting on the *wr_ack* signal.

Figure 4.6.5 specifies how an AXI4 lite read transfer behaves. First off, the master sets the address and asserts the *arvalid* signal, to indicate a valid address and the slave must respond with the *awready* signal for address acceptance. Following, the transfer is complete, with the slave setting its data and asserting the *rvalid* signal, since the read does not have an acknowledgment stage.

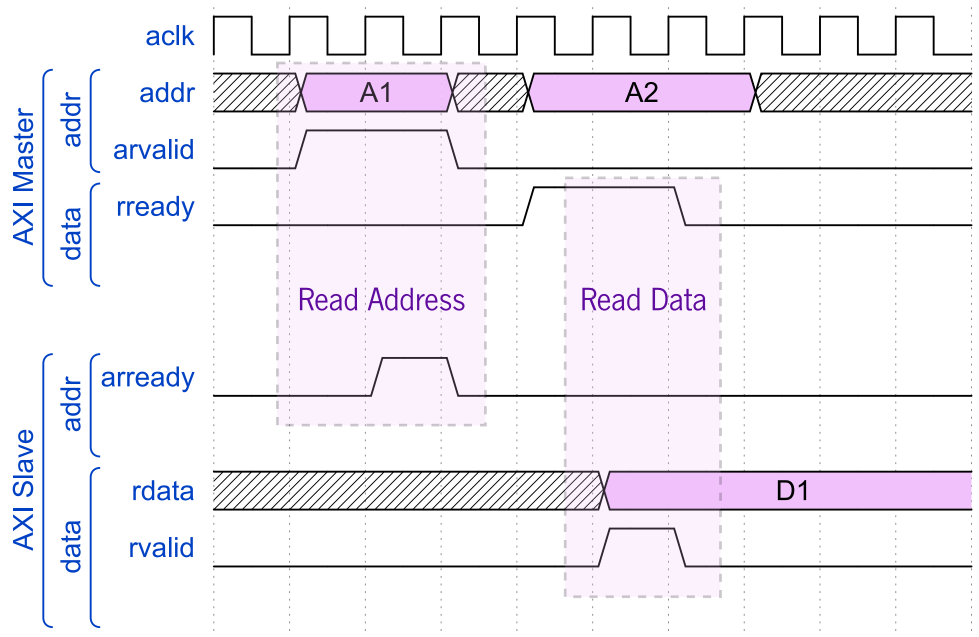


Figure 4.6.5: AXI4 Lite Protocol Waveform: Read Transfer.

As it happened with the write transfer, the AXI4 lite interface's FSM for the read transfer tries to mimic the protocol and stipulate the same stages, i.e., the read address and read data states. The state machine goes into the read address state, named as *rd_addr*, from *idle* if the *arvalid* signal is asserted. When the FSM reaches the *wr_addr* state, it sets *arready* to '1', and enables the read clock enable, *rd_ce*, and slave chip select, *cs_a*, signals. Both of these signals are asserted immediately in the *wr_addr* state because the *rd_ack* must be present in the read data state, and it takes one clock cycle to do so, as explained earlier in the write transfer. The next state after *rd_addr* either is the *rd_data* one if *rready* (coming from master) is asserted, or *rd_wait* if *rready* is not asserted. The *rd_data* state keeps the *cs_a* signal '1' for one more clock cycle, responds with *okay*, i.e., sets *rresp* to "00", latches the read data onto the FF2 flip-flop, fig. 4.6.1, and asserts *rvalid*, to complete the read transfer. The *rready* signal is asserted first in fig. 4.6.5 since the AXI masters tend to be faster than the slaves, even though the protocol flow is from slave to master in the read transfer. The logic behind the read transfer follows the same thought process as the write transfer. For that reason, the event waits on the *rd_wait* and *rd_standby* states also use the slave interface event manager (section 4.5.4) but specify different events to wait on, i.e., the *rready* and *rd_ack* signals, respectively. One extra particularity of the interface is the saving one clock cycle if one leaves the read transfer with *awvalid* and *wvalid* asserted. In this situation, the FSM goes directly to the *wr_addr* state. Note that this still presupposes the verification of the *rd_ack*, *event_timeout*, and *holded_rd_ack* signals, as one can see in fig. 4.6.6. The synchronization flip-flops (FF3 to FF8), fig. 4.6.1, are mainly needed due to the fact that the address, write data, and read data are flopped, FF0, FF1, and FF2, respectively, causing certain datapath signals to fall behind by one clock cycle. Additionally, the local RAM also has a one clock cycle latency to output data on *dout* after an address has been specified, this is also a factor to take into account for synchronization because the AXI4 lite interface module must serve for both the control-oriented S00 interface and also the data-oriented S01 interface with the same datapath and control unit. The HDL of some of the aforementioned flip-flops (FF3 to FF5) are represented in listing 4.30 and listing 4.31, since they all follow the same HDL structure.

Listing 4.30: Flip-Flop Synchronizers: *RVALID* (HDL).

```
-----------------------------------------
SYNC_FF3_FF4: process (S_AXI_ACLK)
-----------------------------------------
begin
   if rising_edge (S_AXI_ACLK) then
       if reset_i = '1' then
            rvalid_Q   <= '0';
            rvalid_Q2  <= '0';
       else
            rvalid_Q   <= rvalid_D;
            rvalid_Q2  <= rvalid_Q;
       end if;
      end if;
end process SYNC_FF3_FF4;
-----------------------------------------
S_AXI_RVALID <= rvalid_Q2;
```

Listing 4.31: Flip-Flop Synchronizers: *WR_CE* (HDL).

```
-----------------------------------------
SYNC_FF5: process (S_AXI_ACLK)
-----------------------------------------
begin
    if rising_edge (S_AXI_ACLK) then
        if reset_i = '1' then
            wr_ce_Q <= '0';
        else
            wr_ce_Q <= wr_ce_D;
        end if;
    end if;
end process SYNC_FF5;
-----------------------------------------
O_WR_CE <= wr_ce_Q;
```

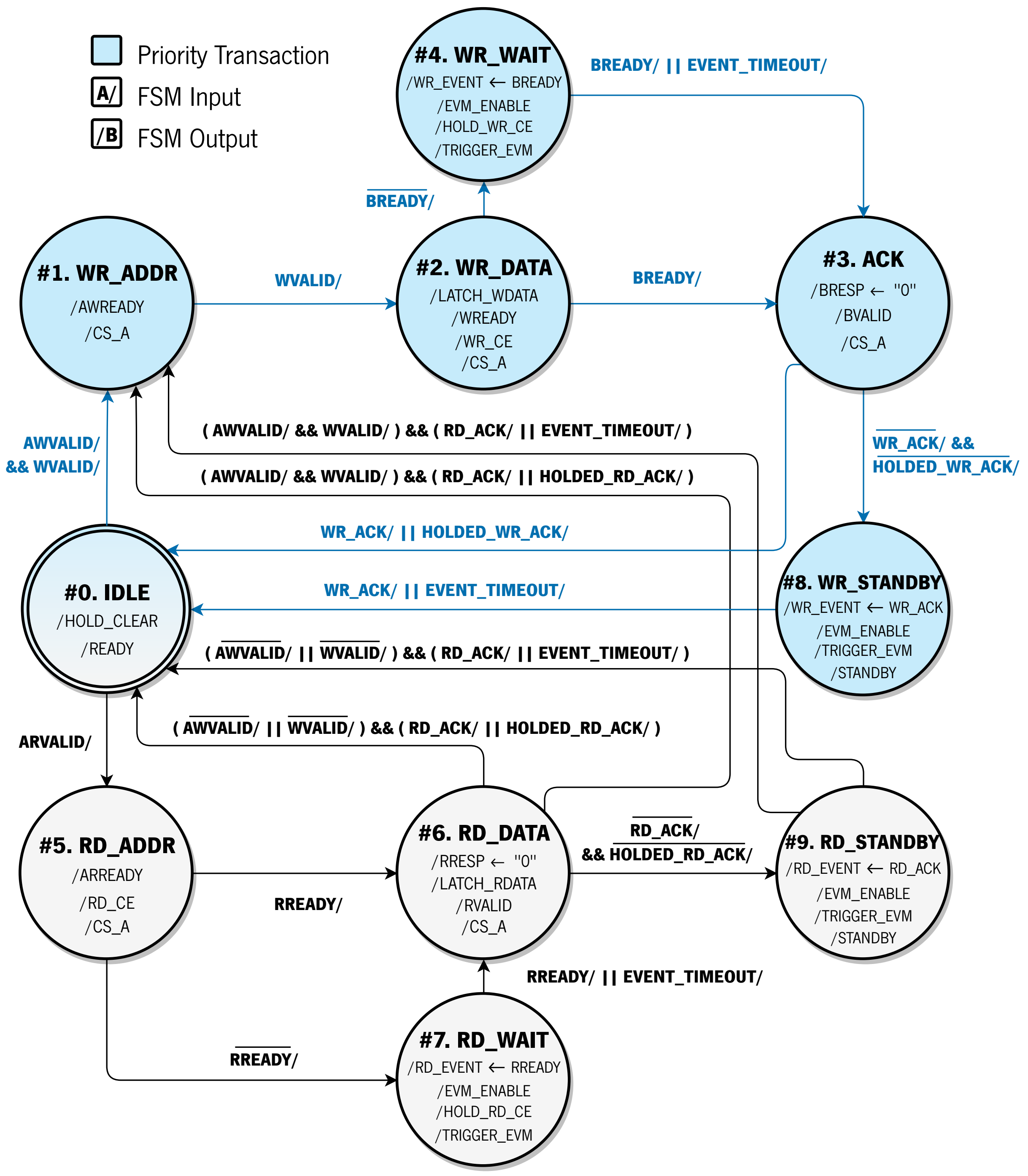

Priority Transaction
FSM Input
FSM Output
#4. WR_WAIT /WR_EVENT ← BREADY /EVM_ENABLE /HOLD_WR_CE /TRIGGER_EVM
BREADY/ || EVENT_TIMEOUT/
BREADY/
#1. WR_ADDR /AWREADY /CS_A
WVALID/
#2. WR_DATA /LATCH_WDATA /WREADY /WR_CE /CS_A
BREADY/
#3. ACK /BRESP ← "0" /BVALID /CS_A
AWVALID/ && WVALID/
(AWVALID/ && WVALID/) && (RD_ACK/ || EVENT_TIMEOUT/)
(AWVALID/ && WVALID/) && (RD_ACK/ || HOLDED_RD_ACK/)
WR_ACK/ && HOLDED_WR_ACK/
WR_ACK/ || HOLDED_WR_ACK/
#0. IDLE /HOLD_CLEAR /READY
WR_ACK/ || EVENT_TIMEOUT/
#8. WR_STANDBY /WR_EVENT ← WR_ACK /EVM_ENABLE /TRIGGER_EVM /STANDBY
(AWVALID/ || WVALID/) && (RD_ACK/ || EVENT_TIMEOUT/)
(AWVALID/ || WVALID/) && (RD_ACK/ || HOLDED_RD_ACK/)
ARVALID/
#5. RD_ADDR /ARREADY /RD_CE /CS_A
RREADY/
#6. RD_DATA /RRESP ← "0" /LATCH_RDATA /RVALID /CS_A
RD_ACK/ && HOLDED_RD_ACK/
#9. RD_STANDBY /RD_EVENT ← RD_ACK /EVM_ENABLE /TRIGGER_EVM /STANDBY
RREADY/ || EVENT_TIMEOUT/
RREADY/
#7. RD_WAIT /RD_EVENT ← RREADY /EVM_ENABLE /HOLD_RD_CE /TRIGGER_EVM

Figure 4.6.6: AXI4 Lite Interface Optimized Finite State Machine (FSM), HDL in listing A.2.

4.6.2 BFM Verification

The AXI Verification IP (VIP) is an IP by AMD-Xilinx that allows system simulation with a SystemVerilog interface. To use the IP in an environment like Vivado (used in the dissertation) the project must ensure Verilog wrappers for designs, connect the IP to a top-level module with an AXI interface, and build

SystemVerilog testbenches. The dissertation modules that use standard AXI interfaces use this type of verification to test for correctness and regression.

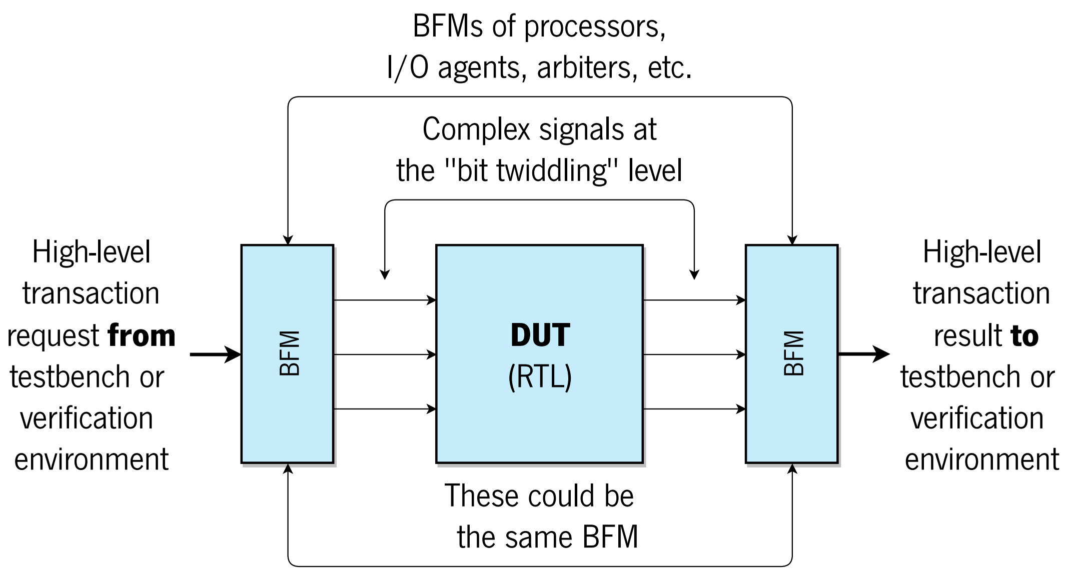

BFMs of processors,
I/O agents, arbiters, etc.
Complex signals at
the "bit twiddling" level
High-level transaction request **from** testbench or verification environment
BFM
DUT
(RTL)
BFM
High-level transaction result **to** testbench or verification environment
These could be the same BFM

Figure 4.6.7: Using Verification IP in the form of BFMs, adapted [37].

The AXI VIP can be seen as a Bus Functional Model, and therefore perform BFM verification. A BFM emulates the behavior of a device at the bus-interface level, in this case an AXI one. Regarding this, the BFM can generate or accept transactions such as writes and reads, as specified by the design's testbench [37, 27]. Figure 4.6.7 depicts the usage of a certain VIP to perform bus transactions. The AXI VIP used has three modes of operation, thus it can be used as an AXI bus master, as an AXI bus slave, and also a pass-through protocol checker between a AXI master and slave.

Concerning the aforementioned verification, one tested the interface in Vivado using the AXI verification IP as a BFM, fig. 4.6.8. The tests represented consisted of writing and reading four times through the S00 interface into the interface registers module, section 4.5.5, and two last reads on the S00 interface event manager's status and timeout counter registers, section 4.5.4, to inquire the interface state, and timeout count, respectively.

Figure 4.6.10 represents the behavior of the interface when *bready* is set to '0'. For this purpose, one uses an internal signal that is always '0' and assigns it to the interface instead of *bready*. With these changes, the interface acts as expected, entering the *wr_wait* state in the write transfers, whilst it still performs all the transfers to completion. In this case, the behavior of the read transfers follow the normal path of operation, not entering the *rd_wait* state since they are not dependent on *bready*. The same test was performed for the read transfers, and they in fact entered the *rd_wait* state if *rready* fails to be

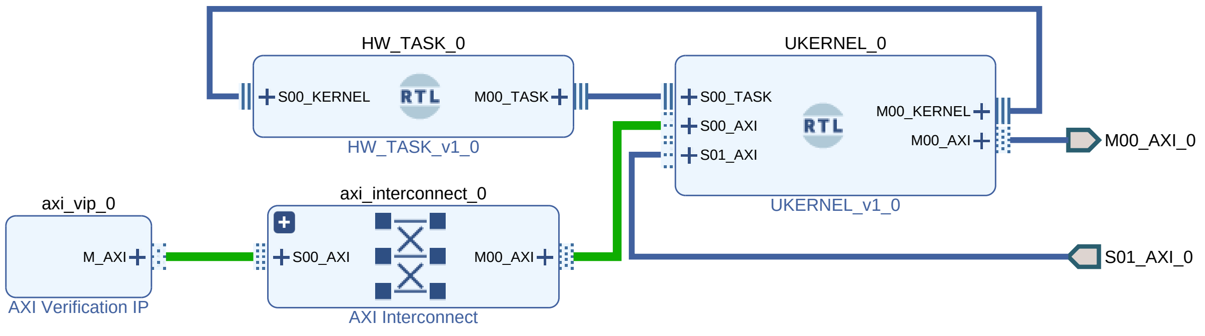


Figure 4.6.8: AXI VIP Simulation – S00 Control Interface Test: Block Design.

asserted. As it is possible to see in fig. 4.6.10, the interface status after entering the *wr_wait* state and occurring an event manager timeout is "fe11dead", i.e., the interface fell dead waiting on *bready* to arrive, and also the timeout counter is "00000004", meaning that the interface timed out four times, which is comprehensible since the *bready* signal was never asserted. Note that the third read performed on the interface registers module is targeting the read-only register regarding the depth of the LRAM. For this reason, the read always returns "00000400" in hexadecimal, or "1024" as an unsigned decimal. The write on read-only registers performs to completion in terms of the AXI4 lite protocol, but acts as a dummy write for the slave in question.

Figure 4.6.11 represents the interface's behavior when one removes the *wr_ack* and *rd_ack* connection from slave to the module. In this case, the interface's *bready* signal is still kept as '0'. As it is possible to perceive, the interface enters the *wr_wait* state once again, with *bready* missing, and now enters *wr_standby* and *rd_standby* as well, with the missing of the write and read acknowledges, respectively. Since the latest interface error is bad interface, i.e., acknowledges not asserted, the S00 interface status is "bad1face", even though there were "fe11dead" cases before. The timeout counter register shows the value "0000000C" since the interface timed out twelve times, i.e., four times with the failure of *bready* on the write transfers, another four with the failure of the *wr_ack* in the write transfers as well, and the last four with the failure of the *rd_ack* in the read transfers pertaining to the interface registers module.

The same test was repeated for the S01 Data interface, writing and reading four times to and from the local memory, respectively. In this test the 4K address space given by the twelve bits of the AXI VIP's "virtual" address (minimum possible) was split between the two interfaces, i.e., giving 2K to the S00 Control interface and 1K to the S01 Data interface, so one could perform the S01 Data functional tests in a similar manner as the S00 Control ones. In this particular case, it did not matter being bounded to a

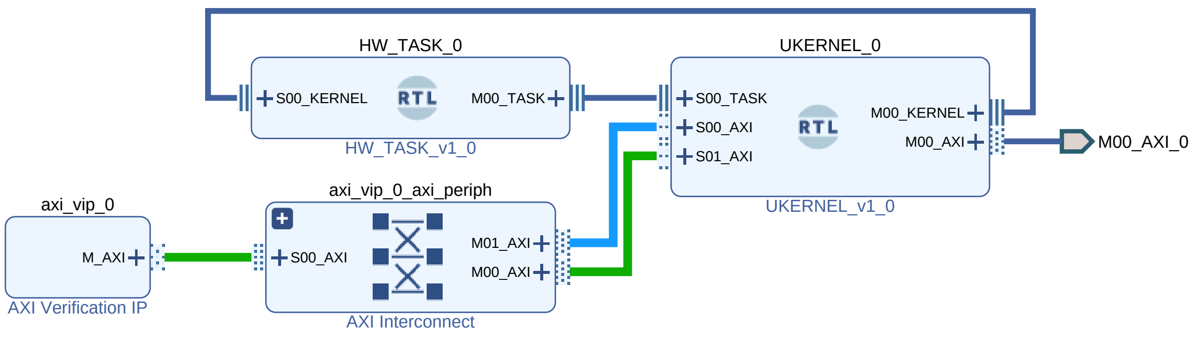


Figure 4.6.9: AXI VIP Simulation – S01 Data Interface Test: Block Design.

LRAM with 1K words of depth since the actual size stipulated for the local memory was 1K words as well, even when using sixteen bits for the address, which allows for the expansion of the LRAM up to 64K words depth. The S00 Control interface was kept as writable/readable by the VIP, since one needed to access the slave interface event manager to inquire the interface state and timeout count, as forementioned in earlier paragraphs. As it is possible to see in fig. 4.6.13, the test results were as expected, reading the same four values written to the memory. As for the slave interface event manager's information, the two reads referent to the S00 and S01 status revealed no errors, but future implementations of the S00 and S01 interface should improve the one clock cycle latency associated with the standby state, and that originates some unwanted timeouts.

4.6.2.1 Limitations

The use of synchronizers to assure data correctness introduced a limitation to the interface. Since the *wr_ce* and *cs_a* need to be delayed by one clock cycle, the *wr_ack* and *rd_ack* of the slave module also arrives one cycle late. This particular situation, causes the module's state machine to go from the *ack* state to the *wr_standby* state and then to *idle*, for the write transfers, and from the *rd_data* state to *rd_standby* state and then to *idle*. This means that the interface enters the standby mode, but the *wr_ack* or *rd_ack* arrive exactly at the beginning of the *standby* state, making the FSM go to *idle* in the next clock cycle. Future improvements on the interface should focus on mitigating this latency of one clock cycle, when there are no waits. The best scenario for the interface state machine can be seen in fig. 4.6.12. In this situation, the reads performed on the S00 status register and the timeout count register of the slave interface event manager, section 4.5.4, return “00” meaning the interface did not encounter any errors and, thus, there were no timeouts.

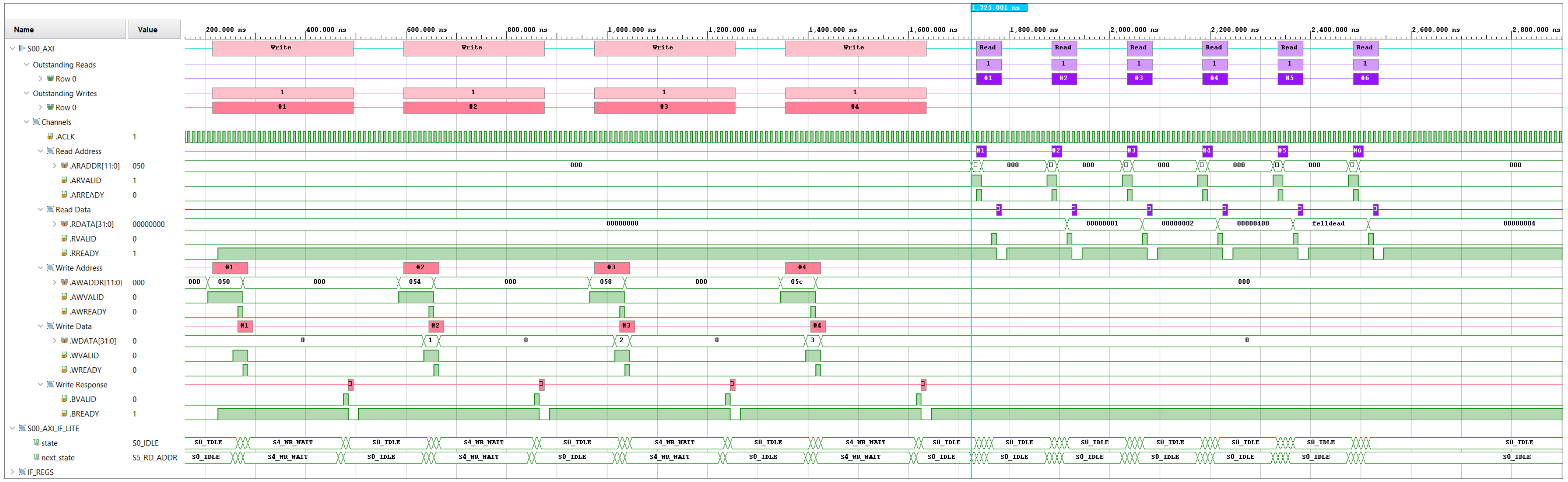


Figure 4.6.10: AXI4 Lite Interface Wave Diagram (Vivado VIP on S00 Control): BREADY Timeout (Fell Dead) Scenario.

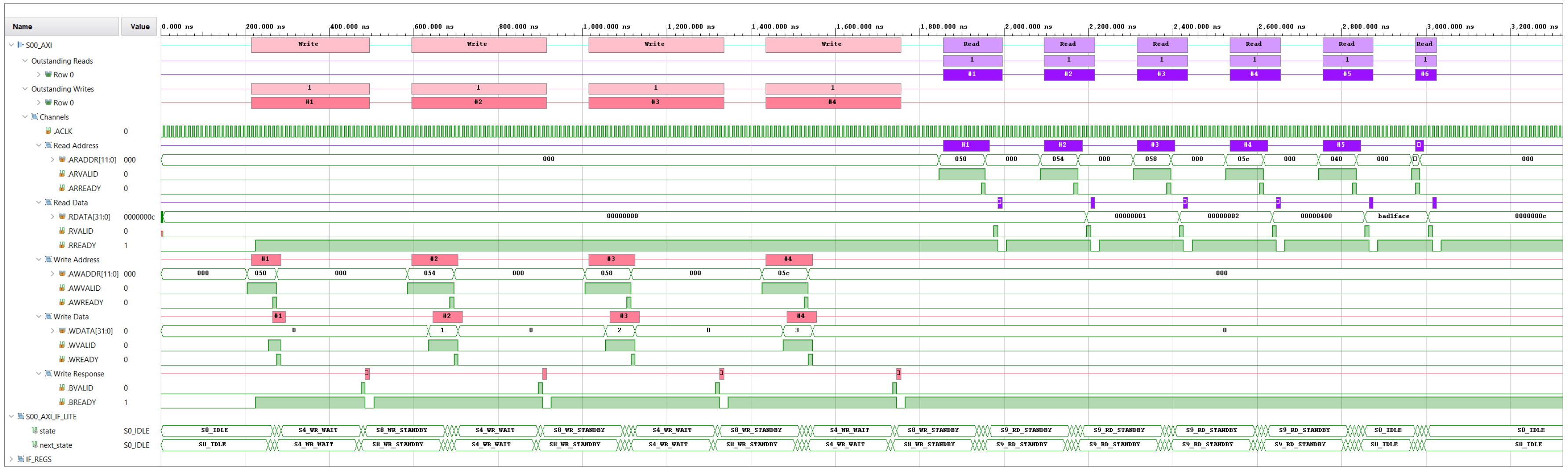


Figure 4.6.11: AXI4 Lite Interface Wave Diagram (Vivado VIP on S00 Control): BREADY Timeout and WR_ACK and RD_ACK Not Asserted (Bad Interface) Scenario.

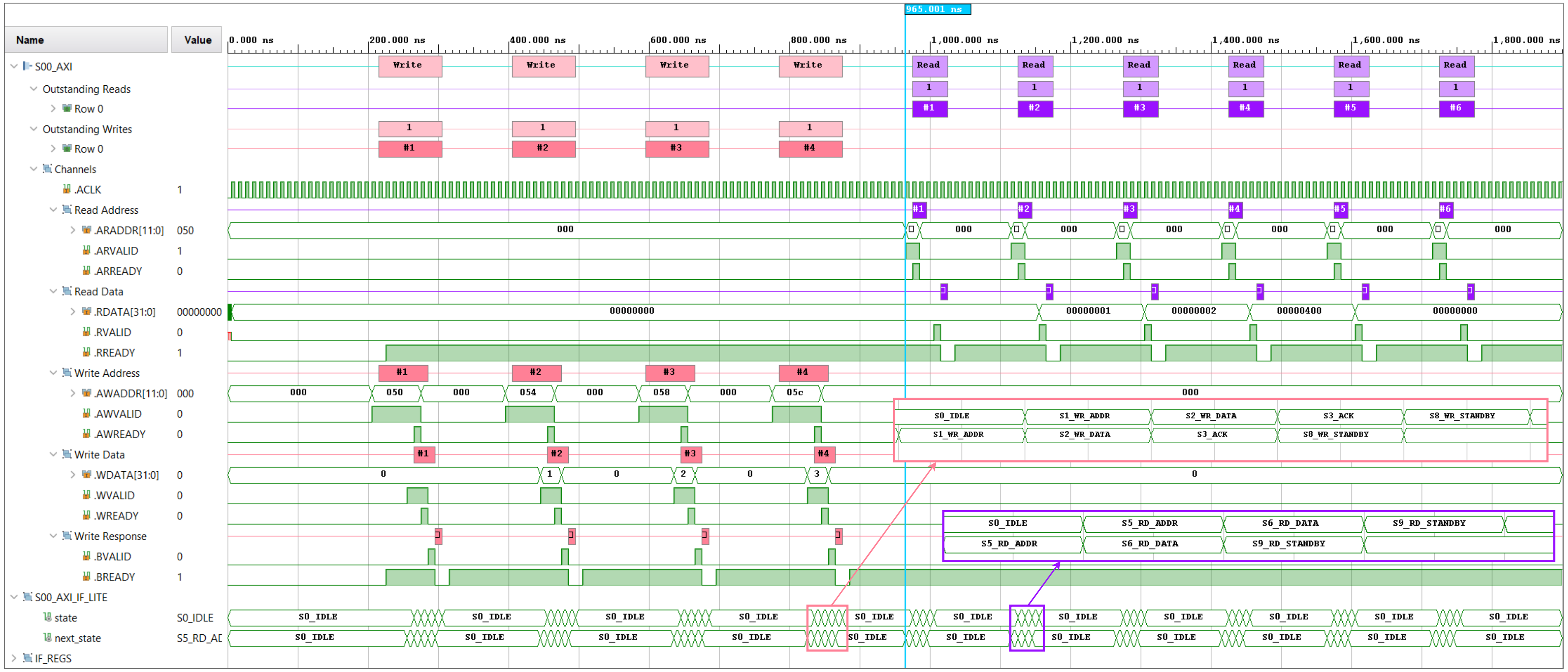


Figure 4.6.12: AXI4 Lite Interface Wave Diagram (Vivado VIP on S00 Control): Best Scenario.

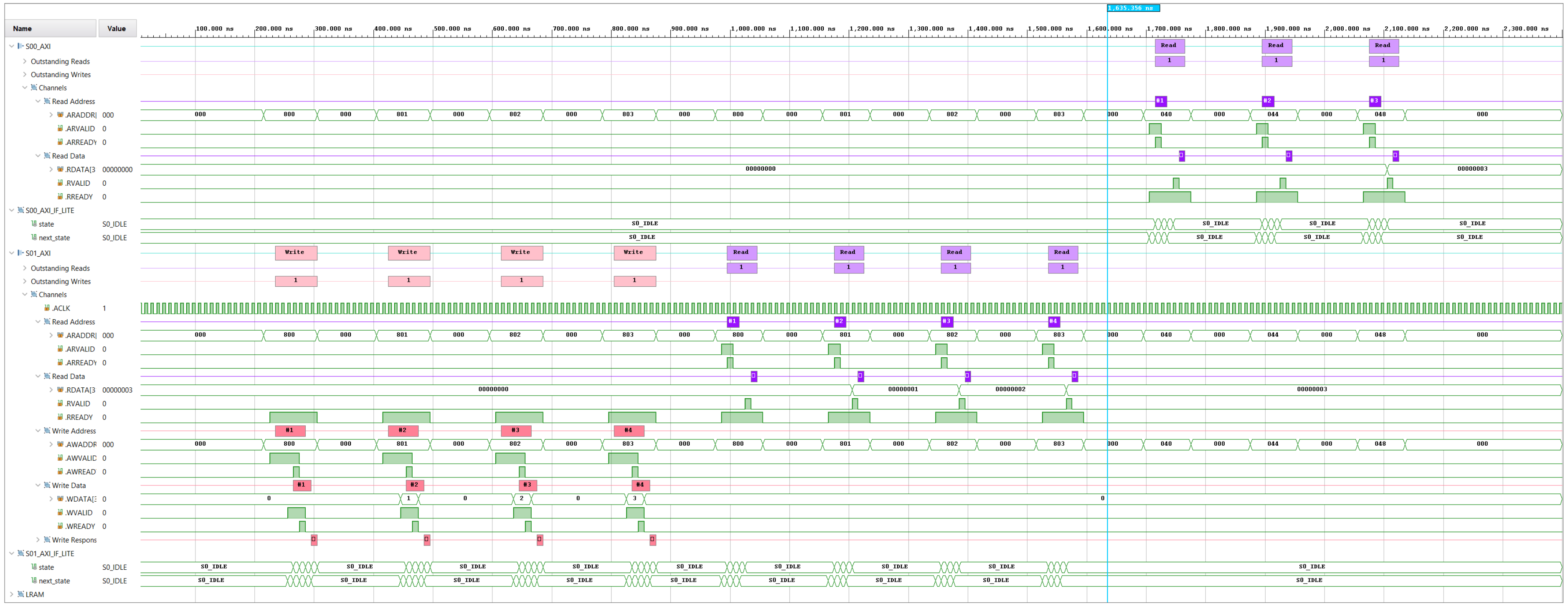


Figure 4.6.13: AXI4 Lite Interface Wave Diagram (Vivado VIP on S01 Data w/ information check on S00 Control): Best Scenario.

4.6.3 S00 Control Interface

The S00 Control establishes a control channel for the host system to access the accelerator's register space. This interface leverages the resource management capabilities described in section 4.4 to divide the register space into multiple pages. Since one is designing a microkernel there are fewer resources when comparing to HAL-ASOS, so the maximum number of pages was established at four. Two of them are not fully mapped (zero and one) and there is also two other pages (two and three) completely unmapped to account for future microkernel expansion in terms of accessible registers. This can be seen in fig. 4.6.14.

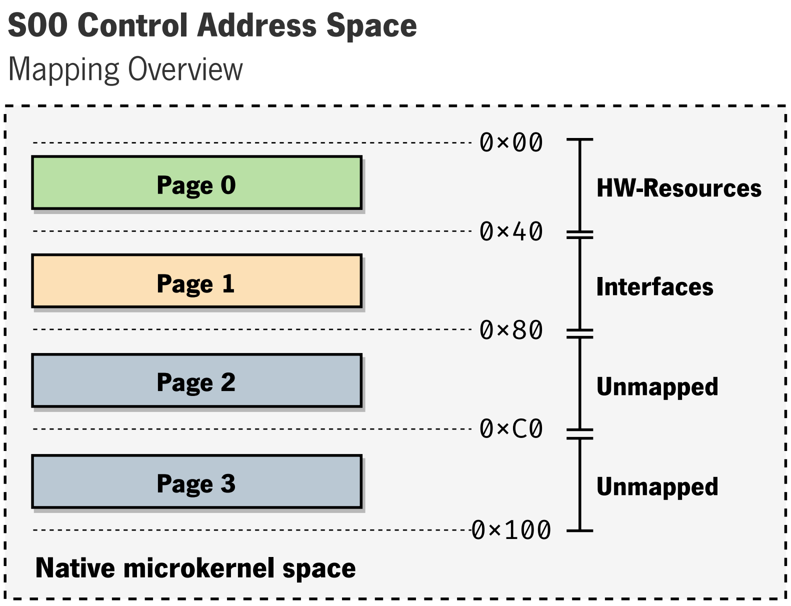


Figure 4.6.14: Native Microkernel: S00 Control Address Space.

The table 4.2 describes the page zero of the accelerator model, and this page is dedicated to the microkernel resources that do not relate to the interfaces. As it is possible to see, the first register mapped is the microkernel control register, belonging to the control register module (*c.f* section 4.2.1), at word offset 0×00 and byte offset 0×00. To allow the write of control words from the host system side and reads to check if the word was correctly written, this particular register was deemed as writable/readable. Moreover, still in the same module (*c.f* section 4.2.1), one has the microkernel status register which is updated internally according to the current state of the kernel. The latter is mapped at word offset 0×01 and byte offset 0×04, and deemed as read-only due to the internal update performed. Note that, each register that is not writable will still carry on an AXI transaction write to completion and generate a write acknowledge, but the write will not have an effect on the register's contents. Additionally, the byte offsets aforementioned are stipulated like this since the microkernel is a 32-bit machine by default. Pertaining to the same page (zero), one also has the registers that refer to the mutexes (*c.f* section 4.5.1) and LINTC (*c.f* section 4.5.2) resources. The local mutex maps its data register at word offset 0×02 and byte offset 0×08, and it is write-only. Each write-only register also carries through the AXI reads to completion but returns the 32-bit word "1facefff" to identify a not permitted access. Following, comes the local mutex status register (read-only) at word offset 0×03 and byte offset 0×0C. Likewise, the system mutex also has a write-only data register at word offset 0×04 and byte offset 0×10, and its read-only status register at word offset 0×05 and byte offset 0×14. For more information on these registers refer to section 4.5.1. The LINTC maps its control register at word offset 0×06 and byte offset 0×18. For the same reasons as the microkernel registers, the LINTC stipulates the control register to be writable/readable and the status register to be read-only. As mentioned before, page zero still has room for eight more registers, since the number of 32-bit words per page was set at sixteen.

Table 4.2: Page Zero Internal Register Mapping.

S00 Control					
Page Offset	**Register**	**Word Offset**	**Byte Offset**	**WR**	**RD**
0x00 (Page 0) 🟩	Microkernel Control	0x00	0x00	●	●
	Microkernel Status	0x01	0x04	○	●
	Local Mutex ID Data	0x02	0x08	●	○
	Local Mutex Status	0x03	0x0C	○	●
	System Mutex ID Data	0x04	0x10	●	○
	System Mutex Status	0x05	0x14	○	●
	Local Interrupt Ctrl. Control	0x06	0x18	●	●
	Local Interrupt Ctrl. Status	0x07	0x1C	○	●

● Access Permitted. ○ Access Not Permitted.

Note: ○ Still performs the generic bus handshake.

Analyzing page one, table 4.3, one can notice that it maps the registers related to the slave and master interfaces. Thus, the first four registers mapped are included in the slave interface event manager module. At word offset 0×00 and byte offset 0×00 one encounters the S00 Control status register, which gives the current state of the interface, i.e., "00000000" indicating normal, "fe11dead" indicating the failure of AXI signals, or "bad1face" indicating the failure of slave signals. In a similar way, one has the S01 Data status register, for the same reasons, mapped to word offset 0×01 and byte offset 0×04. These registers are read-only since they both represent a status, and, therefore, are updated internally. The next read-only register represents the total timeout count of both interfaces (S00 Control and S01 Data), mapped to word offset 0×03 and byte offset 0×08. The final register pertaining to this module (*c.f* section 4.5.4) is the time remaining register (read-only as well), which is useful for checking how much time is left until one of the interfaces reaches timeout. The latter is at word offset 0×03 and byte offset 0×0C. The next four registers belong to the interface registers module (*c.f* section 4.5.5). The first one at word offset 0×04 and byte offset 0×10 represents the SYSRAM base address, and it is useful since it allows the host system to configure the base of the allocated memory for the accelerator. The KFIFO base offset register is useful for the host system to stipulate the beginning of the KFIFO area of the SYSRAM. This means that even tho the allocated memory starts at a certain position, the KFIFO might not start in the same position, with this it is also possible to configure that. The KFIFO base register is at word offset 0×05 and byte offset 0×14. Succeeding, one has the SYSRAM data register, which indicates the starting position for the data-exchange area of the SYSRAM. For the same reasons, this parameter is configurable since the data area might change position depending on various factors. The SYSRAM data register is mapped at word offset 0×06 and byte offset 0×18. All of these three registers of the interface registers module are writable/readable to ensure correct configuration. Additionally, one has the LRAM depth register, which is read-only and indicates to the host the fixed depth (in words) of the local memory. The latter is mapped at word offset 0×07 and byte offset 0×1C. Finally, page one has the accelerator's clock counter control register, and the clock counter's words, mapped to word offset 0×08, 0×09, and 0×0A, respectively.

4.6.4 S01 Data Interface

The S01 Data interface establishes a data-focused channel between the host system and the accelerator's local memory (LRAM). In the accelerator's case there is not a ZCU unit and, thus, the host-system transactions to local memory are limited to word-rated ones. This happens since S01 Data was stipulated as an AXI4 lite interface, which, by itself, only allow single-word transfers. So, for the system co-designer are available two options to interact with the local memory, (1) write words to memory and read them in the hardware task, or (2) store words in memory through dedicated syscalls in the hardware task, and then read them in the host system side through S01. For the contents of this memory to be reflected in the host system's and the accelerator's shared memory, one has the option to store data read from the LRAM locally, in a buffer for example, and then write it to main memory in burst format through M00 System

Table 4.3: Page One Internal Register Mapping.

S00 Control					
Page Offset	**Register**	**Word Offset**	**Byte Offset**	**WR**	**RD**
0x40 (Page 1) 🟧	S00 Control Status	0x00	0x00	○	●
	S01 Data Status	0x01	0x04	○	●
	Timeout Count	0x02	0x08	○	●
	Time Remaining	0x03	0x0C	○	●
	SYSRAM Base Address	0x04	0x10	●	●
	KFIFO Base Offset	0x05	0x14	●	●
	SYSRAM Data Base Offset	0x06	0x18	●	●
	LRAM Depth	0x07	0x1C	○	●
	Acc. Counter Control	0x08	0x20	●	●
	Acc. Counter Word Zero	0x09	0x24	○	●
	Acc. Counter Word One	0x0A	0x28	○	●

● Access Permitted. ○ Access Not Permitted.

Note: ○ Still performs the generic bus handshake.

with the appropriate system calls. With all this, one means that the local memory acts as a cache for the accelerator, but there is not a unit with DMA-like capabilities (ZCU) that transfers the contents from one memory to another independently. Additionally, the local memory was stipulated at 1K words of depth, but since the S01 interface uses a 16-bit address the memory can go up to 64K of depth, depending on application demands, with a machine word of 32 bits by default.

4.6.5 M00 System Interface

The M00 System interface establishes a master channel between the accelerator and the allocated memory region (deemed as shared memory) of the host system. This interface does not have complex logic, unlike its slave counterparts (S00 Control and S01 Data), because it only forwards the microprogram control signals associated with MBUS syscalls for write and read, with the top-level's master outputs. In the case of the dissertation's accelerator the master channel is used to exchange control commands (*c.f* section 5.1) and chunks of data from the hardware task entity to main memory. For this there is a specific region for command and return placement, and also a region for the placing the data.

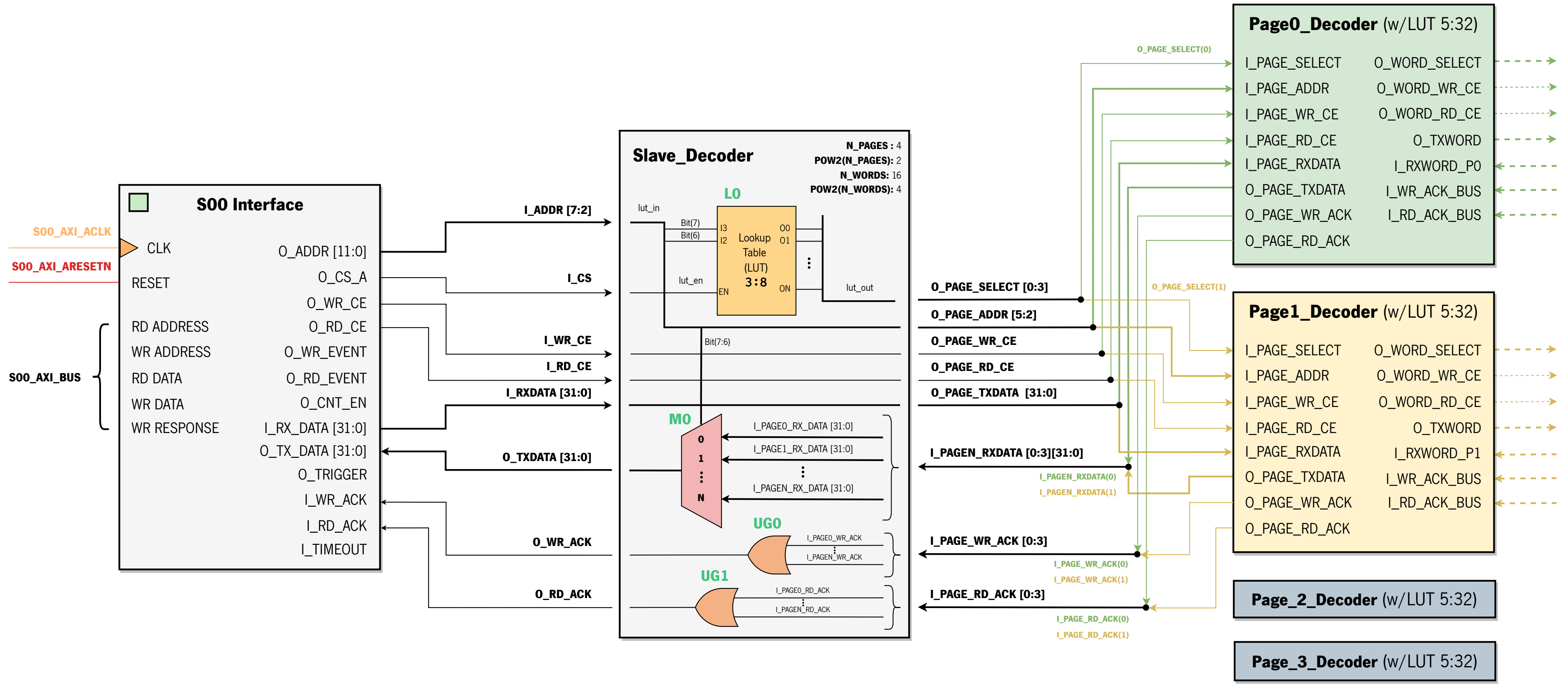


Figure 4.6.15: S00 Control Two-Level Address Decoding Overview.

5. Architecture Extensions

As stated in section 1.2, after establishing the supporting microkernel architecture and understanding its limitations, it is now possible to start improving the architecture by adding functionalities and resolving some issues. This chapter approaches the proposed architecture extensions to the microkernel, which include the creation of memory mechanisms for message/command exchange (*c.f* section 5.1) and the leverage of complex procedure scalability (*c.f* section 5.2). Finally, the chapter culminates into the explanation of how these extensions and some other additions contribute to a diversity-driven hardware task capable of coping with diversity, i.e., adapting through reconfiguration (*c.f* section 5.3).

5.1 KIVIO-Extended Features

One of the goals of the dissertation work was to refactor the HAL-ASOS hardware accelerator model, and that also included investigating new a way to perform the hardware task's extended features, *c.f* section 4.3. Transforming the HAL-ASOS version four M00 System interface from AXI4 lite to AXI4 opens up the possibility of not only making write and read burst transfers to the allocated memory section pertaining to data, but also utilize another section of the allocated memory, KFIFO, to implement a command exchange channel through shared memory. For this, KIVIO maps a Linux kernel FIFO element onto specific main memory positions. Regarding both the software and the hardware point of view, this originates the structure on the left-hand side of fig. 5.1.1. The structure has two main regions: (1) a region concerning arguments, i.e., intended for the hardware to place commands, and another (2) region for the hardware to retrieve the results of extended system call execution, deemed as returns. Observing closely, one can see that the arguments section has five elements that map directly to a KFIFO struct in kernel space. It is important to note that both of the sections forementioned behave as circular FIFOs regarding arguments and returns, respectively. Thus, the *args.in* represents the circular FIFO's write pointer and dictates the *args* block number where the hardware will place the 64-bit command. For the same reasons the *args.out* field represents the read pointer and indicates the block number to read from the host system side by the

KIVIO framework. The *args.mask* is used to make the writing and reading of arguments circular, meaning that each operation applies the mask. In this example, the mask is three because one has four blocks of arguments data. Ultimately, the *args.data* field is a pointer to the starting block of *args*, which means that one only writes something to a *args* block if the writing is done by adding *args.data* to *args.in* times two, since one is dealing with 64-bit command words. The same principles apply to the returns section with also five elements pertaining to the returns circular FIFO, but with eight return entries because the system works with return words of 32 bits.

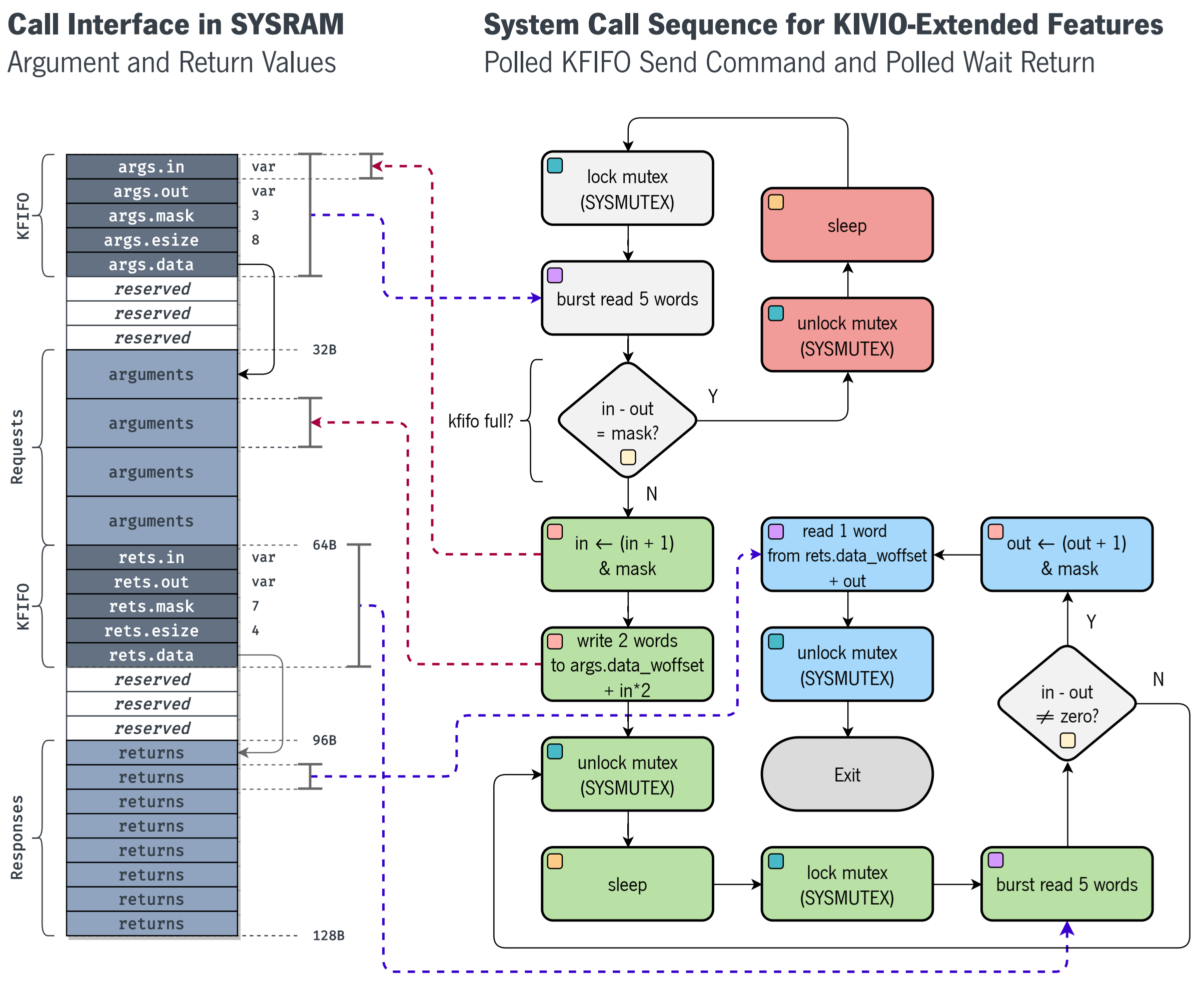


Figure 5.1.1: Composite system call for executing KFIFO-extended features in a polling manner (SYSRAM); The figure on the left represents the memory-mapped kernel FIFO that provides the call interface for extended composite syscalls; The figure on the right depicts the sequence of simpler system calls needed to perform a polled send command and a polled wait for return on the interface. Memory representation in table 5.1.

Then, it is the responsibility of the accelerator to comply with the argument and return bitfields established by KIVIO in its packages and construct the appropriate command depending on the system call performed in the hardware task entity. This can be seen in listing A.4. This command must then follow a standard number of steps to be placed in a *args* block, and another number of steps must be followed to retrieve the return as well. The process of placing a command and recollect its return both in a polling manner is described by the second part of fig. 5.1.1 in the right-hand side. This combination of steps is performed by the kernel of the accelerator by dividing a certain extended user-level procedure into multiple kernel-level procedures of smaller size. Each one of these kernel procedures wraps around a microcoded syscall, *c.f* section 4.2.3, and establishes its arguments and returns. The hardware was developed in a way that every extended system call that uses the mapped KFIFO always uses the same kernel-level procedures and only changes the 64-bit command constructed, e.g., the extended syscall for opening a file is the same as the one for closing it in terms of procedures, the only thing that changes is its ID and, therefore, the 64-bit command constructed upon it, *c.f* listing 5.1.

Listing 5.1: User Package: File Subscribe and Unsubscribe Example (HDL).

```vhdl
-- MORE
-----------------------------------------------------------------------------------------
procedure polled_kernel_bounded_file_subscribe(
    signal kernel_call               : out kernel_call_t;
    signal kernel_response           : in kernel_response_t;
    constant hint_cookie             : in natural;
    constant message_flags_events    : in std_logic_vector(C_KIVIO_FLAGS_WWIDTH-1 downto 0);
    constant wcount_whence           : in natural;
    constant woffset                 : in natural range 0 to 2**C_KIVIO_WOFFSET_WWIDTH-1;
    signal returned_cookie           : out natural) is
-----------------------------------------------------------------------------------------
begin
    kernel_bounded_safe_kfifo_send_command_wait_return(kernel_call,kernel_response,
        KI_SUBSCRIBE,hint_cookie,message_flags_events,wcount_whence,woffset);
    returned_cookie <= to_integer(unsigned(kernel_response.procedure_return(
        KIVIO_RET_COOKIE'RANGE)));
    ---
end procedure polled_kernel_bounded_file_subscribe;
-----------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------
procedure polled_kernel_bounded_file_unsubscribe(
    signal kernel_call               : out kernel_call_t;
    signal kernel_response           : in kernel_response_t;
    constant hint_cookie             : in natural;
    constant message_flags_events    : in std_logic_vector(C_KIVIO_FLAGS_WWIDTH-1 downto 0);
    constant wcount_whence           : in natural;
    constant woffset                 : in natural range 0 to 2**C_KIVIO_WOFFSET_WWIDTH-1) is
-----------------------------------------------------------------------------------------
begin
    kernel_bounded_safe_kfifo_send_command_wait_return(kernel_call,kernel_response,
        KI_UNSUBSCRIBE,hint_cookie,message_flags_events,wcount_whence,woffset);
    ---
end procedure polled_kernel_bounded_file_unsubscribe;
-----------------------------------------------------------------------------------------
-- MORE
```

By analyzing the sequence of kernel-level system calls more deeply, one can notice that it always starts with the lock of the mutex referent to the SYSRAM, i.e., the system mutex. This prevents the occurrence of race conditions for the resource when executing the next steps, which access the allocated memory, listing 5.2.

Listing 5.2: Kernel Package: KFIFO Send Command Wait Return - System Mutex Lock (HDL).

```
-----------------------------------------------------------------------------------------
-- kernel safe (mutex-protected) kfifo polled send command (and polled wait return)
-----------------------------------------------------------------------------------------
procedure kernel_safe_kfifo_send_command_wait_return(
    signal kernel_call       : out kernel_call_t;
    signal kernel_response   : in kernel_response_t;
    signal sched_progress    : in std_logic_vector(C_SCHEDULER_PROGRESS_WIDTH-1 downto 0);
    signal kfifo_args_buffer : inout kfifo_arg_array_t;
    signal kfifo_rets_buffer : inout kfifo_ret_array_t;
    signal timeout           : out boolean;
    signal remaining_time    : out natural;
    constant timeout_value   : in natural;
    signal args_in_masked    : inout std_logic_vector(C_MACHINE_WIDTH-1 downto 0);
    signal rets_out_masked   : inout std_logic_vector(C_MACHINE_WIDTH-1 downto 0);
    signal return_rcvd       : out std_ulogic;
    signal procedure_done    : out std_logic;
    signal mutex_status      : out std_logic_vector(C_MACHINE_WIDTH-1 downto 0);
           data              : in std_logic_vector(C_MESSAGE_WIDTH-1 downto 0);
           read_return       : out std_logic_vector(C_MACHINE_WIDTH-1 downto 0);
    signal kfifo_base_offset : in std_logic_vector(C_MACHINE_WIDTH-1 downto 0)) is
-----------------------------------------------------------------------------------------
-- MORE (Variables)
begin
    return_rcvd <= '0';
    case to_integer(unsigned(sched_progress)) is
        ---------------------------------------------------------------------------------
        when C_B_KFIFO_SYSMUTEX_LOCK | C_B_KFIFO_RET_PATH_SYSMUTEX_LOCK =>
            system_mutex_lock(kernel_call,kernel_response);
        ---------------------------------------------------------------------------------
        -- MORE
```

Firstly, one burst reads the first five elements of the arguments FIFO and stores them in a local buffer in the hardware. This is only possible since the M00 System from HAL-ASOS version four was migrated to AXI4, the same would be possible to do with a AXI4 lite interface in M00 System, but that would incur in dividing a read burst syscall into five single-word read system calls, and ultimately would result in more memory footprint associated with storing them. More on that later in section 5.1.1.

Listing 5.3: Kernel Package: KFIFO Send Command Wait Return - Read Argument Fields (HDL).

```
        -- MORE
        ---------------------------------------------------------------------------------
        when C_B_KFIFO_READ_ARGUMENTS =>
            kernel_call.enable_index <= '1';
            kernel_call.inc_index <= '1';
            mbus_word_read_burst(kernel_call,kernel_response,C_KFIFO_ARG_WIDTH,to_integer(
                unsigned(kfifo_base_offset)) + (C_KFIFO_ARGS_BASE_OFFSET*(C_MACHINE_WIDTH
                /8)),read_args);
            kfifo_args_buffer(kernel_response.index) <= read_args;
        ---------------------------------------------------------------------------------
        -- MORE
```

By having the argument fields stored, it is possible to subtract the *args.out* field from the *args.in* to know if the argument's KFIFO is full or not. This step is also associated with the generation of an interrupt request from PL-to-PS to signal KIVIO of a new command. The IRQ is done before of writing the command to memory since the interrupt source needs some cycles to generate an interrupt request (*c.f* section 4.5.2) and reach the host system. Typically, enough time to be done before. These two steps can be seen in listing 5.4.

Listing 5.4: Kernel Package: KFIFO Send Command Wait Return - Check Argument KFIFO Status and Generate IRQ (HDL).

```
-- MORE
----------------------------------------------------------------------------------
 when C_B_KFIFO_FULL_TEST =>
    kernel_call.enable_index    <= '0';
    kernel_call.inc_index       <= '0';
    kfifo_in    := kfifo_args_buffer(C_KFIFO_ARGS_IN_OFFSET);
    kfifo_out   := kfifo_args_buffer(C_KFIFO_ARGS_OUT_OFFSET);
    kfifo_mask  := kfifo_args_buffer(C_KFIFO_ARGS_MASK_OFFSET);
    kfifo_esize := kfifo_args_buffer(C_KFIFO_ARGS_ESIZE_OFFSET);
    kfifo_data  := kfifo_args_buffer(C_KFIFO_ARGS_DATA_OFFSET);
    result      := std_logic_vector(unsigned(kfifo_in) - unsigned(kfifo_out));
    if result = kfifo_mask then
        kernel_call.kfifo_status <= '1'; -- kfifo full
    else
        kernel_call.kfifo_status <= '0'; -- kfifo not full
    end if;
    lintc_word_write(kernel_call,kernel_response,"010"); -- SYSMUTEX IRQ
----------------------------------------------------------------------------------
-- MORE
```

If the arguments KFIFO happens to be full the flow of kernel procedures shifts and the system mutex is unlocked to allow access to other entities that want to use the shared memory region, *c.f* listing 5.10. After that, one triggers the kernel's timing mechanism to sleep for a specified number of cycles, *c.f* listing 5.5, before acquiring the system mutex again, *c.f* listing 5.2 and checking if the arguments KFIFO got space for a command in the meantime, *c.f* listing 5.4. This would mean that the KIVIO framework was able to dispatch the command to an handler entity.

Listing 5.5: Kernel Package: KFIFO Send Command Wait Return - Sleep (HDL).

```
-- MORE
----------------------------------------------------------------------------------
when C_B_SLEEP_WAIT_RETURN | C_B_KFIFO_FULL_PATH_SLEEP =>
    kernel_call.enable_sched    <= '1';
    kernel_call.resched_req     <= '1';
    wait_timeout_elapsed(kernel_call,kernel_response,timeout,remaining_time,
        timeout_value);
----------------------------------------------------------------------------------
-- MORE
```

In the case where the arguments KFIFO is not full one can proceed to the flow of kernel procedures that summits the 64-bit word to memory. For this, the write pointer, *args.in*, is incremented and masked, which presupposes the use of the single-word write syscall. In the next step the burst capabilities are used again to write the two 32-bit words that constitute the 64-bit command to the argument zone dictated by $args.data_w offset + args.in * 2$, which corresponds to the actual memory position of the block. The multiplication by two is done because one is using 64-bit commands. Both steps can be observed in listing 5.6. The next step to end the placement of the command in memory is the release of the system mutex with the unlock mutex system call, *c.f* section 4.2.3.

Listing 5.6: Kernel Package: KFIFO Send Command Wait Return - Increment *args.in* and Send 64-bit Command (HDL).

```
        -- MORE
        --------------------------------------------------------------------------------
        when C_B_KFIFO_INC_ARGS_IN =>
            kernel_call.enable_index <= '1';
            kernel_call.inc_index <= '0';
            inc_args_in := std_logic_vector(unsigned(kfifo_args_buffer(
                C_KFIFO_ARGS_IN_OFFSET)) + 1);
            args_in_masked <= std_logic_vector(word32_and(inc_args_in,kfifo_args_buffer(
                C_KFIFO_ARGS_MASK_OFFSET)));
            mbus_word_write(kernel_call,kernel_response,(to_integer(unsigned(
                kfifo_base_offset)) + (C_KFIFO_ARGS_BASE_OFFSET + C_KFIFO_ARGS_IN_OFFSET)
                *(C_MACHINE_WIDTH/8)),args_in_masked);
        --------------------------------------------------------------------------------
        when C_B_KFIFO_SEND_CMD_64 =>
            kernel_call.enable_index <= '1';
            kernel_call.inc_index <= '1';
            temp := resize(unsigned(kfifo_args_buffer(C_KFIFO_ARGS_IN_OFFSET)),temp'LENGTH)
                ;
            args_in_x2 := std_logic_vector(resize(shift_left(temp,1),args_in_x2'LENGTH));
            kfifo_woffset := to_integer(unsigned(kfifo_args_buffer(C_KFIFO_RETS_DATA_OFFSET
                )) + unsigned(args_in_x2));
            aux_data_buffer(0) := data(63 downto 32);
            aux_data_buffer(1) := data(31 downto 0);
            mbus_word_write_burst(kernel_call,kernel_response,2,to_integer(unsigned(
                kfifo_base_offset)) + (kfifo_woffset*(C_MACHINE_WIDTH/8)),aux_data_buffer(
                kernel_response.index));
        --------------------------------------------------------------------------------
        -- MORE
```

From here the extended syscall passes to the stage of waiting in polling for the return. As before, the hardware enters a pseudo-sleep stage where it waits a certain amount of clock cycles before continuing, *c.f* listing 5.5. This is supposed to give enough time for the return to be available in the KFIFO that concerns to the returns. Pseudo-sleep is mentioned instead of only sleep since, following HAL-ASOS stipulations, sleep would be associated with the deactivation of certain clock-enabled flip-flops. In this case, the term sleep is only used for clarity, and it is associated with waiting for a specific time period, as stated. The retrieval of returns also starts with lock of the system mutex, *c.f* listing 5.2, for the same reasons as before, i.e., preventing race conditions, followed by a five word burst read regarding the returns KFIFO structure elements, *c.f* listing 5.7.

Listing 5.7: Kernel Package: KFIFO Send Command Wait Return - Read Return Fields (HDL).

```
        -- MORE
        --------------------------------------------------------------------------------
        when C_B_KFIFO_READ_RETURNS =>
            kernel_call.enable_index <= '1';
            kernel_call.inc_index <= '1';
            mbus_word_read_burst(kernel_call,kernel_response,C_KFIFO_RET_WIDTH,to_integer(
                unsigned(kfifo_base_offset)) + (C_KFIFO_RETS_BASE_OFFSET*(C_MACHINE_WIDTH
                /8)),read_args);
            kfifo_rets_buffer(kernel_response.index) <= read_args;
        --------------------------------------------------------------------------------
        -- MORE
```

With the return fields is possible to check if there is a return in the FIFO by subtracting *rets.out* to *rets.in*, *c.f* listing 5.8.

Listing 5.8: Kernel Package: KFIFO Send Command Wait Return - Check Return KFIFO Status (HDL).

```
        -- MORE
        ----------------------------------------------------------------------------------
        when C_B_KFIFO_RET_EMPTY_TEST =>
            kernel_call.enable_index    <= '0';
            kernel_call.inc_index       <= '0';
            kfifo_in    := kfifo_rets_buffer(C_KFIFO_RETS_IN_OFFSET);
            kfifo_out   := kfifo_rets_buffer(C_KFIFO_RETS_OUT_OFFSET);
            kfifo_mask  := kfifo_rets_buffer(C_KFIFO_RETS_MASK_OFFSET);
            kfifo_esize := kfifo_rets_buffer(C_KFIFO_RETS_ESIZE_OFFSET);
            kfifo_data  := kfifo_rets_buffer(C_KFIFO_RETS_DATA_OFFSET);
            result      := std_logic_vector(unsigned(kfifo_in) - unsigned(kfifo_out));
            ---
            if to_integer(unsigned(result)) /= 0 then
                return_rcvd <= '1'; -- return received
            else
                return_rcvd <= '0'; -- return not received
            end if;
            lintc_word_write(kernel_call,kernel_response,"010"); -- SYSMUTEX IRQ
        ----------------------------------------------------------------------------------
        -- MORE
```

If a return is not available the hardware unlocks the system mutex, *c.f* listing 5.10, and enters the waiting stage again, *c.f* listing 5.5, repeating the steps until a return arrives. Upon arrival, the *rets.out* is incremented and masked recurring to a single-word write system call, *c.f* listing 5.9. The following step then reads a single-word from a return block section, using a single-word read system call, *c.f* listing 5.9. Since the accelerator model has a 32-bit machine word, the write and read pertaining to 32 bits are considered single-word and the ones regarding more than 32 bits, i.e., multiple 32-bit words, are considered multiple word bursts because of the burst feature is used. The actual case is that for the AXI4 interface, M00 System, all transfers are burst, but the ones concerning 32 bits are considered bursts of only one word.

Listing 5.9: Kernel Package: KFIFO Send Command Wait Return - Increment *rets.out* and Read Command Return (HDL).

```
        -- MORE
        ----------------------------------------------------------------------------------
        when C_B_KFIFO_INC_RETS_OUT =>
            kernel_call.enable_index <= '1';
            kernel_call.inc_index <= '0';
            inc_rets_out := std_logic_vector(unsigned(kfifo_rets_buffer(
                C_KFIFO_RETS_OUT_OFFSET)) + 1);
            rets_out_masked <= std_logic_vector(word32_and(inc_rets_out,kfifo_rets_buffer(
                C_KFIFO_RETS_MASK_OFFSET)));
            mbus_word_write(kernel_call,kernel_response,to_integer(unsigned(
                kfifo_base_offset)) + ((C_KFIFO_RETS_BASE_OFFSET + C_KFIFO_RETS_OUT_OFFSET
                )*(C_MACHINE_WIDTH/8)),rets_out_masked);
        --------------------------------
        when C_B_KFIFO_READ_PROC_RET_WORD =>
            kernel_call.enable_index <= '1';
            kernel_call.inc_index <= '0';
            kfifo_woffset := to_integer(unsigned(kfifo_rets_buffer(C_KFIFO_RETS_DATA_OFFSET
                )) + unsigned(kfifo_rets_buffer(C_KFIFO_RETS_OUT_OFFSET)));
            mbus_word_read(kernel_call,kernel_response,to_integer(unsigned(
                kfifo_base_offset)) + (C_KFIFO_RETS_BASE_OFFSET + (kfifo_woffset*(
                C_MACHINE_WIDTH/8))),read_return);
        ----------------------------------------------------------------------------------
        -- MORE
```

Logically, the system mutex must be released with the appropriate system call before leaving the extended procedure, listing 5.10.

Listing 5.10: Kernel Package: KFIFO Send Command Wait Return - Unlock System Mutex and Leave Procedure (HDL).

```
        -- MORE
        --------------------------------------------------------------------------------
        when C_B_KFIFO_SYSMUTEX_UNLOCK | C_B_KFIFO_RET_PATH_SYSMUTEX_UNLOCK |
            C_B_KFIFO_FULL_PATH_SYSMUTEX_UNLOCK =>
            system_mutex_unlock(kernel_call,kernel_response,mutex_status);
        --------------------------------------------------------------------------------
        -- MORE
        --------------------------------------------------------------------------------
        when C_B_PROCEDURE_DONE =>
            procedure_done <= '1';
        --------------------------------------------------------------------------------
        -- MORE
```

5.1.1 Procedure Scheduling

Performing these extended system calls requires not only memory storage to store the appropriate flow of kernel procedures to execute but also some type of control to enforce the execution flow. For this, another microcoded entity was introduced in the microkernel, i.e., the procedure scheduler entity. Vertical microcode is based on the premise that a microinstruction activates a certain function through a function code to perform a set of datapath operations, *c.f* section 2.3. Thus, contrarily to the microprogram, *c.f* section 4.2.4, the scheduler employs single address vertical microcode which means that instead of activating control signals directly it activates kernel-level system calls. The decoding of these microinstructions that dictate the flow of low-level microinstructions from the microprogram is not complex since the two entities are connected, which means that the decoding is performed in the scheduler but also in the microprogram. The kernel requests issued by the hardware task are handled by the microprogram unless a procedure establishes that the scheduler should be active, by asserting the *enable_sched* signal. When the scheduler is enabled the kernel switches its execution mode and the scheduler starts instructing the microprogram with the right syscalls IDs. The scheme follows an hierarchy as depicted by fig. 5.1.2. The user-level procedures identified by their procedure ID enter the scheduler and are divided into multiple kernel-level syscalls each one with a maximum of four steps. As in the microprogram, the scheduler selects inputs for testing, the result then influences the low-level system calls to be executed, e.g., if one is testing for the KFIFO status, a full result originates a mutex unlock followed by sleep, but if the KFIFO is not full the scheduler establishes the writing of the 64-bit command to main memory, and after, the waiting for return.

This implementation of the scheduler introduces the branching of system calls based on certain jump conditions, and adds upon the idea of incremental procedure scheduler presented in [62]. With this, one

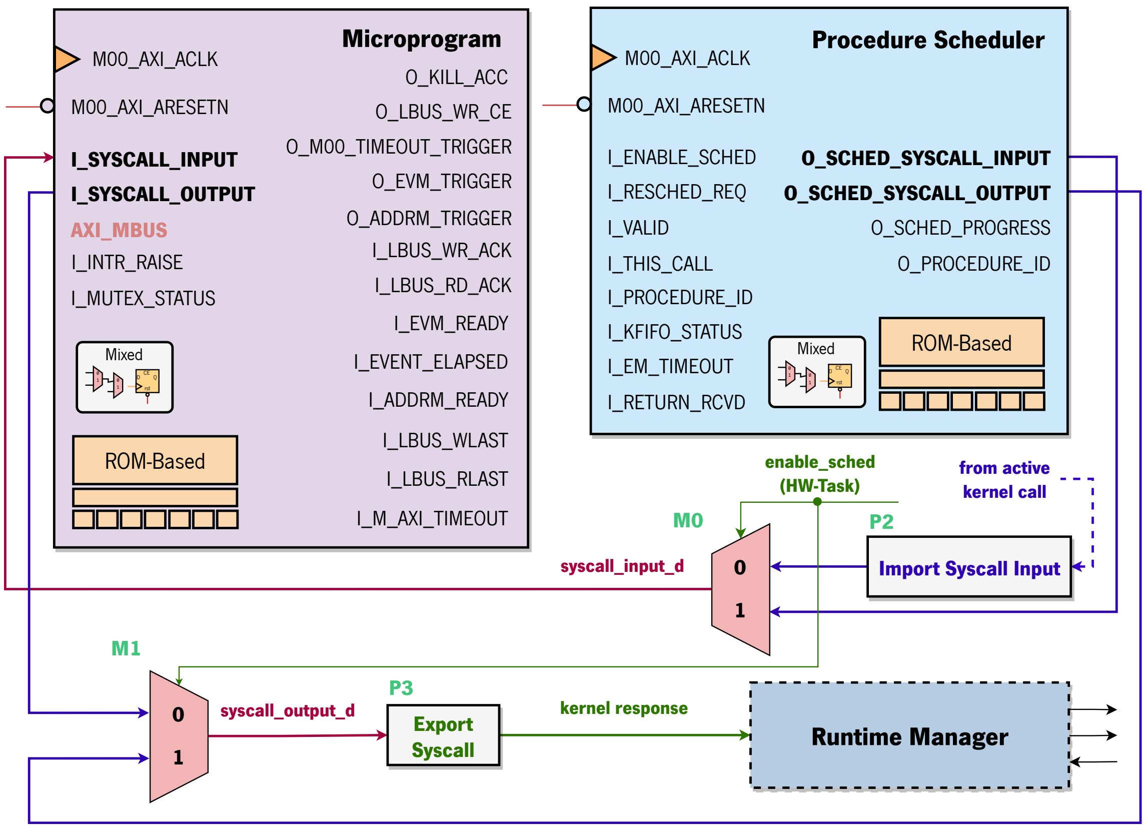


Figure 5.1.2: Microcode Entity Hierarchy Overview. The assertion of the *enable_sched* signal activates scheduler-dictated microprogram execution, otherwise, the kernel uses stand-alone microprogram execution.

is capable of branching between kernel-level system calls in various ways. These include (1) conditional, (2) unconditional, (3) indirect, and (4) call/return. The conditional jumps are processed by analyzing the logical value of the test input signals of the scheduler, e.g., the testing of the input *return_rcvd* originates two possible branches in fig. 5.1.1, one that returns to sleep if the return has not arrived yet (🟩) (signal not asserted), and another branch that retrieves the return (🟦) (signal asserted), *c.f* table 5.1 and fig. 5.1.1. The unconditional jumps are performed by testing an auxiliary true test flag, *aux_true*, that is always '1'. This causes the scheduler to simply increment and pass to next syscall in the next memory position. The indirect jump makes use of another auxiliary signal, *aux_false*, that is always '0'. This induces the scheduler to use the value of NSF, to generate the next address. This type of jump can be done to any memory position that refers to the currently active procedure, i.e., any microinstruction out of the sixteen defined, *c.f* table 5.1. The call/return functionality can be considered as branching to the low-level memory of the microprogram, which has its own address space. The scheduler specifies the system call ID input of the microprogram. The latter is then used to generate the address of microprogram memory by concatenation with the microprogram counter, *c.f* section 4.2.4. The capability to return from low-level subroutines is established by the microprogram, and it is done by providing the microprogram's *valid* signal, *c.f* section 4.2.4, to the scheduler. The logical combination, on a *and* gate (G0 in fig. 5.1.3), of the rescheduling request, *i_resched_req*, coming from the hardware task, the internal test input result, *test_result*, and the microprogram's valid signal, gives origin to the scheduler's program counter (C0 in fig. 5.1.3) *increment* input. This way, a new scheduler address is only generated at the end of a four-step low-level system call of the microprogram. Thus, when the kernel carries out scheduler-dictated execution, the *valid* signal that the hardware task receives is the one pertaining to the scheduler.

The scheduler follows an internal architecture very similar to the microprogram, *c.f* section 4.2.4, since they are both microcoded control units. Regarding this, the major difference in the scheduler is the memory size, with a depth of 256 words, for procedure-granularity of up to 16 kernel-level system calls. The HDL representation of the scheduler's memory is represented in listing 5.11.

Listing 5.11: Procedure Memory: Combinational ROM (HDL).

```vhdl
subtype rom_word_t is std_logic_vector(ROM_DATA_WIDTH-1 downto 0);
type procedure_rom_t is array (0 to ROM_DEPTH-1) of rom_word_t;

signal procedure_rom : procedure_rom_t := (
    --------------------------------------------------------------------------------------
    -- MORE
    ------------------------------------        ------------------------------------------
    -- P02 - POLLED_SAFE_SEND_KFIFO_CMD         -- P05 - POLLED_SAFE_SEND_KFIFO_CMD_LOCAL
    32 => "000000011001111110000001",           80 => "000000011001111110000001",
    33 => "000100101110111110000001",           81 => "000100100111111110000001",
    34 => "001011100100000010000001",           82 => "001011100100000010000001",
    35 => "001101001101111110000001",           83 => "001101000110111110000001",
    36 => "010001011111111110000001",           84 => "010001011000111110000001",
    37 => "010101101011111110000001",           85 => "010101101011111110000001",
    38 => "011001110010111110000001",           86 => "011001110010111110000001",
    39 => "011110001001111110000001",           87 => "011110001001111110000001",
    40 => "100010011110111110000001",           88 => "100010010111111110000001",
    41 => "100101010100000110000001",           89 => "100101010100000110000001",
    42 => "101010111101111110000001",           90 => "101010110110111110000001",
    43 => "101111001100111110000001",           91 => "101111000101111110000001",
    44 => "110011011011111110000001",           92 => "110011011011111110000001",
    45 => "110100000000000000000010",           93 => "110100000000000000000010",
    46 => "111011111011111110000001",           94 => "111011111011111110000001",
    47 => "111100000010000100000001",           95 => "111100000010000100000001",
    ------------------------------------        ------------------------------------------
    -- MORE
    --------------------------------------------------------------------------------------
begin
    O_DOUT <= procedure_rom(to_integer(unsigned(I_ADDR)));
end architecture rtl;
```

In listing 5.11 one can see two user-level procedures to perform the aforementioned extended features. Both procedures represent the blocking version of the extended features, which wait for the return in polling. The procedure on the left does this by mapping the KFIFOs to the SYSRAM, and the procedure on the right maps the KFIFOs to the LRAM. Overall, the kernel still lacks a procedure to access the returns regarding the non-blocking version at later stages of the hardware task execution. Regarding this, the dissertation mainly uses the polling versions to issue requests through shared memory (SYSRAM or LRAM).

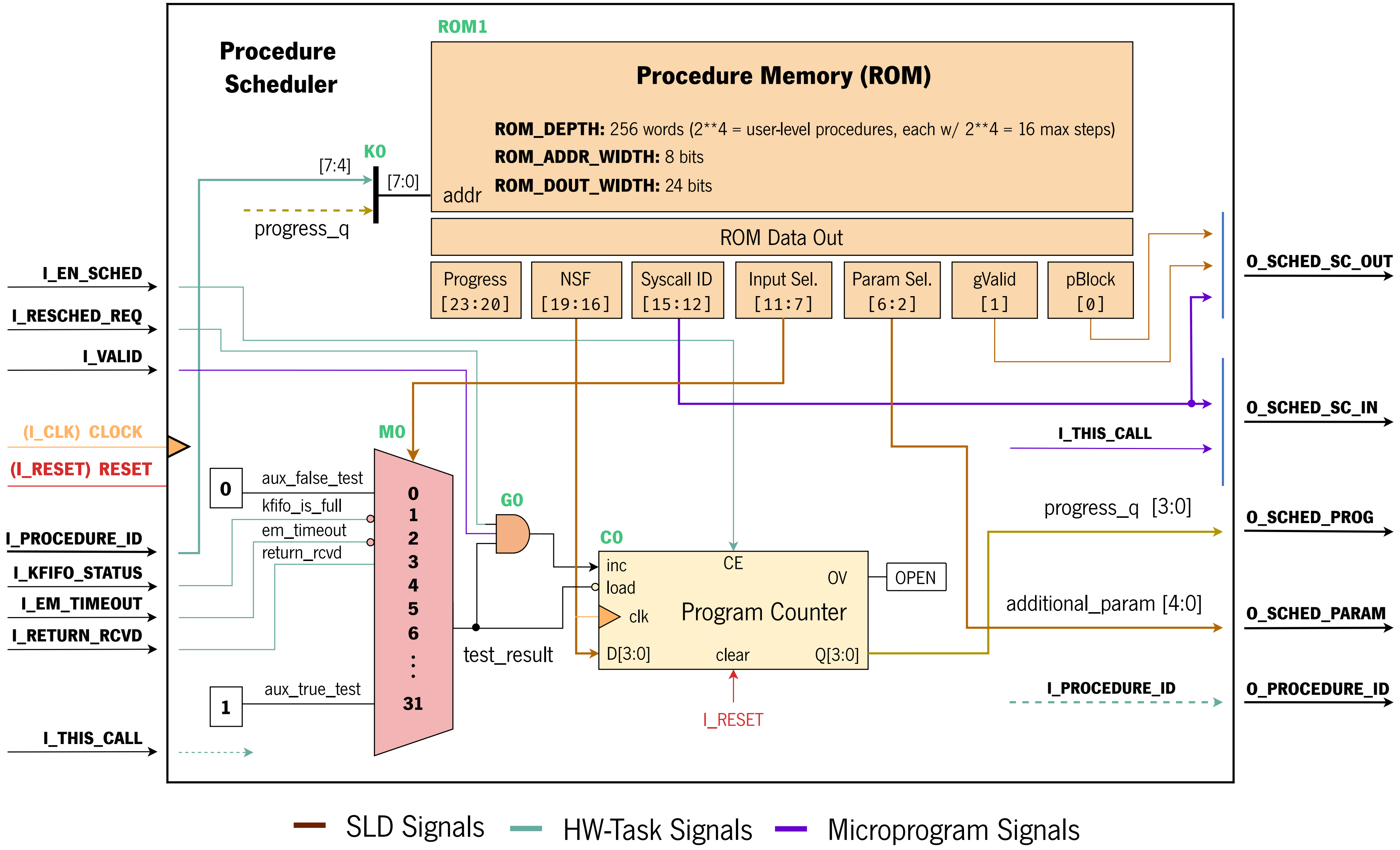


Figure 5.1.3: Procedure Scheduler RTL Design Internal Architecture.

Table 5.1: Simplified Memory Representation of the Polled Safe Send KFIFO Command, Polled Wait for Return, back to fig. 5.1.1.

	KIVIO-Extended Features					
Syscall (UL)	**Progress**	**NSF**	**Syscall (KL)**	**Input Sel.**	**GV**	**PB**
Polled Safe Send KFIFO Cmd., Polled Wait for Return	0	0001	SYSCALL_MUTEX_LOCK	AUX_TRUE	0	1
	1	0010	SYSCALL_MBUS_READ_BURST (5W)	AUX_TRUE	0	1
	2	1110	SYSCALL_LINTC_WRITE	NOT_KFIFO_FULL	0	1
	3	0100	SYSCALL_MBUS_WRITE	AUX_TRUE	0	1
	4	0101	SYSCALL_MBUS_WRITE_BURST (2W)	AUX_TRUE	0	1
	5	0110	SYSCALL_MUTEX_UNLOCK	AUX_TRUE	0	1
	6	0111	SYSCALL_WAIT_EVENT_TIMEOUT	AUX_TRUE	0	1
	7	1000	SYSCALL_MUTEX_LOCK	AUX_TRUE	0	1
	8	1001	SYSCALL_MBUS_READ_BURST (5W)	AUX_TRUE	0	1
	9	0101	SYSCALL_LINTC_WRITE	RET_RCVD	0	1
	10	1011	SYSCALL_MBUS_WRITE	AUX_TRUE	0	1
	11	1100	SYSCALL_MBUS_READ	AUX_TRUE	0	1
	12	1100	SYSCALL_MUTEX_UNLOCK	AUX_TRUE	0	1
	13	0000	SYSCALL_WORK_NONE	AUX_FALSE	1	0
	14	1111	SYSCALL_MUTEX_UNLOCK	AUX_TRUE	0	1
	15	0000	SYSCALL_WAIT_EVENT_TIMEOUT	NOT_TIMEOUT	0	1

(UL) User-Level **(KL)** Kernel-Level **(GV)** Global Valid **(PB)** Procedure Block **(W)** Words (32 bits)

🟩 KFIFO Not Full Branch 🟥 KFIFO Full Branch 🟦 KFIFO Retrieve Return Branch 🟨 Interrupt Request

5.1.2 BFM Verification and Extended Features

The interface M00 System and the KIVIO-extended features, *c.f* section 5.1, were tested with the AXI VIP following the block design of fig. 5.1.5. Thus, the VIP acts a master writing to S00 Control and another VIP acts as a slave to intersect the master interface requests. The tests were straightforward, the main objective established was to test hardware task extended procedures in terms of functionality and M00 System interactions. In this particular case, one used an hardware task developed to exchange information with the host system using Linux pipes.

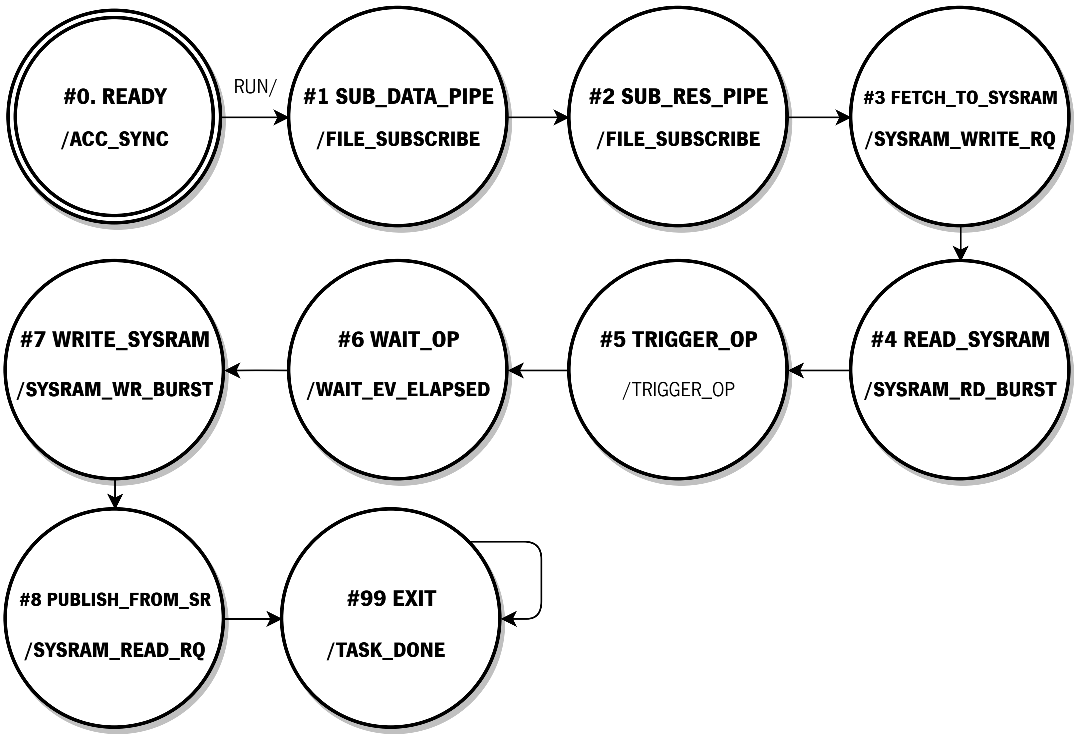


Figure 5.1.4: Pipe Communication Hardware Task's Finite State Machine (FSM).

This hardware task's FSM is represented in fig. 5.1.4. The main objective of the test is to explore pipe communication between the software app on the host system side and the hardware accelerator developed. For this, the test will request to subscribe to two KIVIO pipes, one for getting unencrypted data, and another for returning encrypted data from the hardware. Thus, the latter fetches the data from the app through the first pipe and writes it to shared memory, then the hardware task reads the data placed in the latter and performs an encryption operation over it, writing it again in shared memory when the operation is completed. Then, it requests for the latter to be published from memory again to the app through the return pipe. The test also presupposes that the hardware task stays subscribed to the aforementioned pipes. The test results can be seen in fig. 5.1.10, fig. B.7, and fig. B.8.

The test began by using the master VIP to write to the microkernel's local interrupt controller, *c.f* section 4.5.2, to enable all interrupts, asserting the *enable_intr* bit of the control register to enable all interrupts, and mask the interrupt pertaining to the system mutex, specified as interrupt source one. The latter was done by enabling bit 1 of the mask field of the LINTC's control register. Both of these assertions are made with write at 65.00 nanoseconds observed in fig. B.7. Note that the writes and reads performed through the S00 Control interface and represented in fig. 5.1.10, fig. B.7, and fig. B.8 are made in the VIP's testbench to mimic KIVIO or host system intervention. Following, a read is performed on the same register, i.e., LINTC control, to check if the write succeeded. Then the accelerator is enabled by writing to the kernel's control register and asserting the *run* bit. The assertion of the kernel's *run* bit enables the *task_run* signal and promotes the hardware task to its first running state one cycle later. With this, the hardware task passes from *s0_ready* to *s0_1_write_args*, at 505.00 nanoseconds.

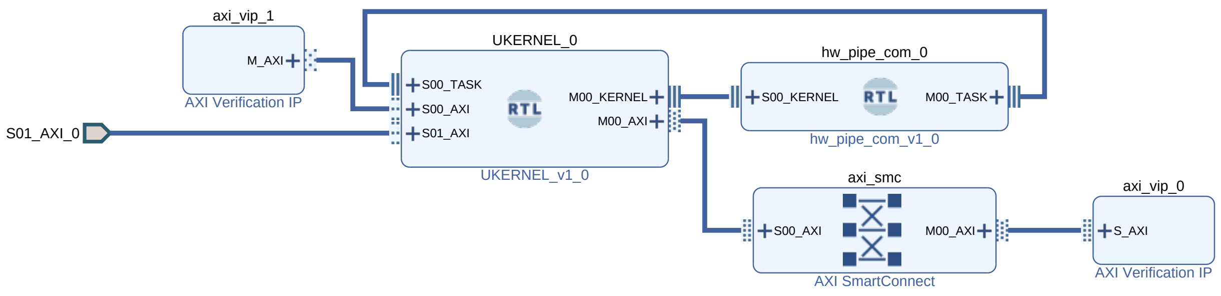
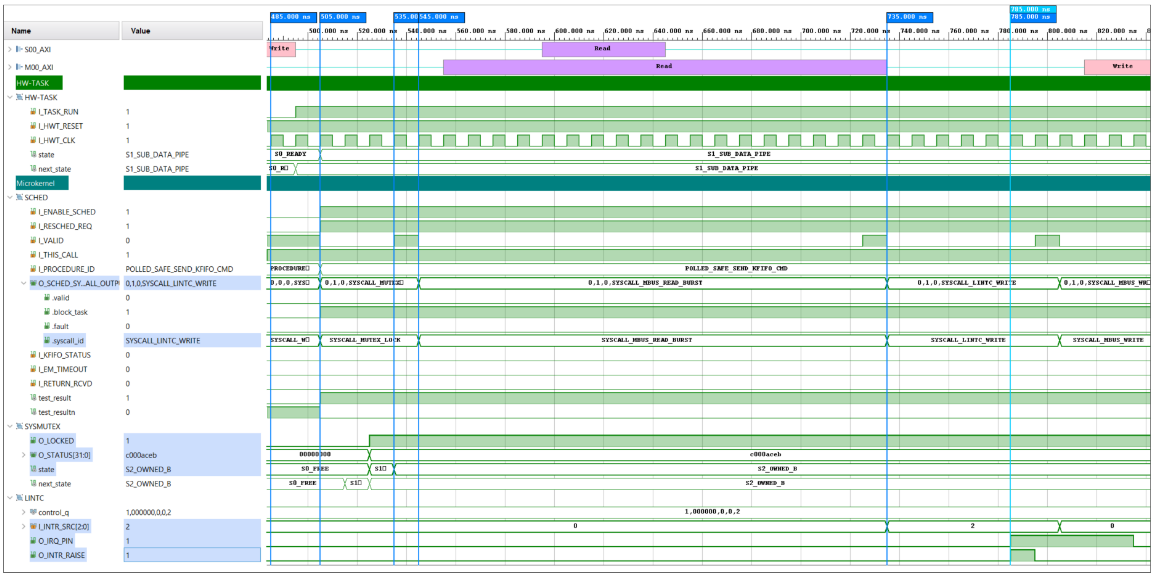


Figure 5.1.5: AXI VIP Simulation – Extended Feature Test: Block Design.

Since one is performing VIP simulation, the main memory positions related to the KFIFO have no information. For this reason, in the state #0_1 and #0_2, one writes to main memory through the M00 System interface to the arguments and returns sections of the KFIFO, respectively. To perform the pipe hardware task to completion, one stipulated an empty argument/command section and a return section with some 32-bit words, so that the extended feature syscalls do not stay in loop because there are no returns. The arguments write happens at 505.00 nanoseconds and the returns write happens at 925.00 nanoseconds, fig. B.7.

The first actual processing state of the hardware task starts at 1,225.00 nanoseconds, *c.f* fig. B.7. In the latter, one performs the placement of the command that subscribes to the data pipe and waits for its returned descriptor. From this point on one will use another simulation instance to describe in more detail the execution of one extended procedure, in this case, related to the *s1_sub_data_pipe* state.

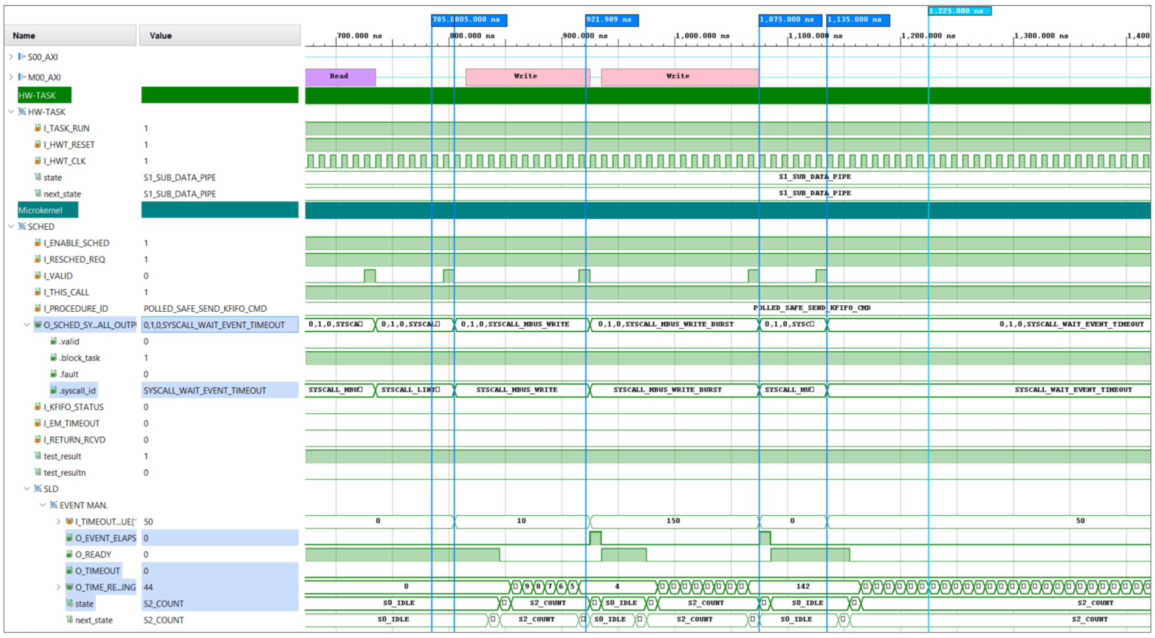
Concerning this, at 485.00 nanoseconds, in fig. 5.1.6, it is also possible to see the extended procedure start with a mutex lock system call. This call relates to the first lock of the system mutex represented in fig. 5.1.1 and happens at the instant of 585.00 nanoseconds. The lock is successful because the mutex's *o_locked* output is asserted, the status now has the owner "c000aceb", which represents the acquisition of the mutex by the microkernel, and the state of the mutex's FSM changed to *s2_owned_b* at 535.00 nanoseconds. The next low-level system call is a burst read performed in the M00 System interface. This step corresponds to the second step of fig. 5.1.1, where the five elements of the arguments KFIFO are read and stored in local buffers. The burst read syscall starts at 545.00 nanoseconds and ends at 735.00 nanoseconds. It is important to note that here on is referring to the syscall ID output of the scheduler since the scheduler-dictated microprogram execution is being used with the enable of *enable_sched*, *c.f* fig. 5.1.2. At instant 735.00 nanoseconds one also reaches the start of the local interrupt controller write system call, which asserts a specific interrupt source and generates an IRQ. The source number two (735.00 nanoseconds) is used since it represents the interrupt source associated with the system mutex and, therefore, related with the writing of a command to the arguments KFIFO. This IRQ is only generated when the check on the KFIFO also performed in this stage establishes that the KFIFO is not full. This is possible because the check is not performed by a low-level system call, then as one has the possibility of using one syscall per case, the LINTC one is used. At instant 785.00 nanoseconds is possible to see the generation of the IRQ with the transition of *o_irq_pin* from '0' to '1' at the rising edge of the system clock. This interrupt request is held for four cycles to be perceived by the host system.



Name	Value
S00_AXI	
M00_AXI	
HW-TASK	
HW-TASK	
I_TASK_RUN	1
I_HWT_RESET	1
I_HWT_CLK	1
state	S1_SUB_DATA_PIPE
next_state	S1_SUB_DATA_PIPE
Microkernel	
SCHED	
I_ENABLE_SCHED	1
I_RESCHED_REQ	1
I_VALID	0
I_THIS_CALL	1
I_PROCEDURE_ID	POLLED_SAFE_SEND_KFIFO_CMD
O_SCHED_SY...ALL_OUTPI	0,1,0,SYSCALL_LINTC_WRITE
.valid	0
.block_task	1
.fault	0
.syscall_id	SYSCALL_LINTC_WRITE
I_KFIFO_STATUS	0
I_EM_TIMEOUT	0
I_RETURN_RCVD	0
test_result	1
test_resultn	0
SYSMUTEX	
O_LOCKED	1
O_STATUS[31:0]	c000aceb
state	S2_OWNED_B
next_state	S2_OWNED_B
LINTC	
control_q	1,000000,0,0,2
I_INTR_SRC[2:0]	2
O_IRQ_PIN	1
O_INTR_RAISE	1

Figure 5.1.6: Extended Feature VIP Simulation: System Mutex Lock, Read Argument Fields, Check Argument KFIFO Status and Generate IRQ.

The instant 805.00 nanoseconds in fig. 5.1.7 depicts the instant where the *args.in* field of the arguments KFIFO is incremented and masked. Moreover, the instant 922.00 nanoseconds represents the start of the burst word syscall that writes the 64-bit command to the shared memory region pertaining to the arguments KFIFO. At 1,075.00 nanoseconds the system mutex is released from the microkernel's control since the command has been written, *c.f* fig. 5.1.1. Additionally, at 1,135.00 nanoseconds the wait event timeout syscall begins, and the event manager unit starts counting from the specified value to zero, 1,225.00 nanoseconds.



Name	Value
S00_AXI	
M00_AXI	
HW-TASK	
HW-TASK	
I_TASK_RUN	1
I_HWT_RESET	1
I_HWT_CLK	1
state	S1_SUB_DATA_PIPE
next_state	S1_SUB_DATA_PIPE
Microkernel	
SCHED	
I_ENABLE_SCHED	1
I_RESCHED_REQ	1
I_VALID	0
I_THIS_CALL	1
I_PROCEDURE_ID	POLLED_SAFE_SEND_KFIFO_CMD
O_SCHED_SY...ALL_OUTPI	0,1,0,SYSCALL_WAIT_EVENT_TIMEOUT
.valid	0
.block_task	1
.fault	0
.syscall_id	SYSCALL_WAIT_EVENT_TIMEOUT
I_KFIFO_STATUS	0
I_EM_TIMEOUT	0
I_RETURN_RCVD	0
test_result	1
test_resultn	0
SLD	
EVENT MAN.	
I_TIMEOUT...UE[	50
O_EVENT_ELAPS	0
O_READY	0
O_TIMEOUT	0
O_TIME_RE...ING	44
state	S2_COUNT
next_state	S2_COUNT

Figure 5.1.7: Extended Feature VIP Simulation: Increment *args.in*, Send 64-bit Command, and Sleep.

In fig. 5.1.8 one can see the procedure sleep coming to an end when zero is reached, with the *o_time_remaining* output of the event manager unit. The timeout output bit, *o_timeout*, is then asserted, which indicates to the kernel that it can go to the next system call established by the scheduler. Following fig. 5.1.1, the next syscall is the one that reads the return KFIFO fields and stores then in a local buffer. It starts at the instant 1,735.00 nanoseconds, and ends at instant 1,925.00 nanoseconds where the next syscall begins. At this stage, the system call active is the one related to no operation since one just wants to check the state of the returns KFIFO. As the test input of the scheduler related with the reception of the return, *i_return_rcvd*, is not asserted the test fails, and the scheduler performs an indirect jump to the mutex unlock system call, 1,935.00 nanoseconds, followed by the sleep once again, 1,995.00 nanoseconds, as depicted by fig. 5.1.1. At 2,0098.63 nanoseconds is possible to see the event manager counting again until timeout.

As a Proof of Concept (PoC), this test was performed so that the scheduler would follow the KFIFO not full branch, and never enter the KFIFO retrieve return branch, *c.f* table 5.1. Despite, in the overall simulation of the hardware task, *c.f* fig. B.7, it is possible to see that one forced the retrieve return branch with the write to the returns section of the SYSRAM in state *s0_2_write_rets*, to assure the hardware task proceeded to the next state, deemed as *s2_sub_res_pipe*.

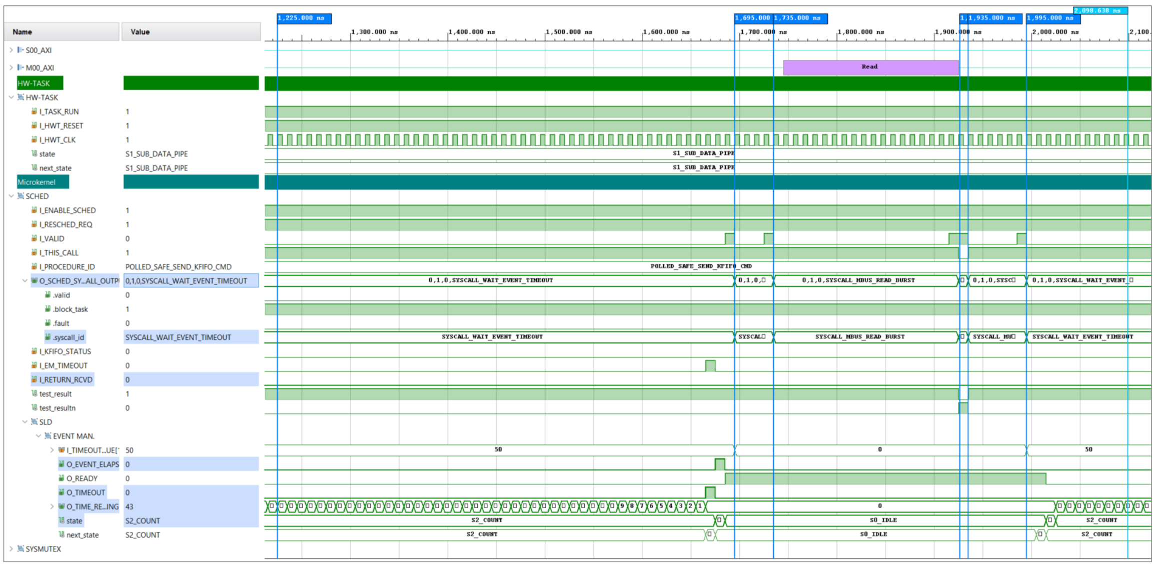


Figure 5.1.8: Extended Feature VIP Simulation: System Mutex Lock, and Read Return Fields.

Another thing to assure when going to the next extended feature state is that the interrupt is cleared, for this, one performed two testbench writes on the LINTC's control register through the S00 Control interface to enable the bit that clears the system mutex interrupt followed by another to put that same bit to '0'. This process is done each time one moves to another extended procedure at the timestamps 2,145.00, 3,925.00, and 5,705.00 nanoseconds. As forementioned the clearing of the interrupt's status registers are meant to be done by KIVIO, but here that was emulated recurring to the master VIP. One subscribes to the results pipe in state *s2_sub_res_pipe* at 2,995.00 nanoseconds. When this state terminates the hardware task's FSM moves to the next state at 4,775.00 nanoseconds. The state #3 requests a write from the app unencrypted data to the SYSRAM through the data pipe. After, the next state, deemed as *s4_read_sysram*, reads the host's main memory section with the data and stores in a local buffer (protected by the system mutex), *in_buffer_0_q*, in fig. B.8. This read (burst) into the buffer ends at about 7,000.00 nanoseconds. The state #5 triggers the *ip_xor* entity to start its operation (or the HAL-ASOS AES IP, in case of AES encryption, *c.f* fig. 5.1.12). Thus, the latter performs a *xor* between all the 32-bit words of the buffer and an established cypher key, outputting the results onto the *out_buffer_0_q* buffer of fig. B.8. The state #6 waits for the *xor_done* signal or until timeout for stipulated amount of cycles. Following fig. 5.1.4, the next state, *s7_write_sysram*, writes the contents of this buffer again to the SYSRAM in burst format through the M00 System interface (also protected by the system mutex). The write ends at 8,125.00 nanoseconds and the last state of the pipe hardware task also starts at this point. As PoC, the read and writes of data performed from/to main memory were established as 32-word bursts, but more complex applications might require bursts of more words. Future research has to make sure the microkernel handles bursts with more words than the current maximum (256 words) while maintaining data integrity. Lastly, in state *s8_publish_from_sysram*, a command request is made to publish the *xor*-encrypted data from main memory back to the app, through the results pipe. After this stage, the hardware task simply goes to an idle state, deemed as *s99_exit* where it can be restarted if the host system decides. In this state the hw-task also signals the end of processing to the microkernel with the *task_done* signal.

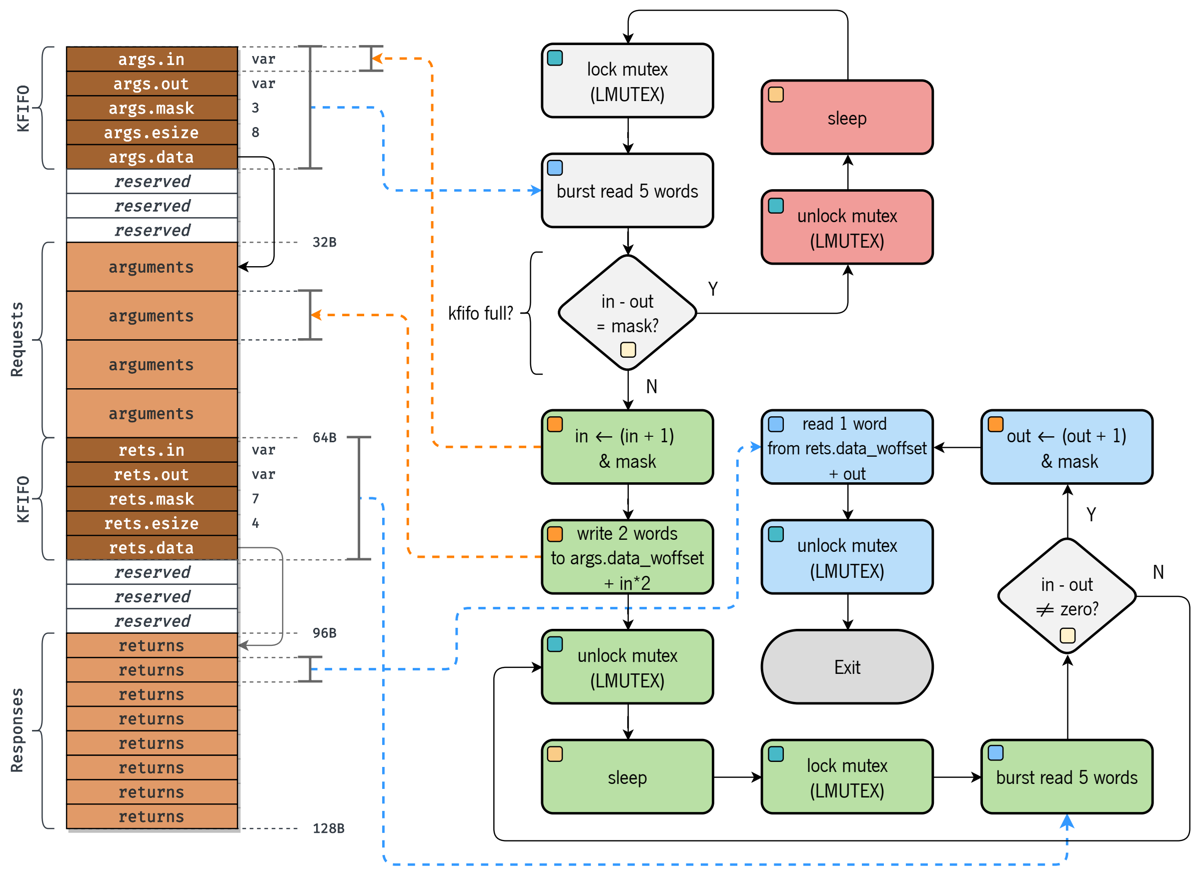
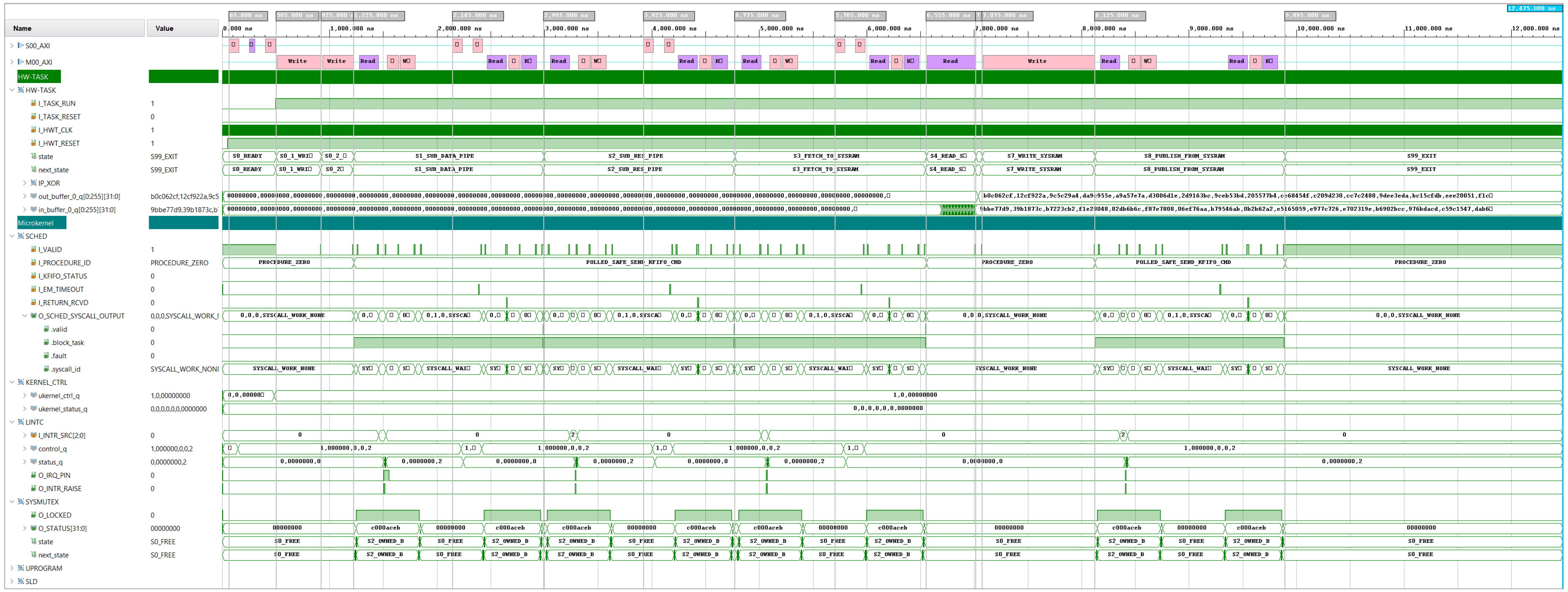
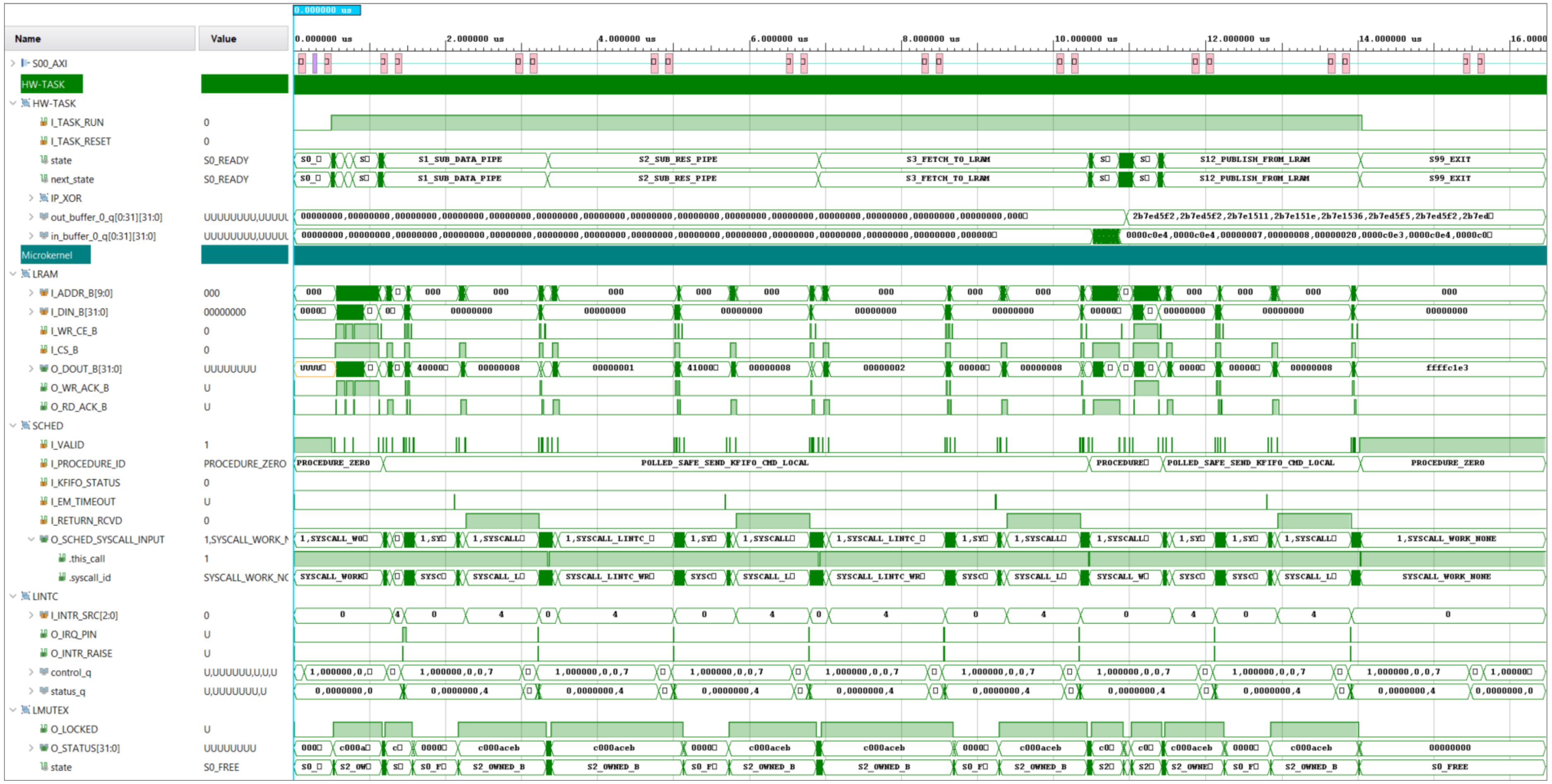


Figure 5.1.9: Composite system call for executing KFIFO-extended features in a polling manner (LRAM); The same tests for the pipe hardware task, *c.f* fig. 5.1.4, were also done for LRAM communication, changing all the extended commands to map the KFIFOs into local memory, and also store the non-encrypted/encrypted words. This scheme relies on the system call sequence HDL of listing A.5. The pipe HW-Task's HDL can be seen in listing A.8 (SYSRAM), and in listing A.9 (LRAM).



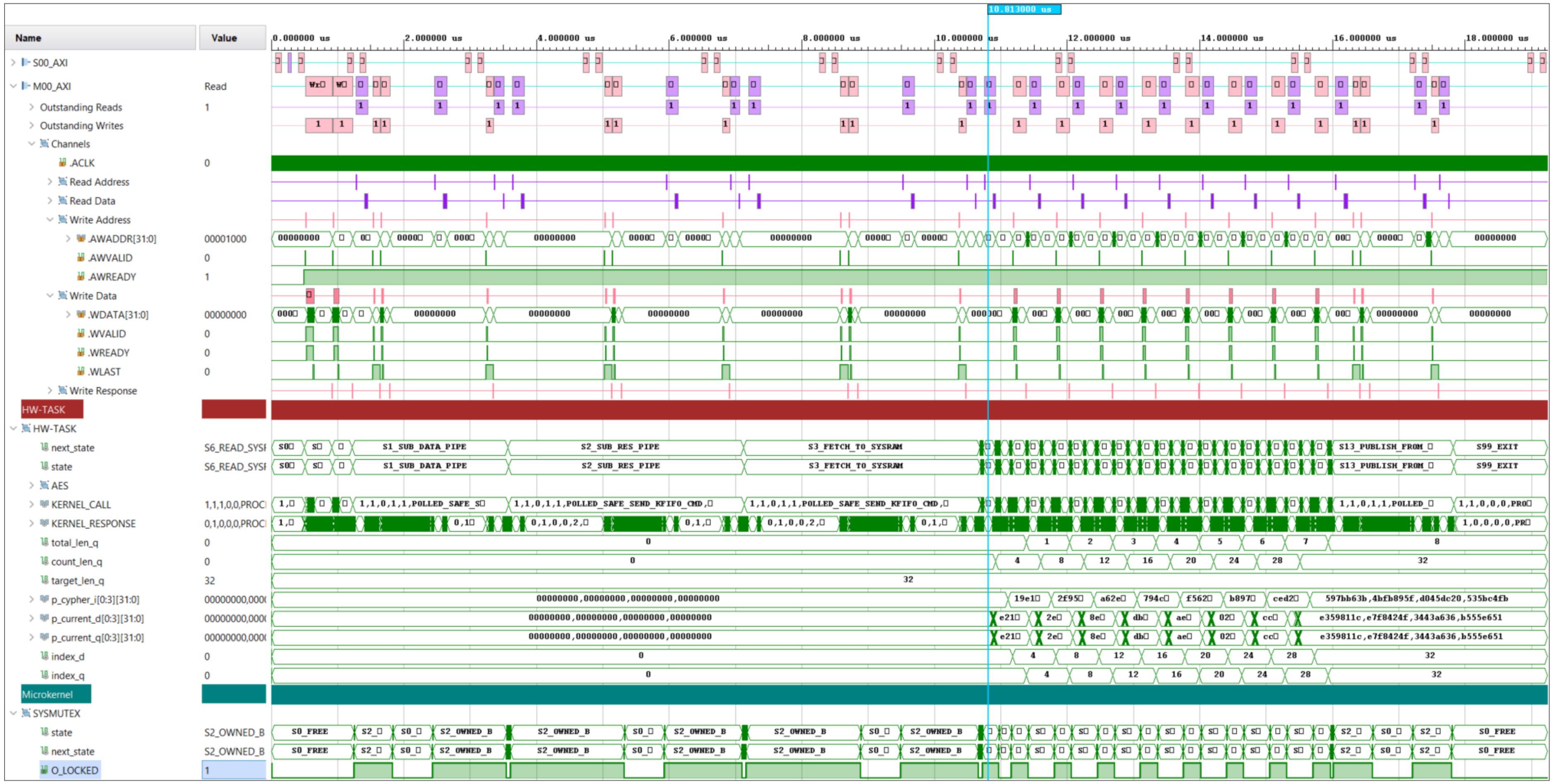
Name	Value
S00_AXI	
M00_AXI	
HW-TASK	
HW-TASK	
I_TASK_RUN	1
I_TASK_RESET	0
I_HWT_CLK	1
I_HWT_RESET	1
state	S99_EXIT
next_state	S99_EXIT
IP_XOR	
out_buffer_0_q[0:255][31:0]	b0c062cf,12cf922a,9c5
in_buffer_0_q[0:255][31:0]	9bbe77d9,39b1873c,b
Microkernel	
SCHED	
I_VALID	1
I_PROCEDURE_ID	PROCEDURE_ZERO
I_KFIFO_STATUS	0
I_EM_TIMEOUT	0
I_RETURN_RCVD	0
O_SCHED_SYSCALL_OUTPUT	0,0,0,SYSCALL_WORK_I
.valid	0
.block_task	0
.fault	0
.syscall_id	SYSCALL_WORK_NONI
KERNEL_CTRL	
ukernel_ctrl_q	1,0,00000000
ukernel_status_q	0,0,0,0,0,0,0000000
LINTC	
I_INTR_SRC[2:0]	0
control_q	1,000000,0,0,2
status_q	0,0000000,2
O_IRQ_PIN	0
O_INTR_RAISE	0
SYSMUTEX	
O_LOCKED	0
O_STATUS[31:0]	00000000
state	S0_FREE
next_state	S0_FREE
UPROGRAM	
SLD	

Figure 5.1.10: Pipe Communication Hardware Task (SYSRAM w/ *XOR* Encryption): Overview.



Name	Value
S00_AXI	
HW-TASK	
HW-TASK	
I_TASK_RUN	0
I_TASK_RESET	0
state	S0_READY
next_state	S0_READY
IP_XOR	
out_buffer_0_q[0:31][31:0]	UUUUUUUU,UUUUL
in_buffer_0_q[0:31][31:0]	UUUUUUUU,UUUUL
Microkernel	
LRAM	
I_ADDR_B[9:0]	000
I_DIN_B[31:0]	00000000
I_WR_CE_B	0
I_CS_B	0
O_DOUT_B[31:0]	UUUUUUUU
O_WR_ACK_B	U
O_RD_ACK_B	U
SCHED	
I_VALID	1
I_PROCEDURE_ID	PROCEDURE_ZERO
I_KFIFO_STATUS	0
I_EM_TIMEOUT	U
I_RETURN_RCVD	0
O_SCHED_SYSCALL_INPUT	1,SYSCALL_WORK_N
.this_call	1
.syscall_id	SYSCALL_WORK_NC
LINTC	
I_INTR_SRC[2:0]	0
O_IRQ_PIN	U
O_INTR_RAISE	U
control_q	U,UUUUUUU,U,U,U
status_q	U,UUUUUUUU,U
LMUTEX	
O_LOCKED	U
O_STATUS[31:0]	UUUUUUUU
state	S0_FREE

Figure 5.1.11: Pipe Communication Hardware Task (LRAM w/ *XOR* Encryption): Overview.



Name	Value
S00_AXI	
M00_AXI	Read
Outstanding Reads	1
Outstanding Writes	
Channels	
ACLK	0
Read Address	
Read Data	
Write Address	
AWADDR[31:0]	00001000
AWVALID	0
AWREADY	1
Write Data	
WDATA[31:0]	00000000
WVALID	0
WREADY	0
WLAST	0
Write Response	
HW-TASK	
HW-TASK	
next_state	S6_READ_SYS
state	S6_READ_SYS
AES	
KERNEL_CALL	1,1,1,0,0,PROC
KERNEL_RESPONSE	0,1,0,0,0,PROC
total_len_q	0
count_len_q	0
target_len_q	32
p_cypher_i[0:3][31:0]	00000000,000
p_current_d[0:3][31:0]	00000000,000
p_current_q[0:3][31:0]	00000000,000
index_d	0
index_q	0
Microkernel	
SYSMUTEX	
state	S2_OWNED_B
next_state	S2_OWNED_B
O_LOCKED	1

Figure 5.1.12: Pipe Communication Hardware Task (SYSRAM w/ AES Encryption): Overview.

5.2 Scalability and Execution Modes

Another limitation of HAL-ASOS was the scalability regarding user-level procedures. The dissertation implementation follows what was established by HAL-ASOS in what concerns the design of the hardware task entity. Thus, the latter is divided into a common control-datapath architecture, establishing the call to hardware user-level procedures in the control unit section. This basis presupposes that only one user-level procedure can be called per state of the hardware task, which logically originates a number of hardware task states that is proportional to the amount of features one wants to implement in hardware to communicate with the host system, and also to activate specific services of the accelerator. Initially, one presented a standalone microprogram flow, based on kernel-level system calls, *c.f* section 4.2.3. If this was the only option to implement procedures in the microkernel the number of states of the hardware task would increase drastically. For this reason, another layer of abstraction was presented, with the introduction of the procedure scheduler, *c.f* section 5.1.1. With this approach, the hardware task can now use single-syscall procedures, that use only one kernel-level procedure, or implement composite-syscall procedures, the rely on multiple kernel-level procedures.

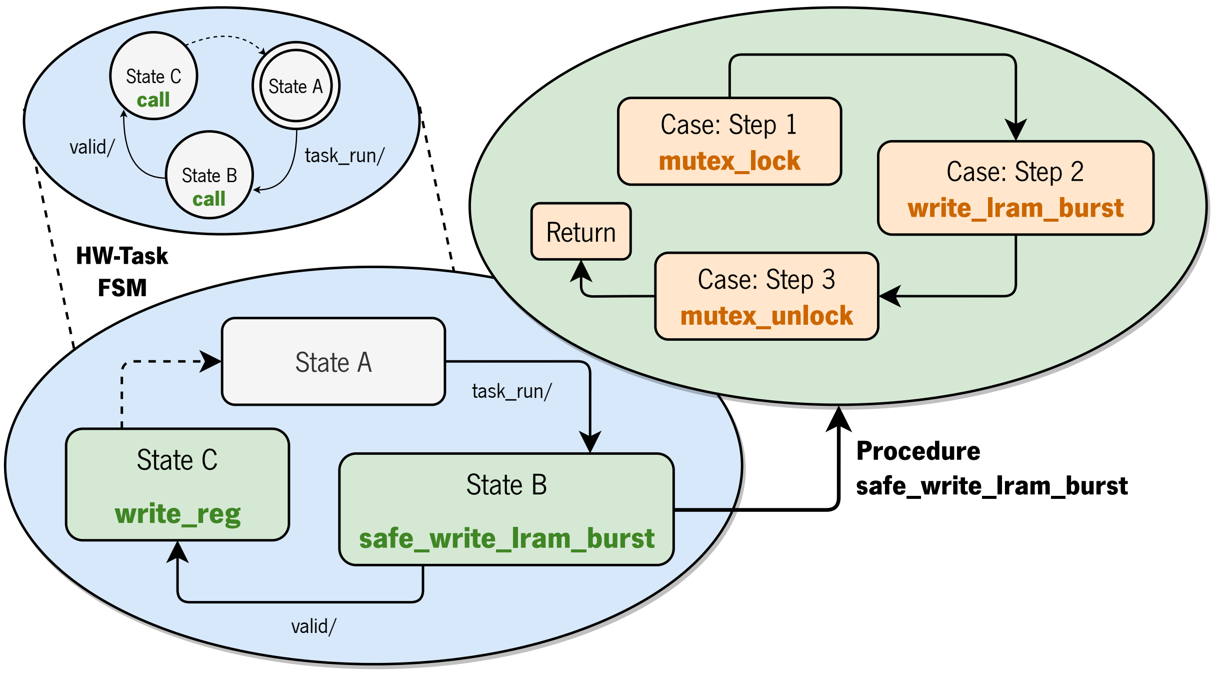


Figure 5.2.1: HAL-ASOS' Hardware Task: Composite Procedure Model.

By increasing the complexity of the task, the procedure paradigm is bounded to fall onto composite procedures to achieve overall-system functionality. The way that HAL-ASOS implements composite procedures is extremely hindered by target-platform resources. This means that the latter implements procedures in a way that ultimately, at the last stage of its design methodology (*c.f* section 3.1.1), uses almost, if not more than, 100% of the resources of the deployment board. This fact is then solved by rearranging the hardware task in a way that fits the board's resource constraints. In these circumstances, a solution that could leverage the resource problem at kernel-level would greatly benefit the accelerator model deployment.

To address the problem one has to analyze the way HAL-ASOS deals with composite procedures. In fact, HAL-ASOS makes use of VHDL *switch* statements (deemed as *cases*) to deal with the composite procedures, meaning that each statement can make a call to a kernel-level system call. This overcomes the limitation of one procedure per control unit state, since there is now logic within a user-level procedure that establishes the active kernel-level procedure based on information coming from the scheduler. Even though, the root of the scalability problem lies with the user-level procedure directly establishing this *case* statements that select the kernel-level procedures, *c.f* fig. 5.2.1. This approach induces the copy of the selection logic for each user-level procedure every time one is called within a state of the hardware task. The scalability problems then arise, after synthesis, with the repeated inferring of ROM/LUT resources from this selection logic within user-level procedures, for each task state. To resolve this issue three main decisions were made:

1. Simplify user-level procedures by passing a procedure ID to the kernel;
2. Add a kernel module responsible for reusing cases based on the procedure ID;
3. Make the module responsible for the switching between the execution modes;

To distinguish between the two executions modes, i.e., scalable and non-scalable, the scalable accelerator execution will be referred as *kernel bounded* and the non-scalable accelerator execution will be referred as *normal* or *not kernel bounded* execution. Additionally, the aforementioned module was named runtime manager, and its RTL design internal architecture is represented in fig. 5.2.4.

A typical execution starts with a user-level procedure call from the hardware task, represented in ① in fig. 5.2.2. This procedure establishes, among other things, the *bound_to_kernel* signal logical level. In the microkernel, the signal serves as logic select to decide if the runtime manager's *case* statements should be used, represented in ② in fig. 5.2.2. In the case of the assertion of *enable_sched* and *bound_to_kernel* in the hardware task, the kernel uses the scheduler and runtime manager, to perform the user-level procedures in the kernel until the end of the procedure, which is determined with the scheduler's *valid* signal assertion, ③ in fig. 5.2.2. Otherwise, if only *enable_sched* is asserted, the kernel behaves as already explained in section 5.1, i.e., by dividing a composite procedure in multiple kernel procedures, but only executing the low-level ones in kernel space. With this, the kernel-level procedure switching is done in the hardware task, and the kernel procedures exchange information with the task through the defined interfaces, namely S00 Kernel and M00 Task, ④ in fig. 5.2.2.

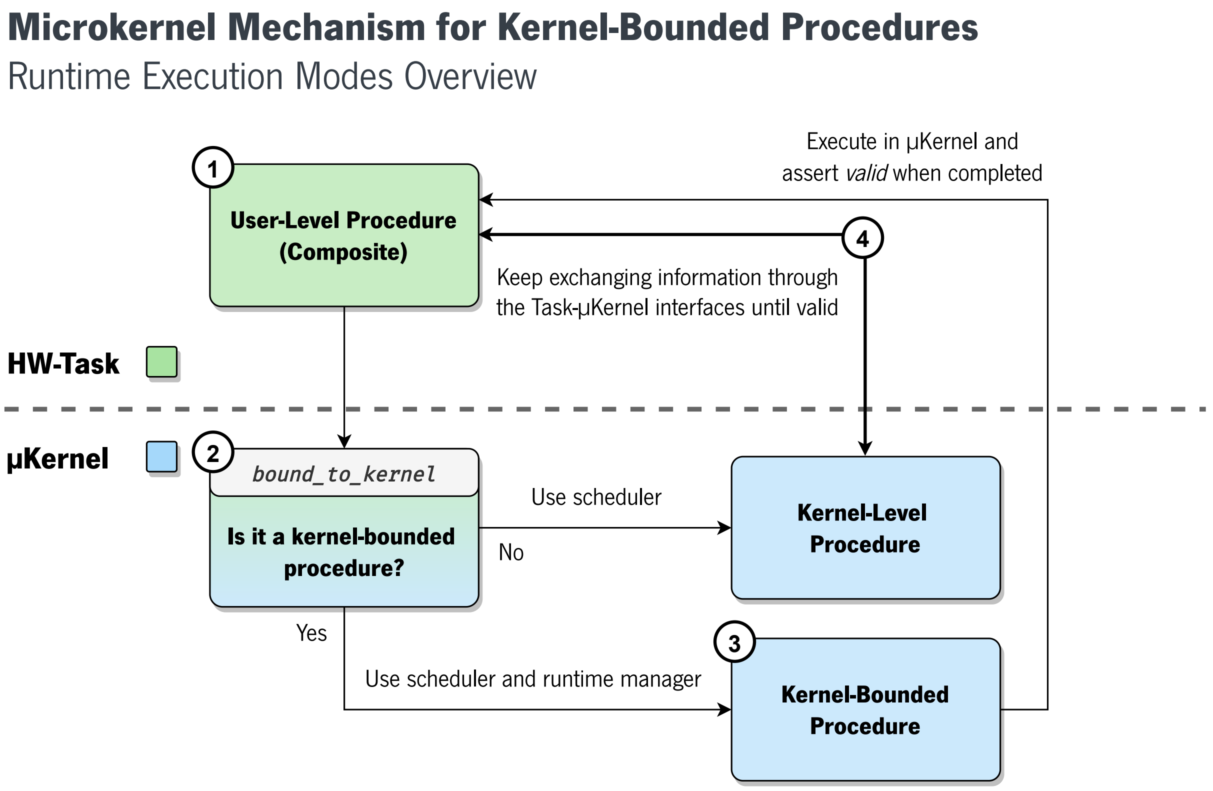

Microkernel Mechanism for Kernel-Bounded Procedures
Runtime Execution Modes Overview
1
User-Level Procedure (Composite)
Execute in µKernel and assert *valid* when completed
4
Keep exchanging information through the Task-µKernel interfaces until valid
HW-Task
µKernel
2
bound_to_kernel
Is it a kernel-bounded procedure?
Use scheduler
No
Kernel-Level Procedure
Yes
Use scheduler and runtime manager
3
Kernel-Bounded Procedure

Figure 5.2.2: Overview of the microkernel mechanism to execute kernel-bounded user-level procedures. User procedures are executed in the microkernel depending on the set procedure ID when the *bound_to_kernel* signal is asserted, otherwise a non-scalable user procedure can be executed in the hardware task by exchanging information with the microkernel through the dedicated interfaces. In either case, the *enable_sched* is considered implicit since one is referring to composite procedures.

Listing 5.12 represents the kernel-bounded version of the file subscribe extended feature. Its purpose is to get a descriptor regarding a named-pipe, socket, or binary file on the host system side. It was named file subscribe instead of file open, since it is bound to KIVIO operation, i.e., the hardware task always calls the same procedure and lets KIVIO handle the matching with a descriptor based on the needs of the overlaying user-level software application. Each extended feature always uses the procedure in line 5 to establish the active kernel services, in this case, the scheduler and the runtime manager, with the assertion of *enable_sched* and *resched_req*, and the assertion of *bound_to_kernel*, respectively. Additionally. it also specifies the procedure ID, in line 19, forwards the KFIFO command ID to the create command procedure, in line 22, while it retrieves the created 64-bit command, in the same line, and lastly, passes it through the kernel call procedure parameters, in line 23.

Listing 5.12: User Package: Simplified File Subscribe (HDL).

```
 1  -- MORE
 2  -------------------------------------------------------------------------------------------
 3  -- (kernel-bounded) kfifo-extended features (return-blocking)
 4  -------------------------------------------------------------------------------------------
 5  procedure kernel_bounded_safe_kfifo_send_command_wait_return(
 6      signal kernel_call              : out kernel_call_t;
 7      signal kernel_response          : in kernel_response_t;
 8      constant kfifo_cmd_id           : ki_feature_t;
 9      constant hint_cookie            : in natural;
10      constant message_flags_events   : in std_logic_vector(C_KIVIO_FLAGS_WWIDTH-1 downto 0);
11      constant wcount_whence          : in natural;
12      constant woffset                : in natural range 0 to 2**C_KIVIO_WOFFSET_WWIDTH-1) is
13  -------------------------------------------------------------------------------------------
14      variable cmd_data : std_logic_vector(C_MESSAGE_WIDTH-1 downto 0) := (others => '0');
15  begin
16      kernel_call.enable_sched    <= '1';
17      kernel_call.resched_req     <= '1';
18      kernel_call.bound_to_kernel <= '1';
19      kernel_call.procedure_id    <= POLLED_SAFE_SEND_KFIFO_CMD;
20      kernel_call.kfifo_cmd_id    <= kfifo_cmd_id;
21      ---
22      kfifo_create_cmd64(kfifo_cmd_id,hint_cookie,message_flags_events,wcount_whence,woffset,
            cmd_data);
23      kernel_call.procedure_parameters <= cmd_data;
24      ---
25  end procedure kernel_bounded_safe_kfifo_send_command_wait_return;
26  -------------------------------------------------------------------------------------------
27  -- MORE
28  -------------------------------------------------------------------------------------------
29  -- (kernel-bounded) kfifo-extended file (socket, named-pipe and binary file) subscribe:
30  -- blocking version (polled to insert in kfifo and polled wait for return)
31  -------------------------------------------------------------------------------------------
32  procedure polled_kernel_bounded_file_subscribe(
33      signal kernel_call              : out kernel_call_t;
34      signal kernel_response          : in kernel_response_t;
35      constant hint_cookie            : in natural;
36      constant message_flags_events   : in std_logic_vector(C_KIVIO_FLAGS_WWIDTH-1 downto 0);
37      constant wcount_whence          : in natural;
38      constant woffset                : in natural range 0 to 2**C_KIVIO_WOFFSET_WWIDTH-1;
39      signal returned_cookie          : out natural) is
40  -------------------------------------------------------------------------------------------
41  begin
42      kernel_bounded_safe_kfifo_send_command_wait_return(kernel_call,kernel_response,
            KI_SUBSCRIBE,hint_cookie,message_flags_events,wcount_whence,woffset);
43      returned_cookie <= to_integer(unsigned(kernel_response.procedure_return(
            KIVIO_RET_COOKIE'RANGE)));
44      ---
45  end procedure polled_kernel_bounded_file_subscribe;
46  -------------------------------------------------------------------------------------------
```

5.2.1 Runtime Management

To achieve execution mode switching and also scalability in user procedures the runtime manager module was added into the scheduler-microprogram scheme of fig. 5.1.2. The result can be seen in fig. 5.2.3. As portrayed, the runtime manager receives the kernel call coming from the hardware task, deemed as *i_kernel_call*, and also the kernel response generated by export response procedure (P3). To construct the kernel response that enters the runtime manager the procedure P3 can receive the syscall output from the microprogram or the scheduler, depending on the microinstruction flow chosen, i.e., if one is considering standalone-microprogram execution or scheduler-dictated execution, respectively. Both the kernel call coming from the S00 Task interface and the internal kernel response enter the runtime manager to ultimately produce an active kernel call and also an active kernel response. This means that, the execution mode selected, i.e., kernel bounded or normal, selects the which kernel call and kernel response should be used.

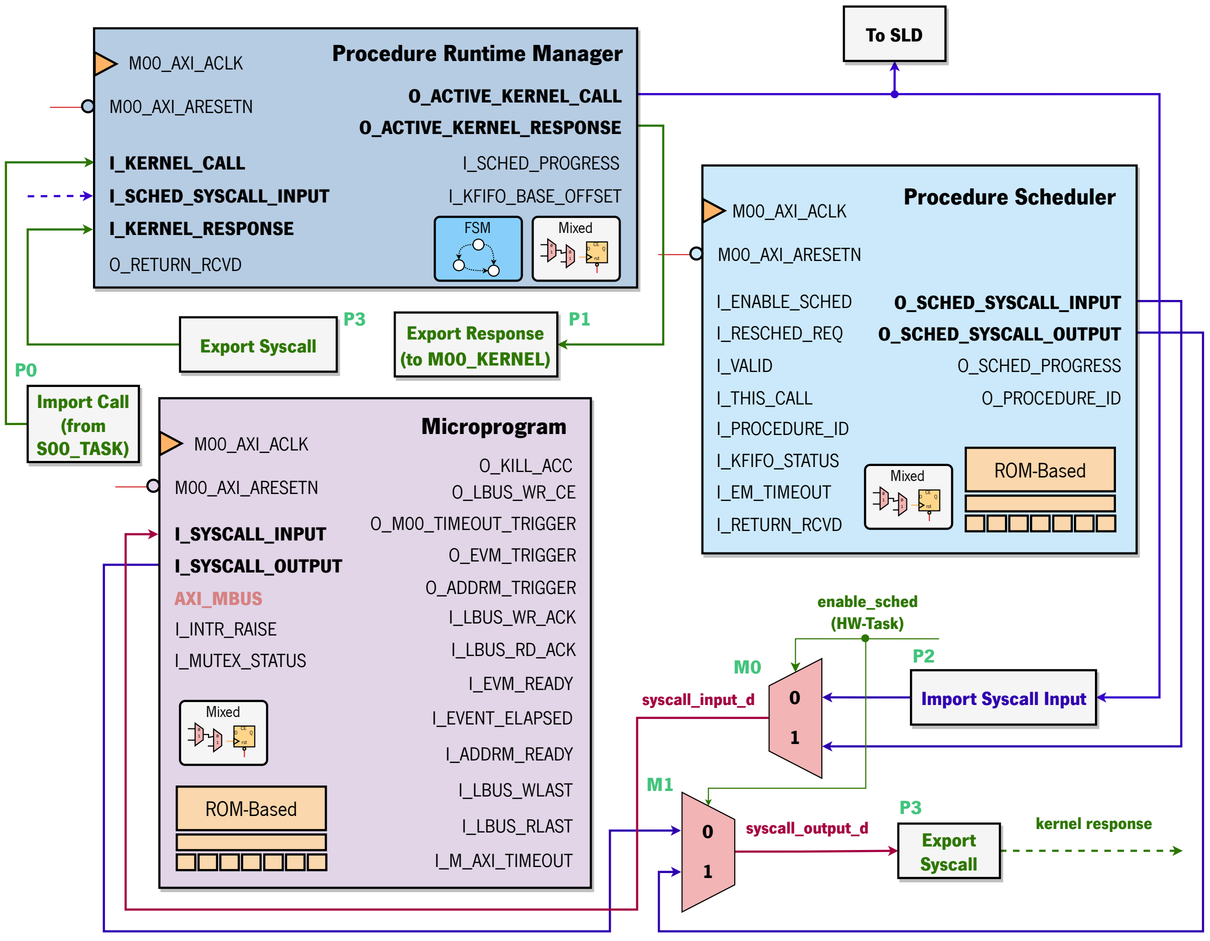


Figure 5.2.3: Runtime Management Overview.

Note that the kernel response stays nearly the same on both execution modes because it is mainly given by the scheduler-microprogram pair, in case of scheduler-dictated execution, or only the microprogram, in standalone microprogram execution, and also the SLD. As described by fig. 4.3.1, the kernel response includes information regarding the task's *valid* bit, the current kernel indexes, procedure ID, scheduler progress, procedure return, syscall ID, and syscall return arguments. On another note, the kernel call is different because shifting between kernel-bounded execution and normal execution, i.e., the one employed by HAL-ASOS and defined in fig. 5.2.1, stipulates the place where the kernel call information is coming from. Thus, with normal execution, one has kernel call information coming from the hardware task, while in kernel-bounded execution, one has information regarding kernel calls coming from the runtime manager. This means that, besides selecting the currently active kernel call, represented in M1 (*c.f* fig. 5.2.4), the runtime manager also has to generate its own kernel calls depending on the procedure ID passed to it as input parameter. This is represented by the M0 and M0.01 case chain of fig. 5.2.4. The normal kernel calls are kept if the aforementioned *bound_to_kernel* signal is not asserted (M1), otherwise, the runtime manager uses its own kernel calls with the *bound_to_kernel* asserted (also in M1). The HDL representation of the runtime manager can be seen in listing 5.13.

Listing 5.13: Runtime Manager (HDL).

```
-------------------------------------------------------------------------------------------
KERNEL_BOUNDED_EXTENDED_FEATURES : process (state,procedure_done_i,bounded_kernel_call.
    procedure_id,I_SCHED_PROGRESS,I_KERNEL_CALL.enable_sched,I_KERNEL_CALL.resched_req,
    I_KERNEL_CALL.procedure_id,I_KERNEL_CALL.bound_to_kernel,I_SCHED_SYSCALL_INPUT.
    syscall_id,I_KERNEL_RESPONSE,timeout_i,remaining_time_i,kfifo_args_buffer_q,
    procedure_done_i,return_arg_i,mutex_status_i,args_in_masked_q,I_KFIFO_BASE_OFFSET,
    kfifo_rets_buffer_q,rets_out_masked_q,kfifo_index_q,kfifo_index_d,kfifo_args_buffer_d,
    args_in_masked_d,kfifo_rets_buffer_d,rets_out_masked_d)
-------------------------------------------------------------------------------------------
begin
    -- MORE
    ---------------------------------------------------------------------------------------
    case state is
        when S0_IDLE =>
            bounded_kernel_call_sync(bounded_kernel_call,I_KERNEL_CALL,
                I_SCHED_SYSCALL_INPUT);
            next_state <= S1_RUNNING;
        when S1_RUNNING =>
            bounded_kernel_call_sync(bounded_kernel_call,I_KERNEL_CALL,
                I_SCHED_SYSCALL_INPUT);
            -------------------------------------------------------------------------------
            case bounded_kernel_call.procedure_id is
                ---------------------------------------------------------------------------
                when PROCEDURE_ZERO                  => null;    --00
                ---------------------------------------------------------------------------
                -- MORE
                ---------------------------------------------------------------------------
                when POLLED_SAFE_SEND_KFIFO_CMD      =>          --02
                    activate_kernel_interface_feature_wait_return(bounded_kernel_call,
                    I_KERNEL_RESPONSE,I_SCHED_PROGRESS,kfifo_args_buffer_d,
                    kfifo_rets_buffer_d,timeout_i,remaining_time_i,50,args_in_masked_d,
                    rets_out_masked_d,procedure_done_i,mutex_status_i,O_RETURN_RCVD,
                    return_arg_i,I_KFIFO_BASE_OFFSET);
                ---------------------------------------------------------------------------
                -- MORE
                ---------------------------------------------------------------------------
                when others                          => null;
            end case;
            -------------------------------------------------------------------------------
            if procedure_done_i = '1' then
                next_state <= S0_IDLE;
            else
                next_state <= S1_RUNNING;
            end if;
        when others => null;
    end case;
end process KERNEL_BOUNDED_EXTENDED_FEATURES;
-------------------------------------------------------------------------------------------
```

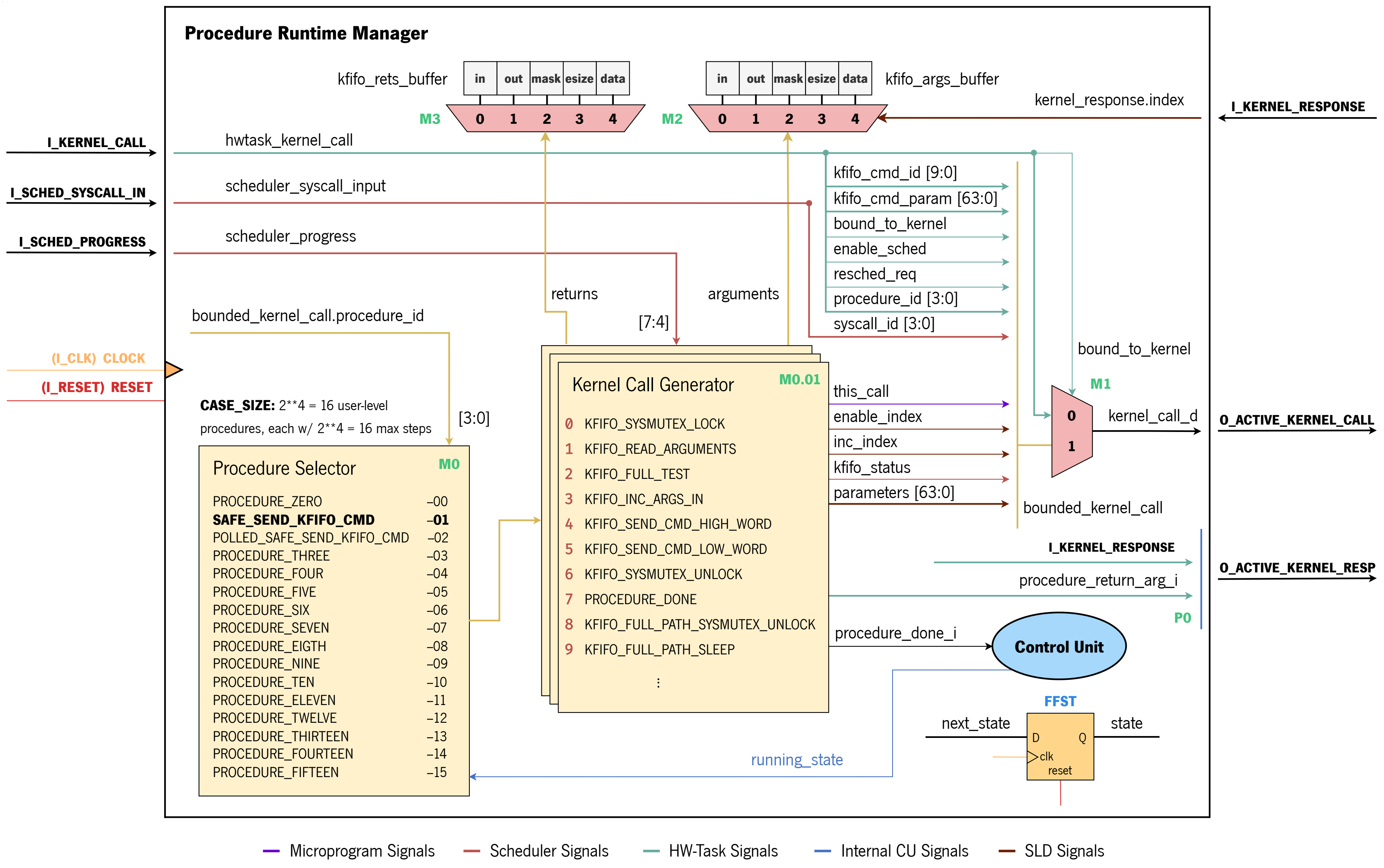


Figure 5.2.4: Runtime Manager RTL Design Internal Architecture (Simplified).

5.3 Diversity-Driven Hardware Task

The idea of a diversity-driven hardware task bases itself on the following premise: **evolution by adaptation**. Following this concept the microprogram should start off with no system calls and be able to issue multiple update requests until one had all low-level system calls in memory. This presupposes that the kernel has mechanisms to handle all the update process, in what concerns to: changing the microcode memory contents at runtime, grant partial host-system access to the microcode memory, and also have error handling capabilities. This section explores these concepts and the design and implementation choices that revolve around turning a normal hardware task into a diversity-driven one, capable of coping with change.

5.3.1 Microcode Storage Model Migration

The first thing to consider is migrating the microcode memory from ROM to RAM, so one is able to write into memory and perform microcode updates. As forementioned in section 2.3.2, the literature also progressed in this direction to endow CPUs with capabilities against errors and also to deploy security measures, *c.f* section 2.3.2, so this was the logical path to take regarding HAL-ASOS hardware task improvements. With this in mind, the only change to make to the microprogram was substituting the ROM presented in section 4.2.4, with a true dual port RAM similar to the one used to construct the local memory, *c.f* section 4.5.3. The microcode RAM needs to be dual port since one intends to use port B to specify a program address when executing from it, while using port A to specify addresses and input data regarding updates. The updatable microprogram scheme is represented in fig. 5.3.1. Additionally, the modifications done to the microprogram HDL are represented in listing 5.14.

Listing 5.14: Microprogram RAM Instantiation (HDL).

```
mem_addr(SYSCALL_MSB'RANGE) <= std_logic_vector(to_unsigned(SYSCALL_T'POS(I_SYSCALL_INPUT.
    SYSCALL_ID),SYSCALL_MSB'LENGTH));
--------------------------------------------------------------------------------
SYSCALL_RAM0 : entity TDP_RAM
--------------------------------------------------------------------------------
generic map(
RAM_W_WIDTH => 16,
RAM_DEPTH => 64)
port map(
--------------------------------------------------------------------------------
-- System Channel --
--------------------------------------------------------------------------------
I_ADDR_A     => I_SYS_ADDR_A, -- 6 bits
I_DIN_A      => I_SYS_DIN_A(15 downto 0), -- 16 bits
I_CLK_A      => I_CLK,
I_WR_CE_A    => I_SYS_WR_CE_A,
I_CS_A       => I_SYS_CS_A,
O_DOUT_A     => OPEN,
O_WR_ACK_A  => O_SYS_WR_ACK_A,
O_RD_ACK_A  => O_SYS_RD_ACK_A,
------------------------------------------------------------------------------------------
-- Microprogram Channel --
------------------------------------------------------------------------------------------
I_ADDR_B    => mem_addr, -- 6 bits
I_DIN_B     => (others => '0'), -- 16 bits
I_CLK_B     => I_CLK,
I_WR_CE_B   => '0',
I_CS_B      => '1',
O_DOUT_B    => mem_dout, -- 16 bits
O_WR_ACK_B  => OPEN,
O_RD_ACK_B  => OPEN
);
------------------------------------------------------------------------------------------
```

To provide scheduler updates, one can also migrate the scheduler's microcode memory from ROM to RAM, following the design of fig. B.9. This approach was not left in the final implementation since it required mechanism that ensures coherency between the order of the low-level syscalls in memory and the datapath associated with them, i.e., at the current research state one has a kernel with capabilities to receive microcode scheduler updates but the datapath is static and, therefore, does not follow the changes introduced in the vertical microcode. This could be resolved with an some sort of arbiter that decided the active datapath elements based on the system call ID dictated by the scheduler, similarly to the SLD element, but regarding the runtime manager parameter stipulation. Another approach could be the introduction of DPR to make use of dynamic datapaths that accompanied scheduler updates.

5.3.2 Host System Access

Another factor to take into consideration when modifying the architecture to a diversity-driven one is the mapping of the microcode RAM onto one of the available interfaces. The interface chosen for this was the S00 Control one. At the time, this interface is divided into four pages but only two pages are used, one for hardware resources, and another for interface registers, both of them not occupied to the fullest. This means that more registers can still be mapped on page zero and page one, and that one still has two more pages (unmapped) that account for a possible microkernel expansion, leaving room to map the microprogram and the even the scheduler here. The microprogram memory has 64 words of 16 bits, and the microkernel was designed to also handle a machine word of 64 bits, but currently the microkernel presupposes a machine word of 32 bits. This means that the microprogram could be mapped to the equivalent of two pages, i.e., 32 words of 32 bits, but this would require temporary storage in the microkernel to handle multiple AXI slave write transactions of 32 bits, and a way to process them as single-word writes of 16 bits (in normal format) to the microprogram memory, or even as a single memory write of multiple 16-bit words (in burst format). This temporary storage, could be a FIFO if one used the

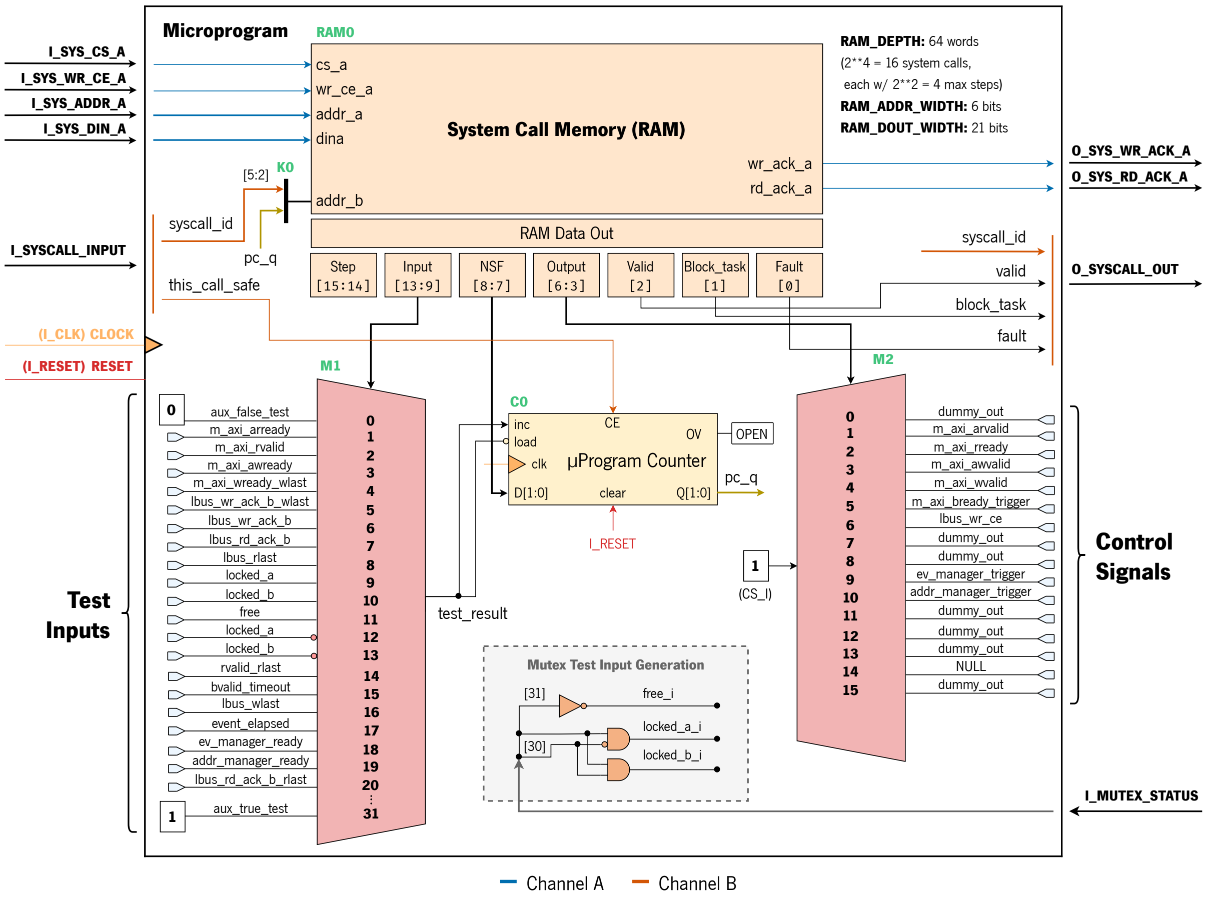


Figure 5.3.1: Updatable Microprogram RTL Design Internal Architecture.

S00 Control interface, or even the LRAM acting as a microinstruction cache if one used the S01 Data interface. Since both of these interfaces are single-word rated, performing burst updates would require a ZCU-like unit to pass the microinstructions from the cache or FIFO to the microprogram RAM. Due to the complexity of this scheme, one opted to view the microcode update as a write of 64 words of 32 bits through the S00 Control interface, whilst discarding random noise added to the higher sixteen bits of the update payload, *c.f* fig. 6.2.2, to avoid (side-channel) attacks that exploit the microcode-based systems with meticulous payload analysis. Refer to section 7.1 for more information on the direction of future research. The microprogram memory was mapped alongside the slave decoder element, receiving its data directly from the AXI4 lite interface module for the S00 Control interface, *c.f* listing 5.15.

Listing 5.15: Microkernel's Top-level: Updatable Microprogram Instantiation (HDL).

```
------------------------------------------------------------------------------------------
UP : entity ram_uProgram
------------------------------------------------------------------------------------------
port map(
I_CLK                    => m00_axi_aclk,
I_RESET                  => RESET_I,
------------------------------------------------------------------------------------------
I_SYSCALL_INPUT          => syscall_input_d,
O_SYSCALL_OUTPUT          => SYSCALL_OUTPUT,
------------------------------------------------------------------------------------------
I_SYS_CS_A                => s00_addr_i(8),
I_SYS_WR_CE_A             => s00_wr_ce_i,
I_SYS_ADDR_A              => s00_addr_i(POW2(C_UP_RAM_DEPTH)+1 downto 2),
I_SYS_DIN_A               => s00_txdata_i,
O_SYS_WR_ACK_A            => ram_uprog_wr_ack,
O_SYS_RD_ACK_A            => ram_uprog_rd_ack,
------------------------------------------------------------------------------------------
-- MORE (Microprogram Test Inputs)
------------------------------------------------------------------------------------------
-- MORE (Microprogram Control Signals)
------------------------------------------------------------------------------------------
```

5.3.3 Error Handling and Update Mechanism

All that is left to have an adaptable hardware task is the introduction of error handling capabilities. These refer to the detection of a internal microprogram fault and the generation of an update request. With this in mind, one stipulated the bit zero of all microprogram instructions as a fault bit. The initial configuration of the microprogram memory should initialize the RAM with zeros and ensure these fault bits are asserted for the unmapped microinstructions. As seen in listing 5.15 the fault bit is propagated to the microkernel's top-level through the output signal *o_syscall_output*. Its assertion makes the kernel's FSM to go from a running state to the fault state, activating a kernel fault signal. This signal serves as an interrupt source input of the LINTC to generate an IRQ associated with the update. The HDL pertaining to the kernel's FSM and the enabling of the interrupt source can be seen in listing 5.16. For more information on the kernel's FSM refer to section 4.2.2.

Listing 5.16: Microkernel Control FSM (HDL).

```
------------------------------------------------------------------------------------------
UKERNEL_CONTROL_FSM : process(state,ukernel_control_i,ACTIVE_KERNEL_CALL.task_done,
    kill_accelerator_i,SYSCALL_OUTPUT.fault)
------------------------------------------------------------------------------------------
begin
-- MORE
    case state is
        ----------------------------------------------------------------------------------
        when S0_K_READY=>
        ----------------------------------------------------------------------------------
            if(ukernel_control_i(C_UKERNEL_CTRL_RUN) = '1') then
                next_state <= S1_K_RUNNING;
            elsif (ukernel_control_i(C_UKERNEL_CTRL_RESET) = '1') then
                next_state <= S3_K_RESTART;
            end if;
        ----------------------------------------------------------------------------------
        when S1_K_RUNNING =>
        ----------------------------------------------------------------------------------
            task_run_i <= '1';
            if(SYSCALL_OUTPUT.fault = '1') then
                next_state <= S2_K_ERROR;
            elsif(ukernel_control_i(C_UKERNEL_CTRL_RUN) = '1') then
                next_state <= S1_K_RUNNING;
            elsif(ukernel_control_i(C_UKERNEL_CTRL_RESET) = '1') then
                task_reset_i <= '1';
                next_state <= S3_K_RESTART;
            elsif(ACTIVE_KERNEL_CALL.task_done = '1') then
```

```
                next_state <= S0_K_READY;
            elsif(kill_accelerator_i = '1') then
                next_state <= S4_K_DEAD;
            end if;
        ---------------------------------------------------------------------------------
        when S2_K_ERROR =>
        ---------------------------------------------------------------------------------
            task_error_i <= '1';
            kernel_fault_i <= "001"; -- update irq
            next_state <= S0_K_READY;
        ---------------------------------------------------------------------------------
        when S3_K_RESTART =>
        ---------------------------------------------------------------------------------
            task_reset_i <= '1';
            next_state <= S0_K_READY;
        ---------------------------------------------------------------------------------
        when S4_K_DEAD =>
        ---------------------------------------------------------------------------------
            accelerator_dead_i <= '1';
            next_state <= S4_K_DEAD;
        ---------------------------------------------------------------------------------
        when others => null;
    end case;
end process UKERNEL_CONTROL_FSM;
-----------------------------------------------------------------------------------------
-- MORE (State Register)
```

This interrupt source is then multiplexed with the interrupt source coming from the SLD and pertaining to the *lintc_write* system call, *c.f* section 4.2.3, to generate the interrupt source of the LINTC module, *c.f* listing 5.17.

Listing 5.17: Microkernel's Top-level: Interrupt Source Selection and LINTC Instantiation (HDL).

```
-----------------------------------------------------------------------
INTR_SOURCE_M2 : process (task_error_i,lintc_intr_src_i,kernel_fault_i)
-----------------------------------------------------------------------
begin
    intr_source_d <= "000";
    case task_error_i is
        when '0' =>
            intr_source_d <= lintc_intr_src_i;
        when '1' =>
            intr_source_d <= kernel_fault_i;
        when others =>
            null;
    end case;
end process INTR_SOURCE_M2;
-----------------------------------------------------------------------
LINTC : entity INTR_CONTROLLER
-----------------------------------------------------------------------
port map(
I_CLK        => s00_axi_aclk,
I_RESET      => RESET_I,
I_CS(0)      => page0_dec_word_select_i(C_WORD6_BIT),
I_CS(1)      => page0_dec_word_select_i(C_WORD7_BIT),
I_WR_CE      => page0_dec_wr_ce_i,
I_RD_CE      => page0_dec_rd_ce_i,
I_RXDATA     => page0_dec_txword_i,
I_INTR_SRC   => intr_source_d,
O_WR_ACK     => lintc_wr_ack_i,
O_RD_ACK     => lintc_rd_ack_i,
O_CONTROL    => lintc_control_i,
O_STATUS     => lintc_status_i,
O_IRQ_PIN    => O_IRQ_PIN,
O_INTR_RAISE => intr_raise_i
);
-----------------------------------------------------------------------
```

6. Experimental Results

The verification of a project's functionality is very important to assure system reliability, i.e., the continuity of service correctness [15], or simply not doing the wrong thing [14], whilst it maintains its integrity, i.e., remains unaltered through several changes [5]. Therefore, having a good testing environment is crucial to ensure the system is fully functional, passes verification tests, and regression tests, i.e., a system that does not degrade or reverts unexpectedly as one introduces a new feature. This section will introduce the methods used to test the system implemented in the dissertation and their results.

6.1 Accelerator Model Resource Utilization

To evaluate the overall resource utilization of the accelerator model, one selected a PoC board, in this case, the Zybo Z7-10 board. The tests were conducted using the final versions of the hardware task and microkernel IPs, using the block designs of fig. B.4 and fig. B.5. The resource utilization for the modules of the dissertation can be found in table 6.1.

6.1.1 Stand-Alone Kernel Single-Task

Following HAL-ASOS in what relates to the design methodology of the hardware modules, one achieved a functional microkernel with minimal resource footprint. As a whole, the microkernel only uses minimal resources considering it is based on the architecture of a microprogrammable CPU with kernel-level and user-level abstractions. With this in mind, the microkernel represents around 10% of the board's LUTs and almost 3% of the board's flip-flops. Additionally, it also uses 3 F7 muxes (less than 1%) and 2.5% of the board's total BRAM tiles. Regarding the hardware task, one made two distinct implementations that achieve the same results of the pipe tests presented in section 5.1.2. The first one, deemed as normal, follows the hardware task model of HAL-ASOS, *c.f* fig. 5.2.1, which does not reutilize cases and, thus, infers combinational logic for each task state containing composite procedures. The other approach, kernel-bounded, aforementioned in section 5.2, makes use of the runtime manager module developed to reutilize the same case statement and avoid unnecessary inference of combinational logic. The experimental results show that for the functional example of section 5.1.2, the kernel-bounded version infers less F7/F8 muxes than the normal version at the cost of a slight increase in LUT usage. Specifically, the kernel-bounded hardware task drops the usage of both mux cells by about 73%, while only incrementing the usage of LUT cells by about 11%. Overall, in this case, the proposed accelerator model represents nearly 29% of the target's LUTs, around 16% of the target's flip-flops, less than 1% of F7/F8 muxes, about 1.5% of BRAMs, about 2% of the total LUTRAMs, and decreases slice usage by approximately 19%.

Table 6.1: Post-Implementation Accelerator Model Resource Utilization targeting Zybo Z7-10.

Module	**LUTs**	**FFs**	**F7 Muxes**	**F8 Muxes**	**BRAMs**	**Slices**
	cnt. (%)	cnt. (%)	cnt. (%)	cnt. (%)	tiles (%)	cnt. (%)
Microkernel*	1801 (10.23)	939 (2.67)	3 (0.03)	0 (0.00)	1.5 (2.50)	567 (12.89)
Interface Regs.	62 (0.35)	98 (0.28)	0 (0.00)	0 (0.00)	0 (0.00)	57 (1.30)
Kernel Regs.	5 (0.03)	37 (0.11)	0 (0.00)	0 (0.00)	0 (0.00)	20 (0.45)
LINTC	15 (0.09)	34 (0.10)	0 (0.00)	0 (0.00)	0 (0.00)	18 (0.41)
LMUTEX	76 (0.43)	99 (0.28)	0 (0.00)	0 (0.00)	0 (0.00)	49 (1.11)
LRAM	1 (<0.01)	4 (0.01)	0 (0.00)	0 (0.00)	1 (1.67)	4 (0.09)
Runtime Manager	299 (1.70)	282 (0.80)	0 (0.00)	0 (0.00)	0 (0.00)	201 (4.57)
S00 Interface	183 (1.04)	88 (0.25)	1 (0.01)	0 (0.00)	0 (0.00)	84 (1.91)
S01 Interface	22 (0.13)	58 (0.16)	0 (0.00)	0 (0.00)	0 (0.00)	22 (0.50)
Scheduler	863 (4.90)	10 (0.03)	2 (0.02)	0 (0.00)	0 (0.00)	330 (7.50)
SL Datapath	123 (0.70)	76 (0.22)	0 (0.00)	0 (0.00)	0 (0.00)	113 (2.57)
Slave Event Manager	62 (0.35)	46 (0.13)	0 (0.00)	0 (0.00)	0 (0.00)	34 (0.77)
SYSMUTEX	87 (0.49)	99 (0.28)	0 (0.00)	0 (0.00)	0 (0.00)	54 (1.23)
Microprogram*	24 (0.14)	4 (0.01)	0 (0.00)	0 (0.00)	0.5 (0.83)	19 (0.43)
HWPipeCom4 (KB)	2046 (11.63)	3200 (9.09)	34 (0.39)	15 (0.34)	0 (0.00)	928 (21.09)
HWPipeCom4 (N)	1848 (10.50)	3284 (9.33)	128 (1.45)	56 (1.27)	0 (0.00)	1149 (26.11)
Difference (N $\rightarrow$ KB)	+10.71%	-2.56%	-73.40%	-73.20%	0.00%	-19.23%
HWPipeCom4_AES (KB)	3298 (18.74)	3029 (8.61)	41 (0.47)	0 (0.00)	0 (0.00)	1616 (36.73)
Accelerator Model***	5014 (28.49)	5573 (15.83)	37 (0.42)	15 (0.34)	1.5 (2.50)	1850 (42.05)

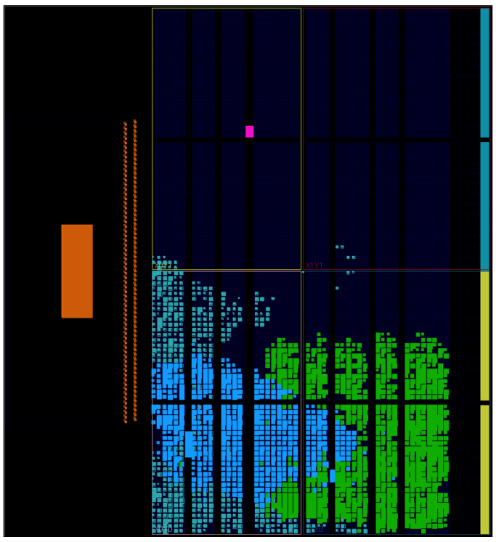
(KB) Kernel-Bounded Execution **(N)** Normal Execution**

*Considering the updatable version of the microprogram.

**Normal execution refers to the one employed by HAL-ASOS [62].

***Including additional processing system, reset, and interconnect logic and a kernel-bounded hardware task (*XOR*).

These values also presuppose the consideration of the logic regarding the processing system, the AXI peripheral, and memory interconnects, and the reset generation, while using a kernel-bounded hardware task. The 1.5% of BRAM usage is supplemented by the correct inference of local memory (LRAM), and the microprogram RAM. Finally, the accelerator model does not utilize LUTRAMs. In any case, the hardware task is always the most resource-demanding entity in the accelerator model, specifically its datapath, which usually contains the offloaded algorithm, e.g., AES, initially implemented in the software application, following the HAL-ASOS methodology, *c.f* fig. 3.1.1. Therefore, incrementing the size of the hardware task's datapath might originate a migration to a target board with more resources in certain cases. This test (*c.f* table 6.1) refers to the use of 4 extended procedures, and this applies for both the kernel-bounded and normal execution modes.


(a) Kernel-Bounded 32-Word XOR Encryption, 4 Ext. Procedures.

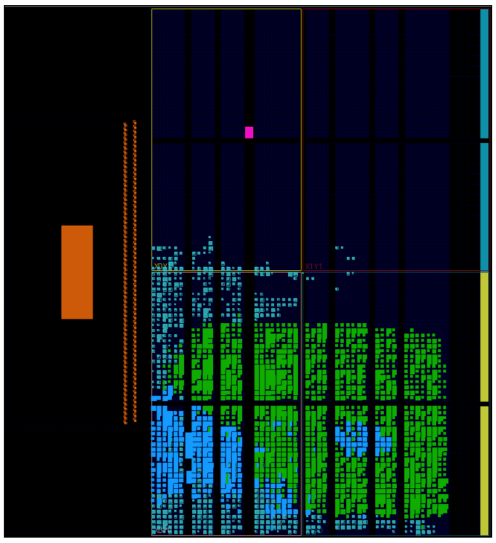

(b) Normal 32-Word XOR Encryption, 4 Ext. Procedures

Figure 6.1.1: Pipe Hardware Task Device Resource Utilization Scenarios for the Encryption of 32 Words.

Additionally, one can discuss resource utilization regarding fig. 6.1.1, which represents the visualization of resource usage and placement on the selected PoC board, in this case, Zybo Z7-10. The fig. 6.1.1 refers to a scheme in which there is a single kernel controlling a single hardware task. With this in mind, fig. 6.1.1a and fig. 6.1.1b depict resource usage when executing with a kernel-bounded hardware task and a normal one, respectively. The information present describes what was already analyzed in table 6.1, which is the use of more mux cells when executing under normal task conditions, compared to kernel-bounded execution. On another note, in fig. 6.1.1b is more perceptible that, in this case (4 extended procedures), the number of FPGA slices used is greater. This is confirmed by analyzing fig. 6.1.1a and fig. 6.1.1b close to each other and comparing the dispersion of the cells. Moreover, one can analyze maintaining the same microkernel but using the two versions of the pipe hardware task with more extended procedures. With this, it was possible to conclude that, for 8 extended procedures, the LUT usage still increases, with a greater percentage increase, probably due to cell optimizations. In addition, the flip-flop usage decrease remains basically the same with a decrease of about 2.53%. The F7 muxes at this phase start to increase, while the F8 muxes decrease drops from around 73.20% to 19.23%. Regarding the use of pipe hardware tasks with 16 extended procedures, one noticed that the LUT usage difference once again becomes close to the initial values with 4 extended procedures, sitting around a 13% increase. For this case, the flip-flop usage difference shifts from a 3% decrease to a 4% decrease. The F7 mux usage passes from a 15% increase to a 18% increase, and the utilization of the F8 muxes decreases a bit more (comparatively) passing from a 19% decrease to a 29% decrease.

Table 6.2: Post-Implementation Tasks' Resource Utilization targeting Zybo Z7-10.

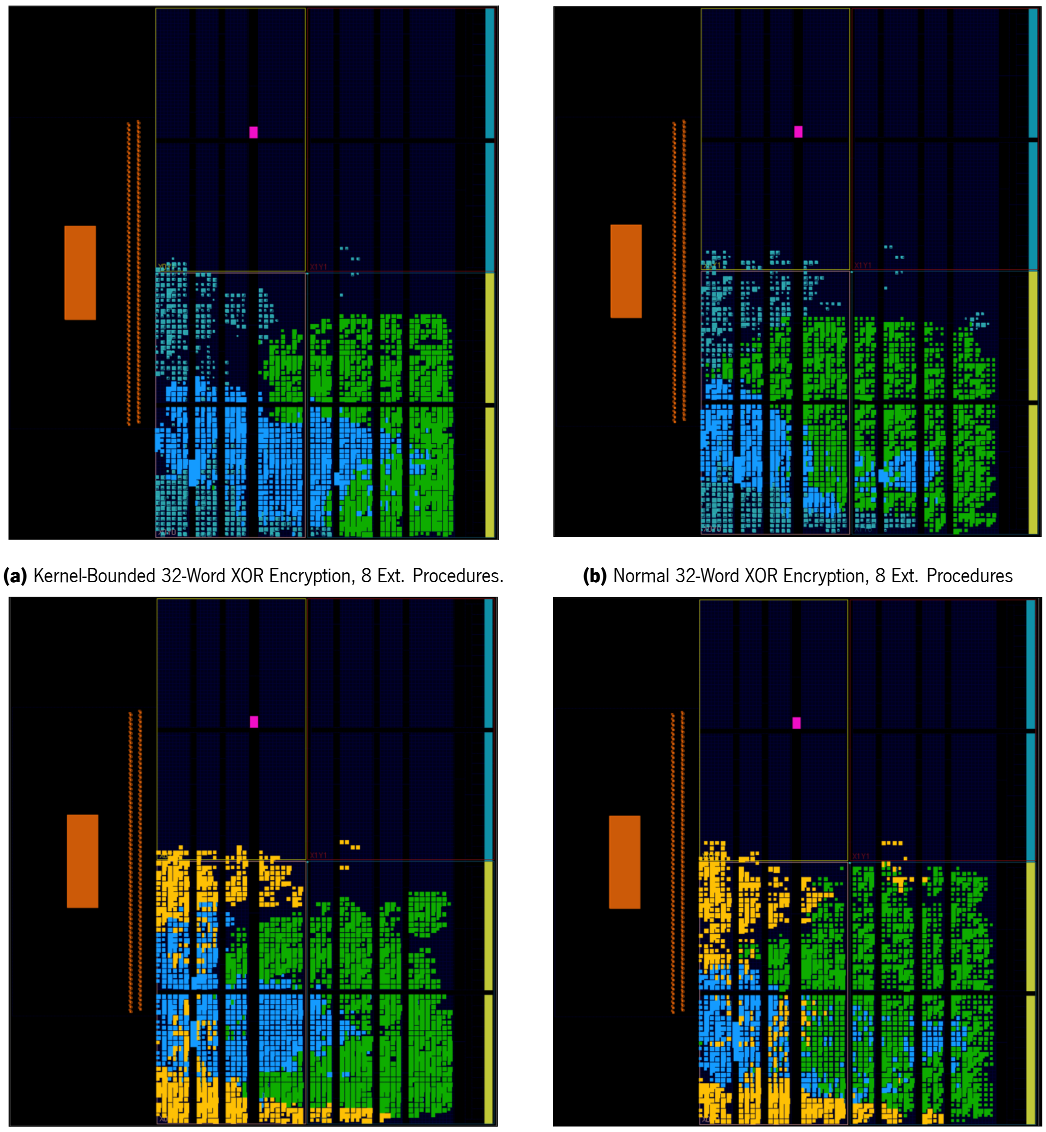
Module	**LUTs**	**FFs**	**F7 Muxes**	**F8 Muxes**	**BRAMs**	**Slices**
	cnt. (%)	cnt. (%)	cnt. (%)	cnt. (%)	tiles (%)	cnt. (%)
HWPipeCom8 (KB)	1962 (11.15)	3201 (9.09)	147 (0.39)	42 (0.34)	0 (0.00)	810 (18.41)
HWPipeCom8 (N)	1492 (8.48)	3284 (9.33)	128 (1.45)	52 (1.18)	0 (0.00)	1094 (24.86)
Difference (N → KB)	+31.50%	-2.53%	+14.84%	-19.23%	0.00%	-25.96%
HWPipeCom16 (KB)	2618 (14.88)	3207 (9.11)	140 (1.59)	30 (0.68)	0 (0.00)	943 (21.43)
HWPipeCom16 (N)	2319 (13.18)	3330 (9.46)	119 (1.35)	42 (0.95)	0 (0.00)	1320 (30.00)
Difference (N → KB)	+12.89%	-3.69%	+17.65%	-28.57%	0.00%	-28.56%

(KB) Kernel-Bounded Execution **(N)** Normal Execution**

**Normal execution refers to the one employed by HAL-ASOS [62].

Overall, the pattern found, when incrementing the number of extended procedures, is that the kernel-bounded hardware task is always able to decrease one type of muxes, and also the total slice usage of the system, with progressively a bigger decrease from 4 to 16 extended procedures, i.e., the difference starts at around 19% decrease, passes to a 26% decrease at 8 extended procedures and ends at a 29% decrease when using 16 extended procedures. These results are complemented by the analysis of cell placement and dispersion of fig. 6.1.2. Both transitions from fig. 6.1.2b to fig. 6.1.2a, and from fig. 6.1.2d to fig. 6.1.2c, represent the usage of fewer FPGA slices by depicting less cell dispersion. Thus, less dispersion means that one has fewer incomplete cells, and, consequently, a better utilization of the resources. For example, let's say that one introduces module changes that originate 5 LUTs after synthesis and implementation.

If a slice has, for example, a maximum of four LUTs and three are already used, this means that the implementation tool might never use the last LUT available. This might happen due to not being able to establish a connection with other cell elements while meeting the timing requirements, and justifies why one should opt by designs with fewer incomplete cells. Regarding fig. 6.1.1 and fig. 6.1.2, the blue area (🟦) refers to the microkernel resources, the green area (🟩) represents the hardware task, the yellow area (🟨) and the teal area (■) represent additional logic referring to interconnect and reset logic, and the orange area (🟧) represents the processing system. Overall, the LUT and F7 Mux increase, in some cases, is related with the case statements used to create the 64-bit command to place in shared memory, future research on this matter should explore a way of also passing these cases to the kernel.



(a) Kernel-Bounded 32-Word XOR Encryption, 8 Ext. Procedures.

(b) Normal 32-Word XOR Encryption, 8 Ext. Procedures

(c) Kernel-Bounded 32-Word XOR Encryption, 16 Ext. Procedures.

(d) Normal 32-Word XOR Encryption, 16 Ext. Procedures.

Figure 6.1.2: Pipe Hardware Task Device Resource Utilization Scenarios for the Encryption of 32 Words, more procedures.

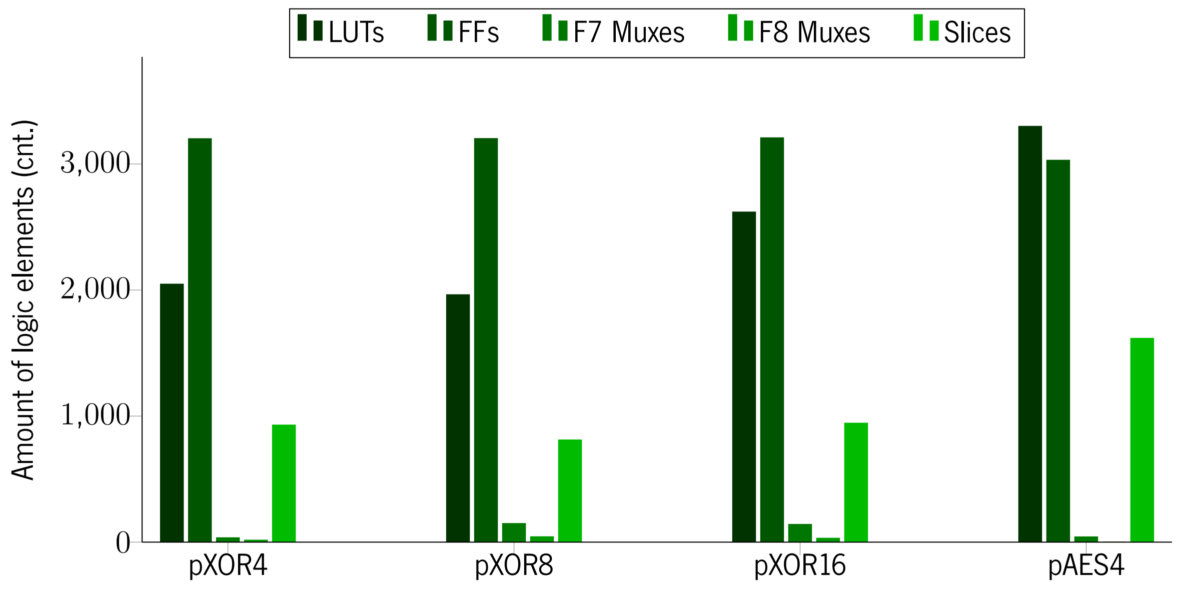


Figure 6.1.3: Kernel-Bounded Pipe Hardware Task Encryption: Resource Utilization Comparison. It is noticeable that kernel-bounded procedures improve the accelerator deployment in terms of scalability since the logic usage remains regular when increasing the amount of extended procedures (*XOR* case), and also when changing to another type of encryption task (AES).

6.1.2 Dual-Kernel Dual-Task

Following, one evaluated the resource implications of shifting from a stand-alone kernel single-task model to a multiple-kernel multiple-task model. The table 6.3 represents the resource usage of a model using two kernels and two hardware tasks. The test were kept at two hardware tasks since the Zybo Z7-10 board was only capable of deploying two accelerators even when using the kernel-bounded versions of the hardware tasks, as seen in fig. 6.1.4. Finally, this dual-kernel dual-task model represented about 58% of the board's LUTs, around 30% of the total flip-flops, almost 2% of F7 muxes, 12% of F8 muxes, 5% of BRAMs to accommodate two local memories and two microprogram memories, nearly 3% of LUTRAMs, and, as a whole, about 77% of the board's slices.

Table 6.3: Post-Implementation Accelerator Model (with a Dual-Task Scheme) Resource Utilization targeting Zybo Z7-10.

Module	**LUTs**	**FFs**	**F7 Muxes**	**F8 Muxes**	**BRAMs**	**LUTRAMs**	**Slices**
	cnt. (%)	cnt. (%)	cnt. (%)	cnt. (%)	tiles (%)	cnt. (%)	cnt. (%)
Accelerator Model*	10262 (58.31)	10506 (29.85)	230 (1.45)	64 (12.36)	3 (5.00)	172 (2.87)	3366 (76.5)

*Considering two updatable microprograms, two kernel-bounded pipe hardware tasks, and additional PS logic.

This means that adding another hardware task to the system would require the migration to a board with more resources, like, for example, a Zynq Ultrascale, to open the possibility of three or more hardware tasks/threads. In the case of the AES HW-Task, one could only deploy one kernel and one task on the Zybo Z7-10 board.

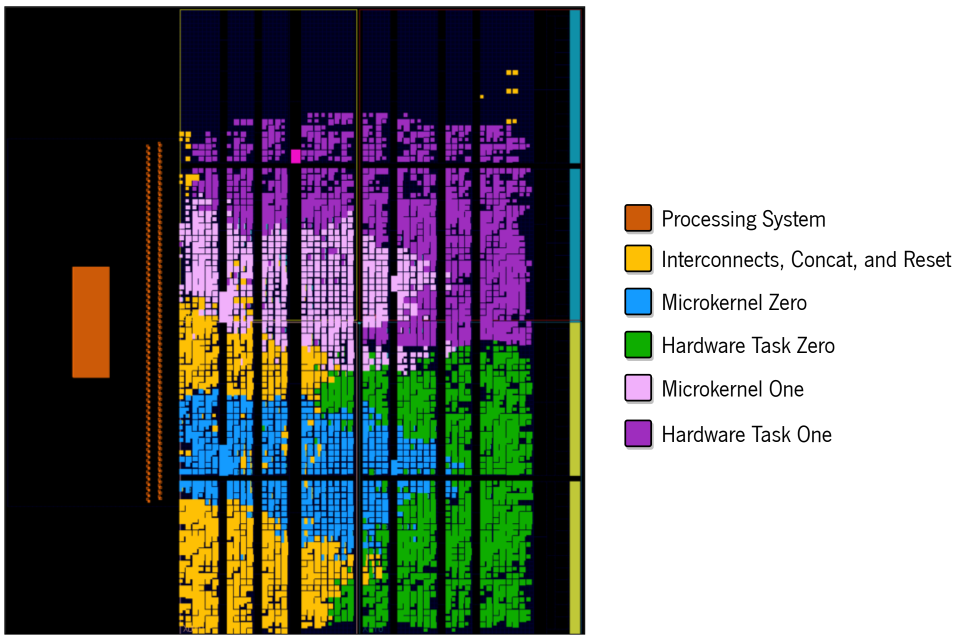


Figure 6.1.4: Pipe Hardware Task Device Resource Utilization with a Dual-Kernel Dual-task Scheme (*XOR* Encryption), block design in fig. B.5. The hardware microkernel associated with a certain hardware task communicates with the host system through shared memory.

6.2 Microprogram Fault Injection

To validate the diversity-driven hardware task developed in the dissertation, one conducted a microprogram fault injection test relying on the HAL-ASOS Link IP [62], fig. 6.2.1. This allows for execution of tests within a full simulation environment, where one can analyze Vivado's behavioral simulation while actually writing and reading to/from QEMU-virtualized memory positions. In the same way, KIVIO is able to write and read through the microkernel's slave interfaces, and interact with the hardware microkernel.

For this, during the initial KIVIO configurations of the accelerator, the microprogram RAM was loaded with zeros, which represents all low-level system calls unmapped, ①. These configurations also include the writing to the interface configuration registers and the masking and enabling of interrupts. Following, KIVIO puts the accelerator in a running state by writing to its control register, this is seen in ②. Following, in ③, the microkernel detects that there is unmapped system calls in the microprogram RAM and issues a kernel fault to the host system, via an update interrupt request. This information is also passed through

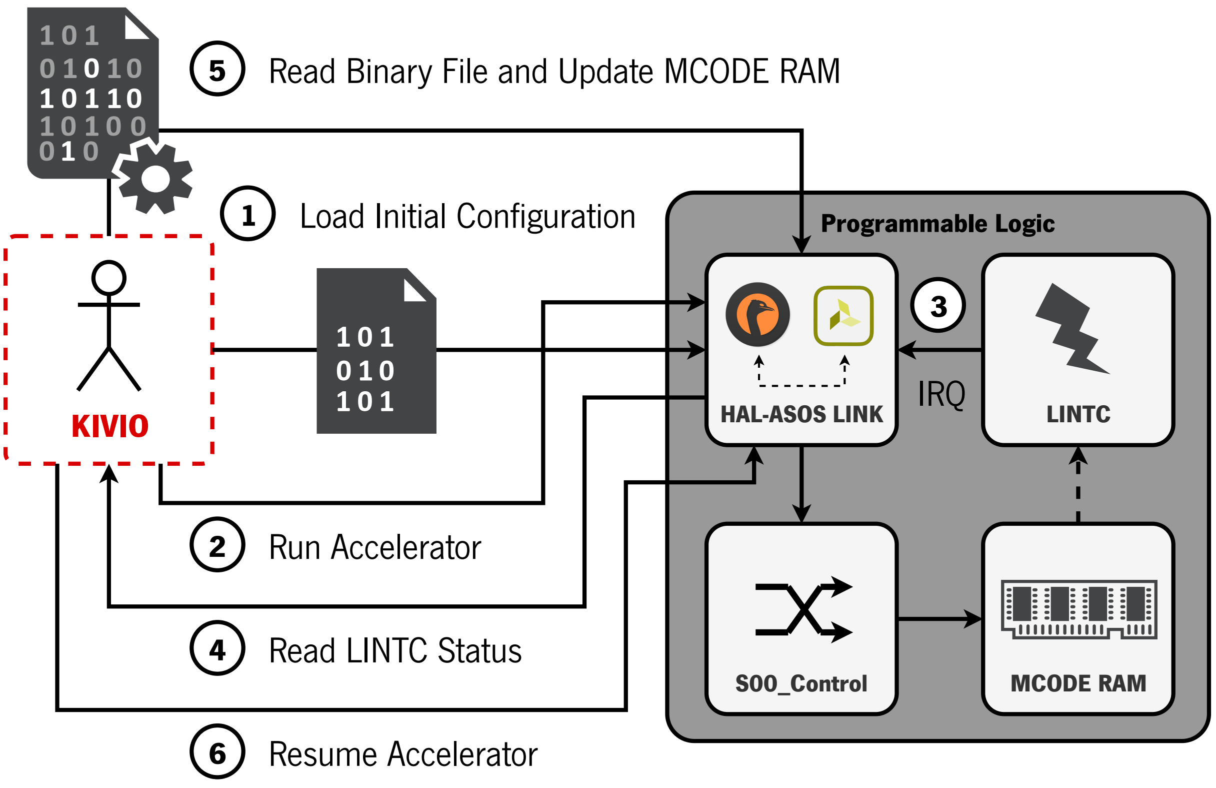


Figure 6.2.1: Microcode Fault Injection Test Overview, block design in fig. B.6.

the HAL-ASOS Link IP and activates KIVIO's interrupt handler. Concerning this, in step ④, already in the handler's bottom-half, the kernel's status register is read and the analyzed. This originates KIVIO to perform a microcode update by seeing the status register's bit pertaining to the update asserted.

This update, depicted in ⑤, contains the mapping of all low-level system calls needed for the correct operation of the accelerator. To avoid side-channel attacks that exploit the microcode update's payload, one opted to use the same method used in Intel's x86 microcode updates, i.e., make the microcode update words bigger than the actual size of the microprogram's memory microinstructions and randomize the rest of the payload bits that do not carry the actual data. An overview of this scheme can be seen in fig. 6.2.2. Thus, the higher sixteen bits (from 31 to 16) of the payload are random for every word written to the microcode RAM, and the lower sixteen bits (from 15 to 0) contain the actual microinstruction. All that is left for the microcode update to take effect is restarting the accelerator, in ⑥, and let the hardware task run to completion. Note that, in the Intel's case, encryption is also used in addition to the random noise added within 2048 bytes of patch data (presupposing a 48-byte header).

The fault injection scheme was tested within a full simulation environment, as aforementioned, and fig. 6.2.4 represents the results. At the beginning of the simulation one can see the load of the initial configurations regarding the masking of interrupts and the set-up of the interface registers, followed by the initial microcode memory configuration with all system calls unmapped and with faults (at about 13.00 microseconds). Then one can observe the hardware task starting, leaving the *s0_ready* state, and going to *s1_sub_data_pipe*. In this state, a fault is detected and an IRQ is issued to the host system, at about 59.00 microseconds. Accordingly, the latter perceives this, and issues a microcode update with the appropriate system call binaries, at about 76.00 microseconds. Then the hardware task resumes the processing as intended, proving that it can handle reconfiguration of its microcode mechanisms at runtime.

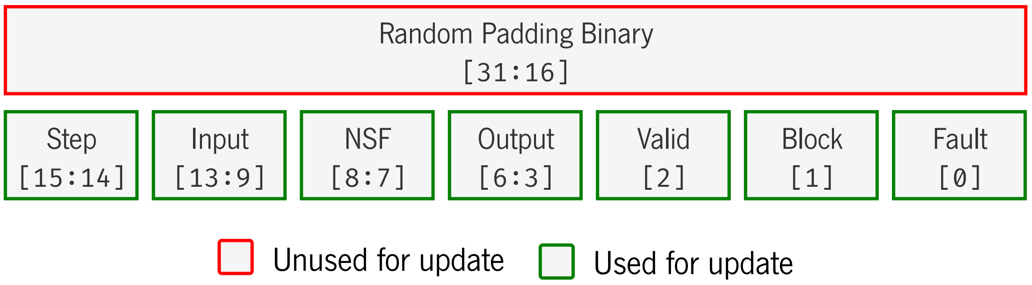
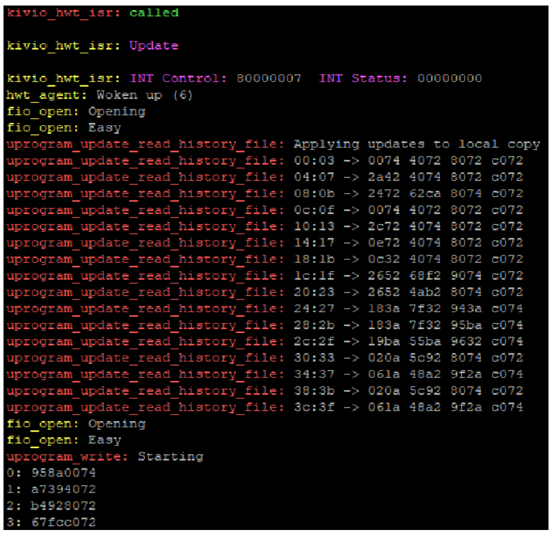


Figure 6.2.2: Microcode Fault Injection Payload Overview, following Intel's x86 example.

After deployment on the target board, i.e., Zybo Z7-10, the microprogram fault injection was also tested, and the system proved to recover from a faulty state by issuing a PL-to-PS IRQ, which led to the update of the microprogram. This can be seen in fig. 6.2.3a, alongside the target board in fig. 6.2.3b.

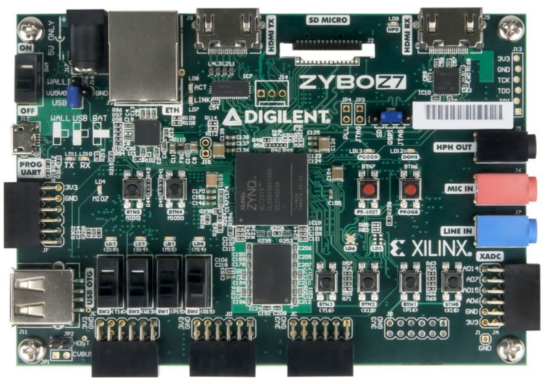


```
kivio_hwt_isr: called

kivio_hwt_isr: Update

kivio_hwt_isr: INT Control: 80000007  INT Status: 00000000
hwt_agent: Woken up (6)
fio_open: Opening
fio_open: Easy
uprogram_update_read_history_file: Applying updates to local copy
uprogram_update_read_history_file: 00:03 -> 0074 4072 8072 c072
uprogram_update_read_history_file: 04:07 -> 2a42 4074 8072 c072
uprogram_update_read_history_file: 08:0b -> 2472 62ca 8074 c072
uprogram_update_read_history_file: 0c:0f -> 0074 4072 8072 c072
uprogram_update_read_history_file: 10:13 -> 2c72 4074 8072 c072
uprogram_update_read_history_file: 14:17 -> 0e72 4074 8072 c072
uprogram_update_read_history_file: 18:1b -> 0c32 4074 8072 c072
uprogram_update_read_history_file: 1c:1f -> 2652 68f2 9074 c072
uprogram_update_read_history_file: 20:23 -> 2652 4ab2 8074 c072
uprogram_update_read_history_file: 24:27 -> 183a 7f32 943a c074
uprogram_update_read_history_file: 28:2b -> 183a 7f32 95ba c074
uprogram_update_read_history_file: 2c:2f -> 19ba 55ba 9632 c074
uprogram_update_read_history_file: 30:33 -> 020a 5c92 8074 c072
uprogram_update_read_history_file: 34:37 -> 061a 48a2 9f2a c074
uprogram_update_read_history_file: 38:3b -> 020a 5c92 8074 c072
uprogram_update_read_history_file: 3c:3f -> 061a 48a2 9f2a c074
fio_open: Opening
fio_open: Easy
uprogram_write: Starting
0: 958a0074
1: a7394072
2: b4928072
3: 67fcc072
```

(a) Deployment: Microprogram Fault Injection.



(b) Deployment Board: Zybo Z7-10 [13].

Figure 6.2.3: Update Mechanism Deployment on the Zybo Z7-10 board.

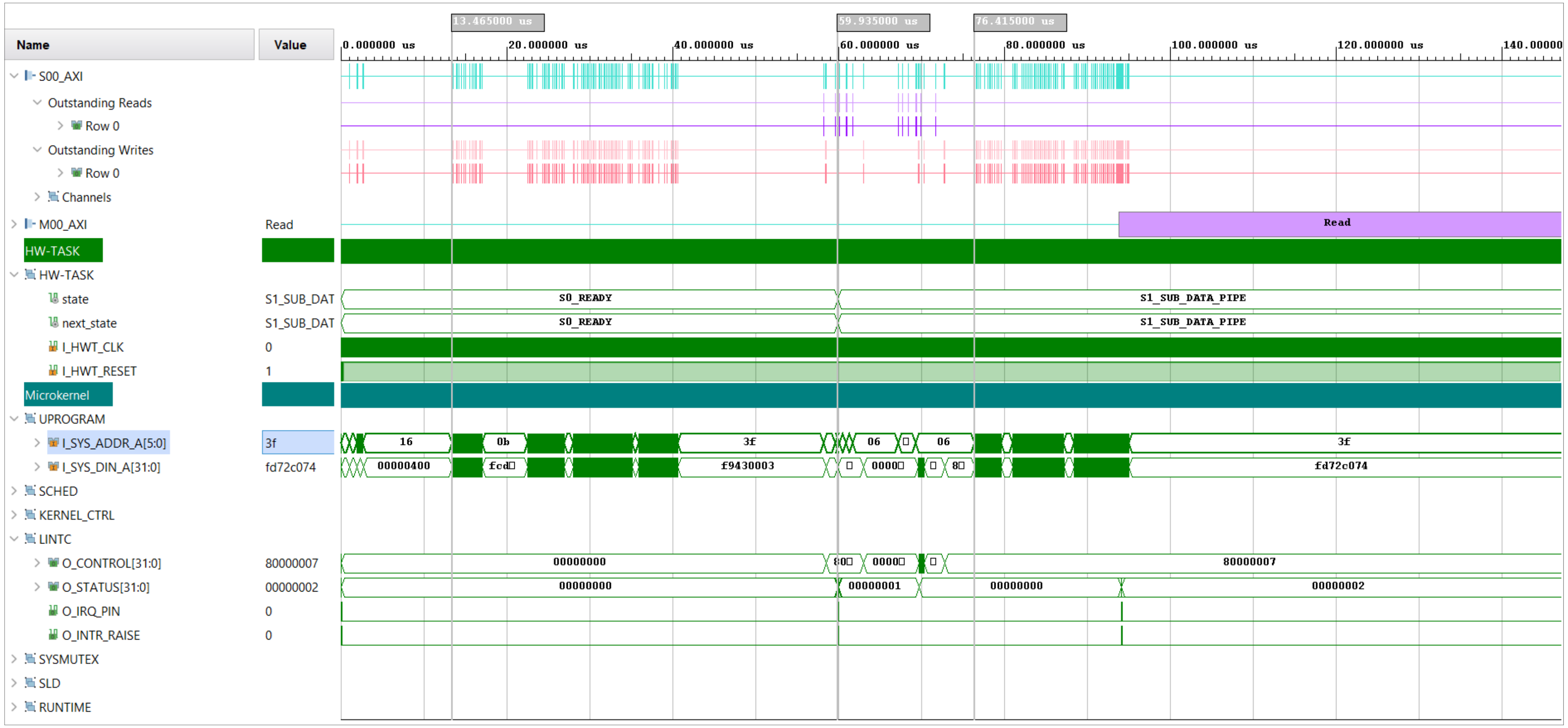


Figure 6.2.4: Microcode Fault Injection: Full Simulation with HAL-ASOS Link IP.

7. Conclusion

The technology improvements of the last few decades allowed for higher-density computing systems to emerge, as their size kept getting smaller. Even though, technology has limitations and systems could not keep getting smaller and denser. Meeting performance goals shifted from increasing the clock rates to using multiple processing elements. Soon after, the increase of processing elements in a system hit a pitfall and this approach could not keep up with the demand for even higher performance. For this, new heterogeneous systems based on different processing elements started to appear, presenting significant results relating to performance improvements. Nonetheless, the use of heterogeneous systems created boundaries that could hinder the design and deployment process of this type of systems, especially when one is designing for systems that lack commonalities on both ends, e.g., CPU+FPGA. To resolve the issues, multiple heterogeneous architectures and programming models surfaced to reduce the overall design effort of hybrid applications. One of which is the Hardware Assisted Linux for Application Specific Operating Systems (HAL-ASOS) accelerator model.

Although the literature has introduced multiple implementations of hardware-assisted operating systems, by offloading certain OS portions, or by creating hardware representations of threads, no one, besides HAL-ASOS, uses microcode techniques to increase design elasticity, extensibility, and regularity. The transposition of CPU-related concepts like microcode to hardware allowed for the creation of CPU-like hardware cores, capable of running hardware threads, and fomented its integration with existing open-source OSes like Linux by adding kernel- and user-level abstractions.

This thesis proposes an accelerator architecture based on HAL-ASOS, built from the ground up. The accelerator applies horizontal microcode to leverage low-level system calls and vertical microcode to leverage high-level system calls. Additionally, following HAL-ASOS' stipulations, the accelerator model is divided into a microkernel region – representing the kernel space – and a hardware task region – representing the user space. To support both system call types the microkernel also grants specific services, including event, memory and buffer management, resource management, and synchronization mechanisms.

The research aim was to develop the accelerator from scratch to explore the openly-stated issues of HAL-ASOS, present solutions, and evaluate their results. Concerning this, three main limitations were explored: (1) the accelerator interface for extended features – that allowed for the hardware task to call Linux software system calls that could not be performed in hardware, e.g., for file I/O – (2) the scalability of procedures in the hardware task – that was a system-resource bottleneck regarding target deployment – and (3) the accelerator's means to cope with internal or Linux-kernel changes, and update its microcode structures – which was left at an early research state. With this purpose, the accelerator extensions introduced a command-exchange framework through shared memory, for the processing of extended features, runtime management capabilities to reduce the amount of combinational logic of the hardware task, and refactored the accelerator to update the microcode.

By deploying a software application portion to hardware using the proposed accelerator model, one is capable of issuing commands through shared memory to perform extended system calls, and perform microcode updates protected using a random payload scheme, as tested with Xilinx's VIP (simulation), HAL-ASOS' Link IP (full simulation), and ultimately on the ZYBO Z7-10 board (deployment). Moreover, the scalability was improved by always achieving a resource usage decrease of at least one type of combinational element, with a lot less cell dispersion. Furthermore, regularity is also provided when increasing the amount of hardware task procedures, for the same encryption operation (*XOR*), and when using other encryption operations (AES). This proved to be accomplished with the trade-off of more logic regarding other elements that were not so scarce.

7.1 Future Works

The work in the dissertation addresses some of the HAL-ASOS limitations, by proposing a new and specific hardware accelerator architecture. Despite, since the subject involves a lot of concepts there is still room for the accelerator model to evolve and expand to various implementation paths, as the following:

Microcode Update Authentication. The current version of the microprogrammable units of the dissertation allow for updates through the S00 Control AXI4 lite interface but if there is kernel tampering in the KIVIO framework one has access to the microcode update mechanism. This could lead to meltdown-like attacks that rely on rouge data load to the microcode RAM under the form of a microcode *Trojans*. This could be avoided with a microcode authentication module that prevents malicious updates. The latter could be also improved with the introduction of a microcode update header containing a safe key, and an update signature, containing, for example, a checksum. Additionally, encryption could be used alongside the random payload method used in the dissertation to protect the overall system against side-channel attacks that exploit the microcode by analyzing the update's payload structure.

Procedure Datapath Reconfiguration. By exposing the procedure scheduler of the dissertation to the host system it is possible to provide scheduler updates that dictate the flow of low-level syscalls. Changing the order of the system calls to be executed creates discrepancies between the microcoded scheduler and its datapath. This occurs since it is introduced a new point of failure related with incorrect parameters and associated with having a datapath that is static. Improvements could be added to provide a mechanism that can use dynamic datapaths with DPR, or implement an arbiter that establishes the active datapath parameters according to the active user-level system call, similar to the SLD but for user-level procedures.

Scatter-Gather Memory Mechanism. Currently, the KIVIO framework allocates a block of contiguous memory dedicated to the accelerator as it happened with HAL-ASOS. This might start to generate problems when the allocation starts requiring contiguous memory blocks of bigger size. To mitigate this, future research could investigate the potentialities of implementing memory allocation in multiple blocks of various sizes like in GPUs. This would require a module in the microkernel that could handle those memory blocks as contiguous memory through a scatter-gather mechanism, for example, when one wants to make burst transfers to the main memory.

Syscall Caching and Argument Checking. At the moment, the microkernel does not have argument checking, so calling system calls with incorrect parameters could happen if the designer does not pay attention to particular system specifications. This could be avoided with the implementation of a syscall cache that could save system calls that are used multiple times with the same arguments. After verifying the parameters are safe, the system call and parameters would be stored in cache to be called later if the need arose. This approach could also improve certain cases where a user-level system call could immediately know its returns without interfacing with the host, e.g., if there is an already opened file and the system call happened to be called again, then the hardware task would already know the descriptor from the previous procedure run. The research could take the example of caching presented in [64].

Hardware Task Lockstep. Presently the hardware task does not have protecting against radiation effects that produce hardware faults, like SEUs. This could be mitigated with the use of lockstep in the hardware task, recurring to Thread Shadowing. This would not only mitigate this kind of faults but could also be a way to add another form of parameter checking and also memory access checks, by having a hardware task monitoring another and performing result comparison, like in [40]. This would, ultimately, add fault tolerance capabilities to the system.

Multitask Arbiter. The hardware task of the dissertation did not explore the concept of multitasking, which means that there is only one kernel controlling one hardware task at any time, except when evaluating resource usage for a dual-task scheme. Contrarily, HAL-ASOS goes beyond the standalone task execution and explores the multitask execution with an inter-task communication mechanism to pass information between tasks. Either way, even in the scenario of HAL-ASOS, the kernel cannot handle requests for multiple control units from more than one hardware task. This means that for every task there is an associated kernel, which might hinder the platform targeting if the final board does not have a lot of resources. This could be solved with the introduction of an arbiter that would decide the active hardware task by employing preemptive scheduling algorithms, e.g., round-robin or static priority.

Evaluate Standalone vs. Multiple Kernel Execution The introduction of a multitask arbiter would allow for the introduction of standalone synchronous multitask execution, i.e., single kernel execution with multiple synchronous tasks (hardware threads). This would open up the possibility of comparing this type of execution with the ones already explored by HAL-ASOS, *c.f* [62]. Specifically, regarding standalone kernel execution versus multiple kernel execution in their synchronous and asynchronous variants. One thing that also would be worth exploring is the comparison of microcode updates under different kernel execution architectures. For example, in the case of multi-kernel/multitask, one would probably need to update the microcode individually for each kernel of the accelerator, similarly to microprogramming in symmetric multiprocessing architectures [11]. Contrarily, the standalone single-task or the standalone multitask make use of a mechanism like the one in the dissertation since there is only one kernel in the accelerator model.

Increase Main Memory Word Burst Size. The microkernel is constrained to main memory burst operations of 256 words. A mechanism that handles bursts above the limit could be implemented by dividing a big burst, e.g., 1024 words, into smaller bursts, e.g., four bursts of 256 words, like HAL-ASOS proposes.

Asynchronous Extended Features. The current implementation of hardware task extended features presupposes waiting in polling for the extended command to be accepted by KIVIO and after that continue to wait in polling for the return of the system call, e.g., the descriptor of a file, socket, or pipe, when considering a syscall that subscribes to a certain form of I/O. This could be improved with the implementation of asynchronous system calls that only wait for the command to be accepted and later return to the KFIFO at a more convenient time to retrieve the return, in an asynchronous manner. This would utilize the KFIFO to its full potential by having multiple command requests in the argument section at a time, and also multiple returns in the return section pertaining to multiple user-level system calls.

Merge the Microcode Adaptation Mechanisms. The system calls of the microkernel are implemented with a vertical microcode entity (scheduler) controlling a horizontal microcode entity (microprogram), for the execution of user-level procedures, whilst using the runtime manager entity to assure scalability. Despite, this is a complex scheme that is not easily analyzed and debugged. To resolve this issues one could merge the three units into one. This would result into a single microprogram with a high memory region dedicated to user syscalls and a low memory region for kernel syscalls. When executing user syscalls, one would then jump from high memory to low memory for the execution of kernel microinstructions. The current scheme already implicitly does something like this but does not unify the address space of the scheduler memory with the address space of the microprogram memory. The overall setup could result in single-RAM execution (as presented in the dissertation but merging the microcode entity address spaces) or ROM-RAM execution, but also still trying to reuse cases for the scalability issue.

Improve Microcode Error Handling. The microcode error handling mechanism only copes with diversity by issuing interrupt requests to the host pertaining to updates. The microcode is then updated, and the system proceeds with its execution. This is a good initial step but a more refined error handling mechanism could be implemented. Regarding this, the microprogram could become aware of errors at an higher level, i.e., it could perceive that a certain order of user-level procedures would produce errors, e.g., executing an operation based on a descriptor of a object that ceased to exist in the host. This would relate to the mitigation of errors like use-after-free, and would also resort to some sort of caching to store the correct order of commands to send to the host to resolve the problem and generate, for example, another object for the same purpose and resume execution in the point where the error initially occurred.

Explore Microprogram ROM-RAM Execution Model. While the merging of the microcode units would be worth exploring, the alteration of the microprogram storage model to ROM-RAM execution would be interesting as well. This approach presupposes maintaining the scheduler and runtime manager unchanged and introducing an additional ROM onto the RAM-based microprogram. At runtime, depending on the most significant bit of the system call ID, one would change between non-volatile memory and the microprogram's RAM to achieve internal error handling in critical scenarios. A PoC would resort to fault injecting rouge data to the RAM, like the case study explored in this dissertation, and see if the microprogram could change between memories to resolve the problem. This approach would also need to explore some sort of bootloader or bootload instruction to process the RAM updates while executing from ROM. Additionally, since the writing of the microcode RAM is processed through the S00 Control interface (dissertation), or possibly through the S01 Data interface (with the LRAM acting as cache and using the aforementioned bootloader in this scheme), the updates are restricted to word-rated transactions, i.e, one cannot perform burst updates. Another topic worth exploring would be introducing burst updates into the system by adding some sort of ZCU unit responsible for transferring data from the local memory (acting as an instruction cache) to the microprogram RAM. As whole, this approach was not explored since it increased the scheduler's memory footprint by $\approx 100\%$ and originated a complex low-level syscall branching scheme. The design of this microprogram architecture would be similar to fig. B.3.

Add Preferred Direction Semantics. Right now every composite procedure has a granularity of sixteen scheduler microinstructions, which means that the depth of branches does not justify the addition of a branch predictor to mitigate the branch-not-taken execution penalties, i.e., the penalties for a not taken branch are not that high since the system is not pipelined. Even though, it would be good to add preferred direction semantics associated with branches. With this, one would additionally have another microintruction field to indicate the likelihood of a branch to be taken, with keywords like *likely_taken* or *unlikely_taken*. This could already be applied to the argument KFIFO full check since, for the polling case, it never becomes full of commands.

References

Book Sources

[15] Elena Dubrova (2013). Fault-Tolerant Design. Springer Publishing Company, Incorporated. ISBN: 1461421128.

[21] Ted Huffmire et al. (2010). Handbook of FPGA Design Security. 1st ed. Springer.

[27] Steve Kilts (2007). Advanced FPGA Design: Architecture, Implementation, and Optimization. Wiley-IEEE Press. ISBN: 0470054379.

[35] M. Morris Mano (2016). Digital Logic and Computer Design. Pearson Education India.

[37] Clive Maxfield (2008). FPGAs: Instant Access. 1st ed. Newnes.

[38] — (2009). FPGAs: World Class Designs. 1st ed. Elsevier.

[43] Bryon Moyer (2013). Real World Multicore Embedded Systems. Newnes.

[47] Robert Oshana and Mark Kraeling (2020). Software Engineering for Embedded Systems. 2nd ed. Elsevier.

[48] David Patterson and John Hennessy (2017). Computer Architecture: A Quantitative Approach. 6th ed. Morgan Kaufmann.

[58] C.H. Roth and L.K. John (2007). Digital Systems Design Using VHDL. Cengage Learning. ISBN: 9780534384623.

[60] Daniel Siewiorek, Gordon Bell, and Allen Newell (1982). Computer Structures: Principles and Examples. McGraw-Hill, Inc.

[67] William Stallings (2010). Computer Organization and Architecture: Designing for Performance. Prentice Hall.

Proceedings Sources

[2] Nils Albartus et al. (Aug. 2021). “On the Design and Misuse of Microcoded (Embedded) Processors — A Cautionary Note”. In: 30th USENIX Security Symposium (USENIX Security 21). USENIX Association, pp. 267–284. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/albartus.

[3] David Andrews et al. (2005). “hthreads: a hardware/software co-designed multithreaded RTOS kernel”. In: 2005 IEEE Conference on Emerging Technologies and Factory Automation. Vol. 2, 8 pp.–338. DOI: 10.1109/ETFA.2005.1612697.

[6] Melanie Berg (2011). “New Developments in Field Programmable Gate Array (FPGA) Single Event Upsets (SEUs) and Fail-Safe Strategies”. In: Revolutionary Electronics in Space (ReSpace) / Military and Aerospace Programmable Logic Devices (MAPLD) 2011 Conference.

[7] Melanie Berg, Kenneth LaBel, and Jonathan Pellish (2015). “New Developments in FPGA: SEUs and Fail-Safe Strategies from the NASA Goddard Perspective”. In: SERRESSA 2015: on the Effects of Radiation on Embedded Systems for Space Applications.

[9] Robert Brodersen, Artem Tkachenko, and Hayden Kwok-Hay So (2006). “A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH”. In: Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS ’06), pp. 259–264. DOI: 10.1145/1176254.1176316.

[18] Tiago Gomes et al. (Nov. 2016). “Hybrid real-time operating systems: deployment of critical FreeRTOS features on FPGA”. In: vol. 8. Inderscience Enterprises, pp. 483–492.

[23] Aws Ismail and Lesley Shannon (2011). “FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators”. In: 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines, pp. 170–177. DOI: 10.1109/FCCM.2011.48.

[25] Matthew Jacobsen and Ryan Kastner (2013). “RIFFA 2.0: A reusable integration framework for FPGA accelerators”. In: 2013 23rd International Conference on Field programmable Logic and Applications, pp. 1–8. DOI: 10.1109/FPL.2013.6645504.

[28] Benjamin Kollenda et al. (2018). “An Exploratory Analysis of Microcode as a Building Block for System Defenses”. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. CCS ’18. Toronto, Canada: Association for Computing Machinery, pp. 1649–1666. ISBN: 9781450356930. DOI: 10.1145/3243734.3243861.

[29] Philipp Koppe et al. (Aug. 2017). "Reverse Engineering x86 Processor Microcode". In: 26th USENIX Security Symposium (USENIX Security 17). Vancouver, BC: USENIX Association, pp. 1163–1180. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/koppe.

[31] Ziyi Liu et al. (2014). "Programmable decoder and shadow threads: Tolerate remote code injection exploits with diversified redundancy". In: 2014 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1–6. DOI: 10.7873/DATE.2014.064.

[32] Enno Lübbers and Marco Platzner (2008). "A portable abstraction layer for hardware threads". In: 2008 International Conference on Field Programmable Logic and Applications, pp. 17–22. DOI: 10.1109/FPL.2008.4629901.

[33] — (2007a). ReconOS: An RTOS Supporting Hardware and Software Threads. URL: http://www.reconos.de/publications/luebbers07_fpl_slides.pdf.

[34] — (2007b). "ReconOS: An RTOS Supporting Hardware and Software Threads". In: 2007 International Conference on Field Programmable Logic and Applications. IEEE, pp. 441–446. DOI: 10.1109/FPL.2007.4380686.

[39] Sebastian Meisner and Marco Platzner (2015). "Comparison of thread signatures for error detection in hybrid multi-cores". In: 2015 International Conference on Field Programmable Technology (FPT), pp. 212–215. DOI: 10.1109/FPT.2015.7393153.

[40] — (2016). "Thread shadowing: On the effectiveness of error detection at the hardware thread level". In: 2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig), pp. 1–8. DOI: 10.1109/ReConFig.2016.7857193.

[41] — (2014). "Thread Shadowing: Using Dynamic Redundancy on Hybrid Multi-cores for Error Detection". In: Reconfigurable Computing: Architectures, Tools, and Applications. Ed. by Diana Goehringer et al. Cham: Springer International Publishing, pp. 283–290. ISBN: 978-3-319-05960-0.

[45] Kevin Nam, Blair Fort, and Stephen Brown (2017). "FISH: Linux system calls for FPGA accelerators". In: 2017 27th International Conference on Field Programmable Logic and Applications (FPL), pp. 1–4. DOI: 10.23919/FPL.2017.8056785.

[46] Soon Ee Ong et al. (2013). "SEOS: Hardware Implementation of Real-Time Operating System for Adaptability". In: 2013 First International Symposium on Computing and Networking, pp. 612–616. DOI: 10.1109/CANDAR.2013.110.

[49] Wesley Peck et al. (2006). “Hthreads: A Computational Model for Reconfigurable Devices”. In: 2006 International Conference on Field Programmable Logic and Applications, pp. 1–4. DOI: 10.1109/FPL.2006.311336.

[50] Jorge Pereira et al. (2014). “Co-Designed FreeRTOS Deployed on FPGA”. In: 2014 Brazilian Symposium on Computing Systems Engineering, pp. 121–125. DOI: 10.1109/SBESC.2014.11.

[52] César Polo (2017). “SEE Single Event Effects: Radiation Environment and its Effects in EEE Components and Hardness Assurance for Space Applications”. In: URL: https://indico.cern.ch/event/635099/contributions/2570672/attachments/1456364/2249943/Single_Event_Effecs_Radiation_Course_May_2017_SEE_CBP.pdf.

[53] Vaughan Pratt (1995). “Anatomy of the Pentium bug”. In: TAPSOFT ’95: Theory and Practice of Software Development. Ed. by Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 97–107. ISBN: 978-3-540-49233-7.

[55] Piyumal Ranawaka et al. (2019a). “Application Specific Architecture for Hardware Accelerating HOG-SVM to Achieve High Throughput on HD Frames”. In: 2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP). Vol. 2160-052X, pp. 131–134. DOI: 10.1109/ASAP.2019.00-18.

[59] Rasool Sharifi and Ashish Venkat (2020). “CHEx86: Context-Sensitive Enforcement of Memory Safety via Microcode-Enabled Capabilities”. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pp. 762–775. DOI: 10.1109/ISCA45697.2020.00068.

[61] V. Silva et al. (2016). “Linux- and FPGA-based accelerated single-phase shunt active power filter”. In: IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society, pp. 4802–4807. DOI: 10.1109/IECON.2016.7793875.

[64] Dimitrios Skarlatos et al. (2020). “Draco: Architectural and Operating System Support for System Call Security”. In: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 42–57. DOI: 10.1109/MICRO50266.2020.00017.

[68] L.T. Su (Feb. 2013). ““Architecting the future through heterogeneous computing””. In: pp. 8–11. ISBN: 978-1-4673-4515-6. DOI: 10.1109/ISSCC.2013.6487618.

[70] Charalampos Vatsolakis and Dionisios Pnevmatikatos (2017). “RACOS: Transparent access and virtualization of reconfigurable hardware accelerators”. In: 2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), pp. 11–19. DOI: 10.1109/SAMOS.2017.8344606.

Article Sources

[1] Andreas Agne et al. (2014). “ReconOS: An Operating System Approach for Reconfigurable Computing”. In: IEEE Micro 34.1, pp. 60–71. DOI: 10.1109/MM.2013.110.

[4] David Andrews et al. (2004). “Programming models for hybrid FPGA-cpu computational components: a missing link”. In: IEEE Micro 24.4, pp. 42–53. DOI: 10.1109/MM.2004.36.

[5] A. Avizienis et al. (2004). “Basic concepts and taxonomy of dependable and secure computing”. In: IEEE Transactions on Dependable and Secure Computing 1.1, pp. 11–33. DOI: 10.1109/TDSC.2004.2.

[8] S. Borkar (2005). “Designing reliable systems from unreliable components: the challenges of transistor variability and degradation”. In: IEEE Micro 25.6, pp. 10–16. DOI: 10.1109/MM.2005.110.

[12] Victor Costan and Srinivas Devadas (2016). “Intel SGX Explained”. In: IACR Cryptol. ePrint Arch. 2016.86, pp. 1–118.

[17] David Gifford and Alfred Spector (Apr. 1987). “Case Study: IBM’s System/360-370 Architecture”. In: Commun. ACM 30.4, pp. 291–307. ISSN: 0001-0782. DOI: 10.1145/32232.32233.

[19] Andreas Herkersdorf et al. (2014). “Resilience Articulation Point (RAP): Cross-layer dependability modeling for nanometer system-on-chip resilience”. In: Microelectronics Reliability 54.6, pp. 1066–1074. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2013.12.012.

[20] Mark D. Hill et al. (2019). “On the Spectre and Meltdown Processor Security Vulnerabilities”. In: IEEE Micro 39.2, pp. 9–19. DOI: 10.1109/MM.2019.2897677.

[24] Xabier Iturbe et al. (2013). “R3TOS: A Novel Reliable Reconfigurable Real-Time Operating System for Highly Adaptive, Efficient, and Dependable Computing on FPGAs”. In: IEEE Transactions on Computers 62.8, pp. 1542–1556. DOI: 10.1109/TC.2013.79.

[30] Chit-Kwan Lin et al. (2018). “Programming Spiking Neural Networks on Intel’s Loihi”. In: Computer 51.3, pp. 52–61. DOI: 10.1109/MC.2018.157113521.

[36] Ivo Marques et al. (2021). “Lock-V: A heterogeneous fault tolerance architecture based on Arm and RISC-V”. In: Microelectronics Reliability 120, p. 114120. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2021.114120. URL: https://www.sciencedirect.com/science/article/pii/S002627142100086X.

[51] Hung-Manh Pham, Ludovic Devaux, and Sébastien Pillement (June 2011). “Re2DA: Reliable and Reconfigurable Dynamic Architectures”. In: DOI: 10.1109/ReCoSoC.2011.5981519.

[56] Piyumal Ranawaka et al. (Dec. 2019b). "High Performance Application Specific Stream Architecture for Hardware Acceleration of HOG-SVM on FPGA". In: IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E102.A, pp. 1792–1803. DOI: 10.1587/transfun.E102.A.1792.

[57] David Ratter (2004). "FPGAs on mars". In: Xcell J 50, pp. 8–11.

[63] Vitor Silva et al. (2021). "HAL-ASOS Accelerator Model: Evolutive Elasticity by Design". In: Electronics 10.17. ISSN: 2079-9292. DOI: 10.3390/electronics10172078. URL: https://www.mdpi.com/2079-9292/10/17/2078.

[66] Hayden Kwok-Hay So and Robert Brodersen (2008). "A Unified Hardware/Software Runtime Environment for FPGA-Based Reconfigurable Computers Using BORPH". In: 7.2. ISSN: 1539-9087. DOI: 10.1145/1331331.1331338.

[69] Anuj Vaishnav et al. (Sept. 2020). "FOS: A Modular FPGA Operating System for Dynamic Workloads". In: ACM Trans. Reconfigurable Technol. Syst. 13.4. ISSN: 1936-7406. DOI: 10.1145/3405794.

[71] Hoang-Gia Vu, Takashi Nakada, and Yasuhiko Nakashima (2021). "Efficient hardware task migration for heterogeneous FPGA computing using HDL-based checkpointing". In: Integration 77, pp. 180–192. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2020.11.011. URL: https://www.sciencedirect.com/science/article/pii/S0167926020302984.

[72] Ying Wang et al. (2013). "SPREAD: A Streaming-Based Partially Reconfigurable Architecture and Programming Model". In: IEEE Transactions on Very Large Scale Integration (VLSI) Systems 21.12, pp. 2179–2192. DOI: 10.1109/TVLSI.2012.2231101.

[73] Maurice Wilkes (1981). "The Best Way to Design an Automatic Calculating Machine". In: Microprocessing and Microprogramming 8.3, pp. 141–144. ISSN: 0165-6074. DOI: https://doi.org/10.1016/0165-6074(81)90018-1.

[74] Zhiguo Yang et al. (2020). "Reverse Engineering of Intel Microcode Update Structure". In: IEEE Access 8, pp. 169676–169687. DOI: 10.1109/ACCESS.2020.3024243.

[75] Zongwei Zhu et al. (2019). "A Hardware and Software Task-Scheduling Framework Based on CPU+FPGA Heterogeneous Architecture in Edge Computing". In: IEEE Access 7, pp. 148975–148988. DOI: 10.1109/ACCESS.2019.2943179.

Academic Sources

[10] Anselm Busse (Dec. 2016). “A Dynamic and Component-Based Process Scheduler Framework for Heterogeneous Many-Core Systems”. Ph.D. Dissertation, Technischen Universität Berlin.

[11] Daming Dominic Chen and Gail-Joon Ahn (Dec. 2014). “Security Analysis of x86 Processor Microcode”. URL: https://www.dcddcc.com/docs/2014_paper_microcode.pdf.

[14] Björn Döbel (Nov. 2014). “Operating System Support for Redundant Multithreading”. Ph.D. Dissertation, Technischen Universität Dresden.

[26] Boris Kettelhoit (Sept. 2009). “Architektur und Entwurf Dynamisch Rekonfigurierbarer FPGA-Systeme”. Ph.D. Dissertation, University of Paderborn.

[42] José Mendes (Jan. 2023). “Handling Linux and HAL-ASOS Model Discrepancies: a Microcode Approach”. Masters Dissertation, Universidade do Minho.

[44] Onur Mutlu (Sept. 2020). Computer Architecture Lecture 1: Introduction and Basics. URL: https://safari.ethz.ch/architecture/fall2020/lib/exe/fetch.php?media=onur-comparch-fall2020-lecture1-intro-afterlecture.pdf.

[54] Ioan Raicu et al. (Jan. 2007). “Harnessing grid resources with data-centric task farms”. University of Chicago.

[62] Vitor Silva (June 2022). “HAL-ASOS – Hardware Assisted Linux for Application Specific Operating Systems”. Ph.D. Dissertation, Universidade do Minho.

[65] Mark Smotherman (2009). “A Brief History of Microprogramming”. School of Computing, Clemson University, Clemson, SC, USA, Tech. Rep.

Other Sources

[13] Diligent (Jan. 2023). Zybo Z7: Zynq-7000 ARM/FPGA SoC Development Board. URL: https://digilent.com/reference/programmable-logic/zybo-z7/start.

[16] GCC (Oct. 2021). GCC Online Documentation: GNU Project. URL: https://gcc.gnu.org/onlinedocs/.

[22] Intel (Nov. 2022). Alder Lake S: Overview and Technical Documentation. URL: https://www.intel.com/content/www/us/en/products/platforms/details/alder-lake-s.html.

A. Source Listings

Listing A.1: Configuration Package: Microprogram and System Call Constants and Definitions (HDL); back to listing 4.5.

```
---------------------------------------------------------------------------------------
-- UPROGRAM INPUT DECODER CONSTANTS
---------------------------------------------------------------------------------------
CONSTANT C_AUX_TEST_FALSE_OFFSET        : NATURAL := 0;
CONSTANT C_ARREADY_OFFSET               : NATURAL := 1;
CONSTANT C_RVALID_OFFSET                : NATURAL := 2;
CONSTANT C_AWREADY_OFFSET               : NATURAL := 3;
CONSTANT C_WREADY_WLAST_OFFSET          : NATURAL := 4;
CONSTANT C_LBUS_WR_ACK_B_WLAST_OFFSET   : NATURAL := 5;
CONSTANT C_LBUS_WR_ACK_B_OFFSET         : NATURAL := 6;
CONSTANT C_LBUS_RD_ACK_B_OFFSET         : NATURAL := 7;
CONSTANT C_LBUS_RLAST_OFFSET            : NATURAL := 8;
CONSTANT C_LOCKED_A_OFFSET              : NATURAL := 9;
CONSTANT C_LOCKED_B_OFFSET              : NATURAL := 10;
CONSTANT C_FREE_OFFSET                  : NATURAL := 11;
CONSTANT C_NOT_LOCKED_A_OFFSET          : NATURAL := 12;
CONSTANT C_NOT_LOCKED_B_OFFSET          : NATURAL := 13;
CONSTANT C_RVALID_RLAST_OFFSET          : NATURAL := 14;
CONSTANT C_BVALID_TIMEOUT_OFFSET        : NATURAL := 15;
CONSTANT C_LBUS_WLAST_OFFSET            : NATURAL := 16;
CONSTANT C_EVENT_ELAPSED_OFFSET         : NATURAL := 17;
CONSTANT C_EV_MANAGER_READY_OFFSET      : NATURAL := 18;
CONSTANT C_ADDR_MANAGER_READY_OFFSET    : NATURAL := 19;
CONSTANT C_LBUS_RD_ACK_B_RLAST_OFFSET   : NATURAL := 20;
CONSTANT C_ACC_DEAD_OFFSET              : NATURAL := 21;
CONSTANT C_INTR_RAISE_OFFSET            : NATURAL := 22;
--- ...                                            ...
CONSTANT C_AUX_TEST_TRUE_OFFSET         : NATURAL := 31;
---------------------------------------------------------------------------------------

---------------------------------------------------------------------------------------
-- UPROGRAM OUTPUT ENCODER CONSTANTS
---------------------------------------------------------------------------------------
CONSTANT C_ARVALID_OFFSET               : NATURAL := 1;
CONSTANT C_RREADY_OFFSET                : NATURAL := 2;
CONSTANT C_AWVALID_OFFSET               : NATURAL := 3;
CONSTANT C_WVALID_OFFSET                : NATURAL := 4;
CONSTANT C_BREADY_M00_TRIGGER_OFFSET    : NATURAL := 5;
CONSTANT C_LBUS_WR_CE_OFFSET            : NATURAL := 6;
CONSTANT C_LBUS_RD_CE_OFFSET            : NATURAL := 7;
--...                                              ...8
CONSTANT C_EV_MANAGER_TRIGGER_OFFSET    : NATURAL := 9;
CONSTANT C_ADDR_MANAGER_TRIGGER_OFFSET  : NATURAL := 10;
--- ...                                            ...
CONSTANT C_NULL_OFFSET                  : NATURAL := 14;
---------------------------------------------------------------------------------------

---------------------------------------------------------------------------------------
-- SYSTEM CALL SPECIFICATION
---------------------------------------------------------------------------------------
type syscall_t is(
---------------------------------------------------------------------------------------
SYSCALL_WORK_NONE,          --00
SYSCALL_WORK_YIELD,         --01
SYSCALL_WAIT_EVENT_TIMEOUT, --02
SYSCALL_LINTC_READ,         --03
SYSCALL_LINTC_WRITE,        --04
SYSCALL_LBUS_READ,          --05
SYSCALL_LBUS_WRITE,         --06
SYSCALL_LBUS_READ_BURST,    --07
SYSCALL_LBUS_WRITE_BURST,   --08
SYSCALL_MUTEX_LOCK,         --09
SYSCALL_MUTEX_TRY_LOCK,     --10
SYSCALL_MUTEX_UNLOCK,       --11
SYSCALL_MBUS_READ,          --12
SYSCALL_MBUS_WRITE,         --13
SYSCALL_MBUS_READ_BURST,    --14
SYSCALL_MBUS_WRITE_BURST);  --15
-----------------------------------------------------------------------------------------
-- SYSTEM CALL ENCODING
-----------------------------------------------------------------------------------------
TYPE syscall_T_ENC IS ARRAY (NATURAL RANGE<>) OF SYSCALL_T;
-----------------------------------------------------------------------------------------
CONSTANT syscall_T_VAL : syscall_T_ENC :=(
SYSCALL_WORK_NONE,          --00
SYSCALL_WORK_YIELD,         --01
SYSCALL_WAIT_EVENT_TIMEOUT, --02
SYSCALL_LINTC_READ,         --03
SYSCALL_LINTC_WRITE,        --04
SYSCALL_LBUS_READ,          --05
SYSCALL_LBUS_WRITE,         --06
SYSCALL_LBUS_READ_BURST,    --07
SYSCALL_LBUS_WRITE_BURST,   --08
SYSCALL_MUTEX_LOCK,         --09
SYSCALL_MUTEX_TRY_LOCK,     --10
SYSCALL_MUTEX_UNLOCK,       --11
SYSCALL_MBUS_READ,          --12
SYSCALL_MBUS_WRITE,         --13
SYSCALL_MBUS_READ_BURST,    --14
SYSCALL_MBUS_WRITE_BURST);  --15
-----------------------------------------------------------------------------------------
CONSTANT C_SYSCALL_LEN : NATURAL := POW2(SYSCALL_T'POS(SYSCALL_T'HIGH)+1); --4
-----------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------
-- SYSTEM CALL FIELD CONSTANTS
-----------------------------------------------------------------------------------------
CONSTANT C_SYSCALL_BASE             : NATURAL := 19;
CONSTANT C_SYSCALL_ERROR_WIDTH      : NATURAL := 5;
CONSTANT C_SYSCALL_INPUT_WIDTH      : NATURAL := 5;
CONSTANT C_SYSCALL_NSF_WIDTH        : NATURAL := 2;
CONSTANT C_SYSCALL_OUTPUT_WIDTH     : NATURAL := 4;
-----------------------------------------------------------------------------------------
-- SYSTEM CALL FIELD DEFINITION
-----------------------------------------------------------------------------------------
SUBTYPE SYSCALL_ERROR  IS std_logic_vector(C_SYSCALL_BASE-1 downto C_SYSCALL_BASE-
    C_SYSCALL_ERROR_WIDTH);  --18:14
SUBTYPE SYSCALL_INPUT  IS std_logic_vector(SYSCALL_ERROR'LOW-1 downto SYSCALL_ERROR'LOW-
    C_SYSCALL_INPUT_WIDTH);  --13:9
SUBTYPE SYSCALL_NSF    IS std_logic_vector(SYSCALL_INPUT'LOW-1 downto SYSCALL_INPUT'LOW-
    C_SYSCALL_NSF_WIDTH);    --8:7
SUBTYPE SYSCALL_OUTPUT IS std_logic_vector(SYSCALL_NSF'LOW-1 downto SYSCALL_NSF'LOW-
    C_SYSCALL_OUTPUT_WIDTH); --6:3
SUBTYPE SYSCALL_MSB    IS std_logic_vector(C_SYSCALL_LEN+SYSCALL_NSF'LENGTH-1 downto
    SYSCALL_NSF'LENGTH);     --5:2
-----------------------------------------------------------------------------------------
CONSTANT C_SYSCALL_VALID_BIT        : NATURAL := SYSCALL_OUTPUT'LOW-1;          --2
CONSTANT C_SYSCALL_BLOCK_TASK_BIT   : NATURAL := C_SYSCALL_VALID_BIT-1;         --1
CONSTANT C_SYSCALL_FAULT_BIT        : NATURAL := C_SYSCALL_BLOCK_TASK_BIT-1;    --0
-----------------------------------------------------------------------------------------
```

Listing A.2: AXI4 Lite Interface FSM (HDL); back to fig. 4.6.6.

```
----------------------------------------------------------------------
NST_CONTROL : process(state, S_AXI_AWVALID, S_AXI_ARVALID,
S_AXI_WVALID, S_AXI_BREADY, S_AXI_RREADY,I_TIMEOUT,I_WR_ACK,I_RD_ACK,
holded_wr_ack_i,holded_rd_ack_i)
----------------------------------------------------------------------
begin
    next_state <= state;
        case state is
            when S0_IDLE =>
                if S_AXI_AWVALID = '1' and S_AXI_WVALID = '1' then
                    next_state <= S1_WR_ADDR;
                elsif S_AXI_ARVALID = '1' then
                    next_state <= S5_RD_ADDR;
                else
                    next_state <= S0_IDLE;
                end if;
            when S1_WR_ADDR =>
                if (S_AXI_WVALID = '1') then
                    next_state <= S2_WR_DATA;
                else
                    next_state <= S1_WR_ADDR;
                end if;
            when S2_WR_DATA =>
                if S_AXI_BREADY = '1' then
                    next_state <= S3_ACK;
                elsif (S_AXI_BREADY = '0') then
                    next_state <= S4_WR_WAIT;
                else
                    next_state <= S2_WR_DATA;
                end if;
            when S3_ACK =>
                if I_WR_ACK = '1' or holded_wr_ack_i = '1' then
                   next_state <= S0_IDLE;
                elsif I_WR_ACK = '0' and holded_wr_ack_i = '0' then
                   next_state <= S8_WR_STANDBY;
                end if;
            when S4_WR_WAIT =>
                if S_AXI_BREADY = '1' or I_TIMEOUT = '1' then
                    next_state <= S3_ACK;
                else
                    next_state <= S4_WR_WAIT;
                end if;
            when S8_WR_STANDBY =>
                if I_WR_ACK = '1' or I_TIMEOUT = '1' then
                   next_state <= S0_IDLE;
                else
                   next_state <= S8_WR_STANDBY;
                end if;
            when S5_RD_ADDR =>
                if S_AXI_RREADY = '1' then
                    next_state <= S6_RD_DATA;
                elsif S_AXI_RREADY = '0' then
                    next_state <= S7_RD_WAIT;
                else
                    next_state <= S5_RD_ADDR;
                end if;
            when S6_RD_DATA =>
                if (S_AXI_AWVALID = '1' and S_AXI_WVALID = '1') and (I_RD_ACK = '1' or
                    holded_rd_ack_i = '1') then
                    next_state <= S1_WR_ADDR;
                elsif (S_AXI_AWVALID = '0' or S_AXI_WVALID = '0') and (I_RD_ACK = '1' or
                    holded_rd_ack_i = '1') then
                    next_state <= S0_IDLE;
                elsif (I_RD_ACK = '0' and holded_rd_ack_i = '0') then
                    next_state <= S9_RD_STANDBY;
                end if ;
            when S7_RD_WAIT =>
                if S_AXI_RREADY = '1' or I_TIMEOUT = '1' then
                    next_state <= S6_RD_DATA;
                else
                    next_state <= S7_RD_WAIT;
                end if;
            when S9_RD_STANDBY =>
```

```
                if (S_AXI_AWVALID = '1' and S_AXI_WVALID = '1') and (I_RD_ACK = '1' or
                    I_TIMEOUT = '1') then
                    next_state <= S1_WR_ADDR;
                elsif (S_AXI_AWVALID = '0' or S_AXI_WVALID = '0') and (I_RD_ACK = '1' or
                    I_TIMEOUT = '1') then
                    next_state <= S0_IDLE;
                else
                   next_state <= S9_RD_STANDBY;
                end if;
            when others =>
                null;
        end case;
end process NST_CONTROL;
-----------------------------------------------------------------------

-----------------------------------------------------------------------
OUT_CONTROL : process (state,S_AXI_BREADY,I_WR_ACK,S_AXI_RREADY,I_RD_ACK)
-----------------------------------------------------------------------
begin
    S_AXI_AWREADY <= '0';
    ready_i       <= '0';
    cs_D          <= '0';
    S_AXI_AWREADY <= '0';
    wr_ce_D       <= '0';
    S_AXI_WREADY  <= '0';
    latch_wdata_i <= '0';
    S_AXI_BVALID  <= '0';
    S_AXI_ARREADY <= '0';
    rd_ce_D       <= '0';
    rvalid_D      <= '0';
    latch_rdata_i <= '0';
    S_AXI_BRESP   <= (others => '0');
    S_AXI_RRESP   <= (others => '0');
    O_TRIGGER     <= '0';
    O_ENABLE      <= '0';
    O_WR_EVENT    <= '0';
    O_RD_EVENT    <= '0';
    hold_clear_i  <= '0';
    hold_wr_ce_i  <= '0';
    hold_rd_ce_i  <= '0';
    O_STANDBY     <= '0';

    case state is
        when S0_IDLE =>
            ready_i <= '1';
            hold_clear_i  <= '1';
        when S1_WR_ADDR =>
            cs_D <= '1';
            S_AXI_AWREADY <= '1';
        when S2_WR_DATA =>
            latch_wdata_i <= '1';
            cs_D <= '1';
            wr_ce_D <= '1';
            S_AXI_WREADY <= '1';
        when S3_ACK =>
            cs_D <= '1';
            S_AXI_BVALID <= '1';
            S_AXI_BRESP <= (others => '0');
        when S4_WR_WAIT =>
            O_TRIGGER     <= '1';
            O_ENABLE      <= '1';
            O_WR_EVENT <= S_AXI_BREADY;
            hold_wr_ce_i  <= '1';
        when S8_WR_STANDBY =>
            O_TRIGGER     <= '1';
            O_WR_EVENT    <= I_WR_ACK;
            O_ENABLE      <= '1';
            O_STANDBY     <= '1';
        when S5_RD_ADDR =>
            cs_D <= '1';
            S_AXI_ARREADY <= '1';
            rd_ce_D <= '1';
        when S6_RD_DATA =>
            cs_D <= '1';
            rvalid_D <= '1';
```

```vhdl
            S_AXI_RRESP <= (others => '0');
            latch_rdata_i <= '1';
        when S7_RD_WAIT =>
            O_TRIGGER     <= '1';
            O_ENABLE      <= '1';
            O_RD_EVENT    <= S_AXI_RREADY;
            hold_rd_ce_i <= '1';
        when S9_RD_STANDBY =>
            O_TRIGGER     <= '1';
            O_RD_EVENT    <= I_RD_ACK;
            O_ENABLE      <= '1';
            O_STANDBY     <= '1';
        when others =>
            null;
    end case;
end process OUT_CONTROL;
-----------------------------------------------------------------------

--------------------------------------------------
---- Datapath
--------------------------------------------------

-- State register
-----------------------------------------------------------------------
FFST : process (S_AXI_ACLK)
-----------------------------------------------------------------------
begin
    if rising_edge(S_AXI_ACLK) then
        if reset_i = '1' then
            state <= S0_IDLE;
        else
            state <= next_state;
        end if;
    end if;
end process FFST;
-----------------------------------------------------------------------
```

Listing A.3: Configuration Package: System-Level Datapath System Call Constants and Definitions (HDL); back to listing 4.8.

```vhdl
CONSTANT C_TIMEOUT_WIDTH        : NATURAL := 16;
CONSTANT RPARAM_TIMEOUT_STATUS  : NATURAL := C_MESSAGE_WIDTH-1;                         --[63]
CONSTANT PARAM_EVENT_TO_MONITOR : NATURAL := C_MESSAGE_WIDTH-2;                         --[62]
SUBTYPE PARAM_TIMEOUT_VAL IS std_logic_vector(C_TIMEOUT_WIDTH-1 downto 0);            --[15:0]
SUBTYPE RPARAM_REMAINING_TIME IS std_logic_vector(C_TIMEOUT_WIDTH-1 downto 0);        --[15:0]
------------------------------------------------------------------------------------------
CONSTANT C_MAX_BURST_LEN_BITS       : NATURAL := POW2(256);                 --(8 bits)
CONSTANT C_LBUS_BURST_SOURCE_WWIDTH : NATURAL := C_MACHINE_WIDTH;
CONSTANT C_LBUS_DEST_WWIDTH         : NATURAL := 10; -- destination address (10 bits)
CONSTANT C_LBUS_UNUSED              : NATURAL := C_MESSAGE_WIDTH-C_MAX_BURST_LEN_BITS-
    C_LBUS_BURST_SOURCE_WWIDTH-C_LBUS_DEST_WWIDTH; --13 bits
------------------------------------------------------------------------------------------
SUBTYPE PARAM_LBUS_BURST_LEN    IS std_logic_vector(C_MESSAGE_WIDTH-1 downto
    C_MESSAGE_WIDTH-C_MAX_BURST_LEN_BITS);                          --[63:56]
SUBTYPE RPARAM_LBUS_BURST_LEN   IS std_logic_vector(C_MESSAGE_WIDTH-1 downto
    C_MESSAGE_WIDTH-C_MAX_BURST_LEN_BITS);                          --[63:56]
SUBTYPE PARAM_LBUS_SOURCE       IS std_logic_vector(PARAM_LBUS_BURST_LEN'LOW-1 downto
    PARAM_LBUS_BURST_LEN'LOW-C_LBUS_BURST_SOURCE_WWIDTH);           --[55:24]
SUBTYPE RPARAM_LBUS_SOURCE      IS std_logic_vector(PARAM_LBUS_BURST_LEN'LOW-1 downto
    PARAM_LBUS_BURST_LEN'LOW-C_LBUS_BURST_SOURCE_WWIDTH);           --[55:24]
SUBTYPE PARAM_LBUS_OFFSET       IS std_logic_vector(PARAM_LBUS_SOURCE'LOW-C_LBUS_UNUSED-1
    downto 0);                                                      --[9:0]
------------------------------------------------------------------------------------------
CONSTANT C_MBUS_BURST_SOURCE_WWIDTH : NATURAL := C_MACHINE_WIDTH;
CONSTANT C_MBUS_OFFSET_WWIDTH       : NATURAL := C_MACHINE_WIDTH - C_MAX_BURST_LEN_BITS;
CONSTANT C_MBUS_TIMEOUT_VALUE       : NATURAL := 10;
CONSTANT C_MBUS_BURST_TIMEOUT_VALUE : NATURAL := 150;
------------------------------------------------------------------------------------------
SUBTYPE PARAM_MBUS_BURST_LEN    IS std_logic_vector(C_MESSAGE_WIDTH-1 downto
    C_MESSAGE_WIDTH-C_MAX_BURST_LEN_BITS);                                --[63:56]
CONSTANT RPARAM_M00_TIMEOUT     : NATURAL := PARAM_MBUS_BURST_LEN'LOW-1; --[55]
SUBTYPE PARAM_MBUS_SOURCE       IS std_logic_vector(PARAM_MBUS_BURST_LEN'LOW-1 downto
    PARAM_MBUS_BURST_LEN'LOW-C_MBUS_BURST_SOURCE_WWIDTH);                 --[55:24]
SUBTYPE RPARAM_MBUS_SOURCE      IS std_logic_vector(PARAM_MBUS_BURST_LEN'LOW-1 downto
    PARAM_MBUS_BURST_LEN'LOW-C_MBUS_BURST_SOURCE_WWIDTH);                 --[55:24]
SUBTYPE PARAM_MBUS_OFFSET       IS std_logic_vector(PARAM_MBUS_SOURCE'LOW-1 downto 0);
                                                                          --[23:0]
SUBTYPE RPARAM_MBUS_BURST_LEN   IS std_logic_vector(C_MESSAGE_WIDTH-1 downto
    C_MESSAGE_WIDTH-C_MAX_BURST_LEN_BITS);                                --[63:56]
------------------------------------------------------------------------------------------
CONSTANT MUTEX_OWNER_ID_WWIDTH  : NATURAL := C_MACHINE_WIDTH-2;
CONSTANT MUTEX_STATUS_WWIDTH    : NATURAL := C_MACHINE_WIDTH;
CONSTANT MTX_UNUSED_H_WIDTH     : NATURAL := 9;
CONSTANT MTX_ADDR_WIDTH         : NATURAL := 1;
------------------------------------------------------------------------------------------
SUBTYPE PARAM_MUTEX_DEC_ADDR    IS std_logic_vector(C_MESSAGE_WIDTH-MTX_UNUSED_H_WIDTH-1
    downto C_MESSAGE_WIDTH-MTX_UNUSED_H_WIDTH-MTX_ADDR_WIDTH);          --[54]
SUBTYPE PARAM_MUTEX_OWNER_ID    IS std_logic_vector(PARAM_MUTEX_DEC_ADDR'LOW-1 downto
    PARAM_MUTEX_DEC_ADDR'LOW-MUTEX_OWNER_ID_WWIDTH);                    --[53:24]
SUBTYPE RPARAM_MUTEX_STATUS     IS std_logic_vector(C_MESSAGE_WIDTH-MTX_UNUSED_H_WIDTH-1
    downto C_MESSAGE_WIDTH-MTX_UNUSED_H_WIDTH-MUTEX_STATUS_WWIDTH);     --[55:24]
------------------------------------------------------------------------------------------
```

Listing A.4: Kernel Package: KFIFO Create Command 64 (HDL); back to section 5.1.

```vhdl
-----------------------------------------------------------------------------------------
-- kfifo create word64
-----------------------------------------------------------------------------------------
procedure kfifo_create_cmd64(
    constant ki_feature           : in ki_feature_t;
    constant hint_cookie          : in natural;
    constant message_flags_events : in std_logic_vector(C_KIVIO_FLAGS_WWIDTH-1 downto 0);
    constant wcount_whence        : in natural;
    constant woffset              : in natural range 0 to 2**C_KIVIO_WOFFSET_WWIDTH-1;
             data                 : out std_logic_vector(C_MESSAGE_WIDTH-1 downto 0)) is
-----------------------------------------------------------------------------------------
begin
    data := (others => '0');
    case ki_feature is
        -------------------------------------------------------------------------------
        when KI_ECHO =>
            data(C_MESSAGE_WIDTH-1)     := '0'; --[63]
            data(C_MESSAGE_WIDTH-2)     := '1'; --[62]
            data(KIVIO_HINT'RANGE)      := std_logic_vector(to_unsigned(hint_cookie,
                KIVIO_HINT'LENGTH)); --[61:56]
            data(KIVIO_FEATURE_TYPE'RANGE) := std_logic_vector(to_unsigned(
                C_KIVIO_ARGS_QUERY,KIVIO_FEATURE_TYPE'LENGTH)); --[31:28]
            data(KIVIO_OP_ID'RANGE)     := std_logic_vector(to_unsigned(C_KI_ECHO_ID,
                KIVIO_OP_ID'LENGTH)); --[27:24]
            data(KIVIO_MESSAGE'RANGE)   := message_flags_events; --[15:0]
        -------------------------------------------------------------------------------
        when KI_SUBSCRIBE =>
            data(C_MESSAGE_WIDTH-1)     := '0'; --[63]
            data(C_MESSAGE_WIDTH-2)     := '1'; --[62]
            data(KIVIO_HINT'RANGE)      := std_logic_vector(to_unsigned(hint_cookie,
                KIVIO_HINT'LENGTH)); --[61:56]
            data(KIVIO_FEATURE_TYPE'RANGE) := std_logic_vector(to_unsigned(
                C_KIVIO_ARGS_CTRL,KIVIO_FEATURE_TYPE'LENGTH)); --[31:28]
            data(KIVIO_OP_ID'RANGE)     := std_logic_vector(to_unsigned(C_KI_SUBSCRIBE_ID,
                KIVIO_OP_ID'LENGTH)); --[27:24]
            data(KIVIO_FLAGS'RANGE)     := message_flags_events; --[15:0]
        -------------------------------------------------------------------------------
        when KI_UNSUBSCRIBE =>
            data(C_MESSAGE_WIDTH-1)     := '0'; --[63]
            data(C_MESSAGE_WIDTH-2)     := '0'; --[62]
            data(KIVIO_COOKIE'RANGE)    := std_logic_vector(to_unsigned(hint_cookie,
                KIVIO_COOKIE'LENGTH)); --[61:56]
            data(KIVIO_FEATURE_TYPE'RANGE) := std_logic_vector(to_unsigned(
                C_KIVIO_ARGS_CTRL,KIVIO_FEATURE_TYPE'LENGTH)); --[31:28]
            data(KIVIO_OP_ID'RANGE)     := std_logic_vector(to_unsigned(C_KI_UNSUBSCRIBE_ID
                ,KIVIO_OP_ID'LENGTH)); --[27:24]
        -------------------------------------------------------------------------------
        when KI_GET_SIZE =>
            data(C_MESSAGE_WIDTH-1)     := '0'; --[63]
            data(C_MESSAGE_WIDTH-2)     := '0'; --[62]
            data(KIVIO_COOKIE'RANGE)    := std_logic_vector(to_unsigned(hint_cookie,
                KIVIO_COOKIE'LENGTH)); --[61:56]
            data(KIVIO_FEATURE_TYPE'RANGE) := std_logic_vector(to_unsigned(
                C_KIVIO_ARGS_QUERY,KIVIO_FEATURE_TYPE'LENGTH)); --[31:28]
            data(KIVIO_OP_ID'RANGE)     := std_logic_vector(to_unsigned(C_KI_GET_SIZE_ID,
                KIVIO_OP_ID'LENGTH)); --[27:24]
        -------------------------------------------------------------------------------
        when KI_SYNC =>
            data(C_MESSAGE_WIDTH-1)     := '0'; --[63]
            data(C_MESSAGE_WIDTH-2)     := '0'; --[62]
            data(KIVIO_COOKIE'RANGE)    := std_logic_vector(to_unsigned(hint_cookie,
                KIVIO_COOKIE'LENGTH)); --[63:56]
            data(KIVIO_FEATURE_TYPE'RANGE) := std_logic_vector(to_unsigned(
                C_KIVIO_ARGS_CTRL,KIVIO_FEATURE_TYPE'LENGTH)); --[31:28]
            data(KIVIO_OP_ID'RANGE)     := std_logic_vector(to_unsigned(C_KI_SYNC_ID,
                KIVIO_OP_ID'LENGTH)); --[27:24]
        -------------------------------------------------------------------------------
        when KI_READ_LRAM =>
            data(C_MESSAGE_WIDTH-1)     := '0'; --[63]
            data(C_MESSAGE_WIDTH-2)     := '0'; --[62]
            data(KIVIO_COOKIE'RANGE)    := std_logic_vector(to_unsigned(hint_cookie,
                KIVIO_COOKIE'LENGTH)); --[61:56]
            data(KIVIO_WCOUNT'RANGE)    := std_logic_vector(to_unsigned(wcount_whence,
                KIVIO_WCOUNT'LENGTH)); --[55:32]
            data(KIVIO_FEATURE_TYPE'RANGE) := std_logic_vector(to_unsigned(
                C_KIVIO_ARGS_DXCHG,KIVIO_FEATURE_TYPE'LENGTH)); --[31:28]
            data(KIVIO_OP_ID'RANGE)     := std_logic_vector(to_unsigned(C_KI_READ_LRAM_ID,
                KIVIO_OP_ID'LENGTH)); --[27:24]
            data(KIVIO_WOFFSET'RANGE)   := std_logic_vector(to_unsigned(woffset,
                KIVIO_WOFFSET'LENGTH)); --[23:0]
        ----------------------------------------------------------------------------------
        when KI_WRITE_LRAM =>
            data(C_MESSAGE_WIDTH-1)     := '0'; --[63]
            data(C_MESSAGE_WIDTH-2)     := '0'; --[62]
            data(KIVIO_COOKIE'RANGE)    := std_logic_vector(to_unsigned(hint_cookie,
                KIVIO_COOKIE'LENGTH)); --[61:56]
            data(KIVIO_WCOUNT'RANGE)    := std_logic_vector(to_unsigned(wcount_whence,
                KIVIO_WCOUNT'LENGTH)); --[55:32]
            data(KIVIO_FEATURE_TYPE'RANGE) := std_logic_vector(to_unsigned(
                C_KIVIO_ARGS_DXCHG,KIVIO_FEATURE_TYPE'LENGTH)); --[31:28]
            data(KIVIO_OP_ID'RANGE)     := std_logic_vector(to_unsigned(C_KI_WRITE_LRAM_ID,
                KIVIO_OP_ID'LENGTH)); --[27:24]
            data(KIVIO_WOFFSET'RANGE)   := std_logic_vector(to_unsigned(woffset,
                KIVIO_WOFFSET'LENGTH)); --[23:0]
        ----------------------------------------------------------------------------------
        when KI_READ_SYSRAM =>
            data(C_MESSAGE_WIDTH-1)     := '0'; --[63]
            data(C_MESSAGE_WIDTH-2)     := '0'; --[62]
            data(KIVIO_COOKIE'RANGE)    := std_logic_vector(to_unsigned(hint_cookie,
                KIVIO_COOKIE'LENGTH)); --[61:56]
            data(KIVIO_WCOUNT'RANGE)    := std_logic_vector(to_unsigned(wcount_whence,
                KIVIO_WCOUNT'LENGTH)); --[55:32]
            data(KIVIO_FEATURE_TYPE'RANGE) := std_logic_vector(to_unsigned(
                C_KIVIO_ARGS_DXCHG,KIVIO_FEATURE_TYPE'LENGTH)); --[31:28]
            data(KIVIO_OP_ID'RANGE)     := std_logic_vector(to_unsigned(C_KI_READ_SYSRAM_ID
                ,KIVIO_OP_ID'LENGTH)); --[27:24]
            data(KIVIO_WOFFSET'RANGE)   := std_logic_vector(to_unsigned(woffset,
                KIVIO_WOFFSET'LENGTH)); --[23:0]
        ----------------------------------------------------------------------------------
        when KI_WRITE_SYSRAM =>
            data(C_MESSAGE_WIDTH-1)     := '0'; --[63]
            data(C_MESSAGE_WIDTH-2)     := '0'; --[62]
            data(KIVIO_COOKIE'RANGE)    := std_logic_vector(to_unsigned(hint_cookie,
                KIVIO_COOKIE'LENGTH)); --[61:56]
            data(KIVIO_WHENCE'RANGE)    := std_logic_vector(to_unsigned(wcount_whence,
                KIVIO_WHENCE'LENGTH)); --[55:32]
            data(KIVIO_FEATURE_TYPE'RANGE) := std_logic_vector(to_unsigned(
                C_KIVIO_ARGS_DXCHG,KIVIO_FEATURE_TYPE'LENGTH)); --[31:28]
            data(KIVIO_OP_ID'RANGE)     := std_logic_vector(to_unsigned(
                C_KI_WRITE_SYSRAM_ID,KIVIO_OP_ID'LENGTH)); --[27:24]
            data(KIVIO_WOFFSET'RANGE)   := std_logic_vector(to_unsigned(woffset,
                KIVIO_WOFFSET'LENGTH)); --[23:0]
        ----------------------------------------------------------------------------------
        when KI_SEEK =>
            -- MORE
        ----------------------------------------------------------------------------------
        when others => null;
    end case;
    ---
end procedure kfifo_create_cmd64;
------------------------------------------------------------------------------------------
```

Listing A.5: Kernel Package: LRAM Extended Features (HDL); back to fig. 5.1.9.

```
-----------------------------------------------------------------------------------------
-- kernel safe kfifo polled send command (and polled wait return) (local - using lram)
-----------------------------------------------------------------------------------------
procedure kernel_safe_kfifo_send_command_wait_return_local(
        signal kernel_call          : out kernel_call_t;
        signal kernel_response      : in kernel_response_t;
        signal sched_progress       : in std_logic_vector(C_SCHEDULER_PROGRESS_WIDTH-1
            downto 0);
        signal kfifo_args_buffer    : inout kfifo_arg_array_t;
        signal kfifo_rets_buffer    : inout kfifo_ret_array_t;
        signal timeout              : out boolean;
        signal remaining_time       : out natural;
        constant timeout_value      : in natural;
        signal args_in_masked       : inout std_logic_vector(C_MACHINE_WIDTH-1 downto 0);
        signal rets_out_masked      : inout std_logic_vector(C_MACHINE_WIDTH-1 downto 0);
        signal return_rcvd          : out std_ulogic;
        signal procedure_done       : out std_logic;
        signal mutex_status         : out std_logic_vector(C_MACHINE_WIDTH-1 downto 0);
               data                 : in std_logic_vector(C_MESSAGE_WIDTH-1 downto 0);
               read_return          : out std_logic_vector(C_MACHINE_WIDTH-1 downto 0);
        signal kfifo_base_offset    : in std_logic_vector(C_MACHINE_WIDTH-1 downto 0)) is
-----------------------------------------------------------------------------------------
-- MORE (Variables)
begin
    return_rcvd <= '0';
    case to_integer(unsigned(sched_progress)) is
        ---------------------------------------------------------------------------------
        when C_B_KFIFO_LMUTEX_LOCK | C_B_KFIFO_RET_PATH_LMUTEX_LOCK =>
            local_mutex_lock(kernel_call,kernel_response);
        ---------------------------------------------------------------------------------
        when C_B_KFIFO_READ_ARGUMENTS =>
            kernel_call.enable_index <= '1';
            kernel_call.inc_index <= '1';
            lbus_word_read_burst(kernel_call,kernel_response,C_KFIFO_ARG_WIDTH,to_integer(
                unsigned(kfifo_base_offset)) + (C_KFIFO_ARGS_BASE_OFFSET),read_args);
            kfifo_args_buffer(kernel_response.index_delayed_1) <= read_args;
        ---------------------------------------------------------------------------------
         when C_B_KFIFO_FULL_TEST =>
            kernel_call.this_call <= '1';
            kernel_call.enable_index    <= '0';
            kernel_call.inc_index       <= '0';
            kfifo_in    := kfifo_args_buffer(C_KFIFO_ARGS_IN_OFFSET);
            kfifo_out   := kfifo_args_buffer(C_KFIFO_ARGS_OUT_OFFSET);
            kfifo_mask  := kfifo_args_buffer(C_KFIFO_ARGS_MASK_OFFSET);
            kfifo_esize := kfifo_args_buffer(C_KFIFO_ARGS_ESIZE_OFFSET);
            kfifo_data  := kfifo_args_buffer(C_KFIFO_ARGS_DATA_OFFSET);
            result      := std_logic_vector(unsigned(kfifo_in) - unsigned(kfifo_out));
            ---
            if result = kfifo_mask then
                kernel_call.kfifo_status <= '1'; -- kfifo full
            else
                kernel_call.kfifo_status <= '0'; -- kfifo not full
            end if;
            lintc_word_write(kernel_call,kernel_response,"100"); -- LMUTEX IRQ
        ---------------------------------------------------------------------------------
        when C_B_KFIFO_INC_ARGS_IN =>
            kernel_call.enable_index <= '1';
            kernel_call.inc_index <= '0';
            inc_args_in := std_logic_vector(unsigned(kfifo_args_buffer(
                C_KFIFO_ARGS_IN_OFFSET)) + 1);
            args_in_masked <= std_logic_vector(word32_and(inc_args_in,kfifo_args_buffer(
                C_KFIFO_ARGS_MASK_OFFSET)));
            lbus_word_write(kernel_call,kernel_response,(to_integer(unsigned(
                kfifo_base_offset)) + (C_KFIFO_ARGS_BASE_OFFSET + C_KFIFO_ARGS_IN_OFFSET))
                ,args_in_masked);
        ---------------------------------------------------------------------------------
        when C_B_KFIFO_SEND_CMD_64 =>
           kernel_call.enable_index <= '1';
           kernel_call.inc_index <= '1';
           temp := resize(unsigned(kfifo_args_buffer(C_KFIFO_ARGS_IN_OFFSET)),temp'LENGTH);
           args_in_x2 := std_logic_vector(resize(shift_left(temp,1),args_in_x2'LENGTH));
           kfifo_offset := to_integer(unsigned(kfifo_args_buffer(C_KFIFO_RETS_DATA_OFFSET)) + unsigned(args_in_x2));
           aux_data_buffer(0) := data(63 downto 32); aux_data_buffer(1) := data(31 downto 0);
        lbus_word_write_burst(kernel_call,kernel_response,2,to_integer(unsigned(kfifo_base_offset)) + (kfifo_offset),aux_data_buffer(kernel_response.index));
        -----------------------------------------------------------------------------
        when C_B_KFIFO_LMUTEX_UNLOCK | C_B_KFIFO_RET_PATH_LMUTEX_UNLOCK | C_B_KFIFO_FULL_PATH_LMUTEX_UNLOCK =>
            local_mutex_unlock(kernel_call,kernel_response,mutex_status);
        -----------------------------------------------------------------------------
        when C_B_SLEEP_WAIT_RETURN | C_B_KFIFO_FULL_PATH_SLEEP =>
            kernel_call.enable_sched    <= '1';
            kernel_call.resched_req     <= '1';
            wait_timeout_elapsed(kernel_call,kernel_response,timeout,remaining_time,timeout_value);
        -----------------------------------------------------------------------------
        when C_B_KFIFO_READ_RETURNS =>
            kernel_call.enable_index <= '1';
            kernel_call.inc_index <= '1';
            lbus_word_read_burst(kernel_call,kernel_response,C_KFIFO_RET_WIDTH,to_integer(unsigned(kfifo_base_offset)) + (C_KFIFO_RETS_BASE_OFFSET),read_args);
            kfifo_rets_buffer(kernel_response.index_delayed_1) <= read_args;
        -----------------------------------------------------------------------------
        when C_B_KFIFO_RET_EMPTY_TEST =>
            kernel_call.this_call <= '1';
            kernel_call.enable_index    <= '0';
            kernel_call.inc_index       <= '0';
            kfifo_in    := kfifo_rets_buffer(C_KFIFO_RETS_IN_OFFSET);
            kfifo_out   := kfifo_rets_buffer(C_KFIFO_RETS_OUT_OFFSET);
            kfifo_mask  := kfifo_rets_buffer(C_KFIFO_RETS_MASK_OFFSET);
            kfifo_esize := kfifo_rets_buffer(C_KFIFO_RETS_ESIZE_OFFSET);
            kfifo_data  := kfifo_rets_buffer(C_KFIFO_RETS_DATA_OFFSET);
            result      := std_logic_vector(unsigned(kfifo_in) - unsigned(kfifo_out));
            ---
            if to_integer(unsigned(result)) /= 0 then
                return_rcvd <= '1'; -- return received
            else
                return_rcvd <= '0'; -- return not received
            end if;
            lintc_word_write(kernel_call,kernel_response,"100"); -- LMUTEX IRQ
        -----------------------------------------------------------------------------
        when C_B_KFIFO_INC_RETS_OUT =>
            kernel_call.enable_index <= '1';
            kernel_call.inc_index <= '0'; --
            inc_rets_out := std_logic_vector(unsigned(kfifo_rets_buffer(C_KFIFO_RETS_OUT_OFFSET)) + 1);
            rets_out_masked <= std_logic_vector(word32_and(inc_rets_out,kfifo_rets_buffer(C_KFIFO_RETS_MASK_OFFSET)));
            lbus_word_write(kernel_call,kernel_response,to_integer(unsigned(kfifo_base_offset)) + (C_KFIFO_RETS_BASE_OFFSET + C_KFIFO_RETS_OUT_OFFSET),rets_out_masked);
        -----------------------------------------------------------------------------
        when C_B_KFIFO_READ_PROC_RET_WORD =>
            kernel_call.enable_index <= '1';
            kernel_call.inc_index <= '0';
            kfifo_offset := to_integer(unsigned(kfifo_rets_buffer(C_KFIFO_RETS_DATA_OFFSET)) + unsigned(kfifo_rets_buffer(C_KFIFO_RETS_OUT_OFFSET)));
            lbus_word_read(kernel_call,kernel_response,to_integer(unsigned(kfifo_base_offset)) + (C_KFIFO_RETS_BASE_OFFSET + kfifo_offset),read_return);
        -----------------------------------------------------------------------------
        when C_B_PROCEDURE_DONE =>
            procedure_done <= '1';
        -----------------------------------------------------------------------------
        when others =>
            null;
     end case;
    ---
end procedure kernel_safe_kfifo_send_command_wait_return_local;
-----------------------------------------------------------------------------------------
```

Listing A.6: Hardware Mutex Acknowledge Generation (HDL); back to section 4.5.1.

```
-- Channel A write ack generation
-------------------------------------------------------------------------------------------
reset_FF2 <= not(I_WR_CE_A) and write_a_Q;
write_a_i <= cs_a_i and I_WR_CE_A and not(write_a_Q);
-------------------------------------------------------------------------------------------
WR_ACK_A_GEN_FF2 : process (I_CLK)
-------------------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if (I_RESET = '1') or (reset_FF2 = '1') then
            write_a_Q <= '0';
        elsif (I_WR_CE_A = '1') then
            write_a_Q <= cs_a_i;
        end if;
    end if;
end process WR_ACK_A_GEN_FF2;
-------------------------------------------------------------------------------------------
O_WR_ACK_A <= write_a_Q;

-- Channel B write ack generation
-------------------------------------------------------------------------------------------
reset_FF3 <= not(I_WR_CE_B) and write_b_Q;
write_b_i <= I_CS_B and I_WR_CE_B and not(write_b_Q);
-------------------------------------------------------------------------------------------
WR_ACK_B_GEN_FF3 : process (I_CLK)
-------------------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if (I_RESET = '1') or (reset_FF3 = '1') then
            write_b_Q <= '0';
        elsif (I_WR_CE_B = '1') then
            write_b_Q <= I_CS_B;
        end if;
    end if;
end process WR_ACK_B_GEN_FF3;
-------------------------------------------------------------------------------------------
O_WR_ACK_B <= write_b_Q;

-- Channel A read ack generation
-------------------------------------------------------------------------------------------
reset_FF4 <= not(I_RD_CE_A) and read_a_Q;
-------------------------------------------------------------------------------------------
RD_ACK_A_GEN_FF4 : process (I_CLK)
-------------------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if (I_RESET = '1') or (reset_FF4 = '1') then
            read_a_Q <= '0';
        elsif (I_RD_CE_A = '1') then
            read_a_Q <= cs_a_i;
        end if;
    end if;
end process RD_ACK_A_GEN_FF4;
-------------------------------------------------------------------------------------------
O_RD_ACK_A <= read_a_Q;

-- Channel B read ack generation
-------------------------------------------------------------------------------------------
reset_FF5 <= not(I_RD_CE_B) and read_b_Q;
-------------------------------------------------------------------------------------------
RD_ACK_B_GEN_FF5 : process (I_CLK)
-------------------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if (I_RESET = '1') or (reset_FF5 = '1') then
            read_b_Q <= '0';
        elsif (I_RD_CE_B = '1') then
            read_b_Q <= I_CS_B;
        end if;
    end if;
end process RD_ACK_B_GEN_FF5;
```

```
------------------------------------------------------------------------------------------
O_RD_ACK_B <= read_b_Q;
```

Listing A.7: Hardware Mutex FSM (HDL); back to fig. 4.5.2.

```
-- MORE
------------------------------------------------------------------------------------------
---- Control
------------------------------------------------------------------------------------------

-- Next state logic
------------------------------------------------------------------------------------------
NST_CONTROL : process(state,write_b_i,write_a_i,write_b_q,valid_i,
write_a_q)
------------------------------------------------------------------------------------------
begin
    next_state <= state;
        case state is
            -- Idle state
            when S0_FREE =>
                if write_b_i = '1' then
                    next_state <= S1_ACCEPT_B;
                elsif write_a_i = '1' and write_b_i = '0' then
                    next_state <= S4_ACCEPT_A;
                else
                    next_state <= S0_FREE;
                end if;
            when S1_ACCEPT_B =>
                    next_state <= S2_OWNED_B;
            when S2_OWNED_B =>
                if write_b_q = '1' and valid_i = '1' then
                    next_state <= S3_RELEASE_B;
                else
                    next_state <= S2_OWNED_B;
                end if;
            when S3_RELEASE_B =>
                next_state <= S0_FREE;
            when S4_ACCEPT_A =>
                next_state <= S5_OWNED_A;
            when S5_OWNED_A =>
                if write_a_q = '1' and valid_i = '1' then
                    next_state <= S6_RELEASE_A;
                else
                    next_state <= S5_OWNED_A;
                end if;
            when S6_RELEASE_A =>
                next_state <= S0_FREE;
            when others =>
                null;
        end case;
end process NST_CONTROL;
------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------
OUT_CONTROL : process (state)
------------------------------------------------------------------------------------------
begin
    select_i    <= (others => '0');
    write_i     <= '0';
    clear_b_i   <= '0';
    clear_a_i   <= '0';
    lock_i      <= '0';
    clear_i     <= '0';

    case state is
        when S0_FREE =>
            select_i  <= "11";
            clear_i   <= '1';
        when S1_ACCEPT_B =>
            select_i  <= "01";
            clear_b_i <= '1';
            write_i   <= '1';
            lock_i    <= '1';
        when S2_OWNED_B =>
            select_i  <= "11";
            lock_i    <= '1';
        when S3_RELEASE_B =>
            select_i  <= "01";
            clear_b_i <= '1';
            write_i   <= '1';
        when S4_ACCEPT_A =>
            select_i  <= "00";
            clear_a_i <= '1';
            write_i   <= '1';
            lock_i    <= '1';
        when S5_OWNED_A =>
            select_i  <= "10";
            lock_i    <= '1';
        when S6_RELEASE_A =>
            select_i  <= "00";
            clear_a_i <= '1';
            write_i   <= '1';
        when others =>
            null;
    end case;
end process OUT_CONTROL;
------------------------------------------------------------------------------------------

-- State register
------------------------------------------------------------------------------------------
FFST : process (I_CLK)
------------------------------------------------------------------------------------------
begin
    if rising_edge(I_CLK) then
        if I_RESET = '1' then
            state <= S0_FREE;
        else
            state <= next_state;
        end if;
    end if;
end process FFST;
------------------------------------------------------------------------------------------
end architecture rtl;
```

Listing A.8: Pipe Hardware Task: SYSRAM Communication (HDL); back to section 5.1.2.

```
------------------------------------------------------------------------------------------
-- MORE
------------------------------------------------------------------------------------------
hwtask_reset_i <= not (I_HWT_RESET);
------------------------------------------------------------------------------------------
TASK_REGS : process (I_HWT_CLK)
------------------------------------------------------------------------------------------
begin
    if rising_edge(I_HWT_CLK) then
        if (hwtask_reset_i = '1') then
            data_pipe_ret_cookie_q <= 0;
            res_pipe_ret_cookie_q  <= 0;
            wwritten_q <= 0;
            wread_q <= 0;
            out_buffer_0_q <= (others => (others => '0'));
            in_buffer_0_q <= (others => (others => '0'));
            -- More: VIP Setup (arguments)
            -- More: VIP Setup (returns)
        else
            data_pipe_ret_cookie_q <= data_pipe_ret_cookie_d;
            res_pipe_ret_cookie_q  <= res_pipe_ret_cookie_d;
            wwritten_q <= wwritten_d;
            wread_q <= wread_d;
            out_buffer_0_q <= out_buffer_0_d;
            in_buffer_0_q <= in_buffer_0_d;
            -- More: VIP Setup (arguments)
            -- More: VIP Setup (returns)
        end if;
    end if;
end process TASK_REGS;
------------------------------------------------------------------------------------------

-- import kernel response
------------------------------------------------------------------------------------------
kernel_to_task_import_response(KERNEL_RESPONSE,
                               I_SYSCALL_VALID,
                               I_SYSCALL_BLOCK_TASK,
                               I_SERVICE_INDEX,
                               I_SERVICE_INDEX_D1,
                               I_SCHEDULER_PROGRESS,
                               I_SCHEDULER_PROCEDURE_ID,
                               I_SYSCALL_SYSCALL_ID,
                               I_SYSCALL_RETURN_ARG,
                               I_PROCEDURE_RETURN_ARG,
                               I_TASK_RUN,
                               I_TASK_RESET);
------------------------------------------------------------------------------------------

-- export task call
------------------------------------------------------------------------------------------
task_to_kernel_export_call(KERNEL_CALL,
                           O_SYSCALL_THIS_CALL,
                           O_SERVICE_ENABLE_INDEX,
                           O_SERVICE_INC_INDEX,
                           O_SCHEDULER_ENABLE,
                           O_SCHEDULER_RESCHED_REQ,
                           O_SCHEDULER_PROCEDURE_ID,
                           O_KFIFO_STATUS,
                           O_KFIFO_CMD_ID,
                           O_BOUND_TO_KERNEL,
                           O_SYSCALL_SYSCALL_ID,
                           O_SYSCALL_PARAMETERS,
                           O_PROCEDURE_PARAMETERS,
                           O_TASK_DONE);
------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------
HW_TASK_CONTROL : process(state,KERNEL_CALL,KERNEL_RESPONSE,
data_pipe_ret_cookie_q,res_pipe_ret_cookie_q,wwritten_q,out_buffer_0_q,
in_buffer_0_q,xor_done_q2,timeout_i,remaining_time_i,wread_q,out_buffer_0_d,
aux_buffer_0_q,aux_buffer_1_q)
------------------------------------------------------------------------------------------
```

```
begin
    data_pipe_ret_cookie_d <= data_pipe_ret_cookie_q;
    res_pipe_ret_cookie_d  <= res_pipe_ret_cookie_q;
    in_buffer_0_d <= in_buffer_0_q;
    trigger_op_i <= '0';
    timeout_i <= false;
    remaining_time_i <= 0;
    wwritten_d <= wwritten_q;
    wread_d <= wread_q;
    next_state <= state;
    task_ukernel_sync(KERNEL_CALL,KERNEL_RESPONSE);
    -------------------------------------------------------------------------------------
        case state is
            -----------------------------------------------------------------------------
            when S0_READY =>
                if KERNEL_RESPONSE.task_run = '1' then
                    next_state <= S1_SUB_DATA_PIPE;
                    task_ukernel_sync(KERNEL_CALL,KERNEL_RESPONSE);
                end if;
            -----------------------------------------------------------------------------
            -- More: VIP State Setup (arguments)
            -----------------------------------------------------------------------------
            -- More: VIP State Setup (returns)
            -----------------------------------------------------------------------------
            when S1_SUB_DATA_PIPE =>
                polled_kernel_bounded_file_subscribe(KERNEL_CALL,KERNEL_RESPONSE,128,
                    data_pipe_flags_q,0,0,data_pipe_ret_cookie_d);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S2_SUB_RES_PIPE;
                end if;
            ----------------------------------------------------------------------
            when S2_SUB_RES_PIPE  =>
                polled_kernel_bounded_file_subscribe(KERNEL_CALL,KERNEL_RESPONSE,129,
                    res_pipe_flags_q,0,0,res_pipe_ret_cookie_d);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S3_FETCH_TO_SYSRAM;
                end if;
            ----------------------------------------------------------------------
            when S3_FETCH_TO_SYSRAM  =>
                polled_kernel_bounded_sysram_write(KERNEL_CALL,KERNEL_RESPONSE,
                    data_pipe_ret_cookie_q,(others => '0'),C_WCOUNT,C_WOFFSET,wwritten_d);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S4_LOCK_SYSMUTEX;
                end if;
            ----------------------------------------------------------------------
            when S4_LOCK_SYSMUTEX  =>
                hwt_system_mutex_lock(KERNEL_CALL,KERNEL_RESPONSE);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S5_READ_SYSRAM;
                end if;
            ----------------------------------------------------------------------
            when S5_READ_SYSRAM  =>
                sysram_read_burst(KERNEL_CALL,KERNEL_RESPONSE,in_buffer_0_d,32,C_WOFFSET);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S6_UNLOCK_SYSMUTEX;
                end if;
            ----------------------------------------------------------------------
            when S6_UNLOCK_SYSMUTEX  =>
                hwt_system_mutex_unlock(KERNEL_CALL,KERNEL_RESPONSE,mutex_status_i);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S7_TRIGGER_OP;
                end if;
            ----------------------------------------------------------------------

            when S7_TRIGGER_OP  =>
                trigger_op_i <= '1';
                next_state <= S8_WAIT_OP;
            ----------------------------------------------------------------------
            when S8_WAIT_OP  =>
                wait_event_elapsed(KERNEL_CALL,KERNEL_RESPONSE,xor_done_q2,timeout_i,
                    remaining_time_i,C_MAX_WAIT_TIME);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S9_LOCK_SYSMUTEX;
                end if;
```

```vhdl
            ----------------------------------------------------------------------
            when S9_LOCK_SYSMUTEX  =>
                hwt_system_mutex_lock(KERNEL_CALL,KERNEL_RESPONSE);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S10_WRITE_SYSRAM;
                end if;
            ----------------------------------------------------------------------
            when S10_WRITE_SYSRAM  =>
               sysram_write_burst(KERNEL_CALL,KERNEL_RESPONSE,out_buffer_0_q,32,C_WOFFSET);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S11_UNLOCK_SYSMUTEX;
                end if;
            ----------------------------------------------------------------------
            when S11_UNLOCK_SYSMUTEX  =>
                hwt_system_mutex_unlock(KERNEL_CALL,KERNEL_RESPONSE,mutex_status_i);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S12_PUBLISH_FROM_SYSRAM;
                end if;
            ----------------------------------------------------------------------
            when S12_PUBLISH_FROM_SYSRAM  =>
                polled_kernel_bounded_sysram_read(KERNEL_CALL,KERNEL_RESPONSE,
                    res_pipe_ret_cookie_q,(others => '0'),C_WCOUNT,C_WOFFSET,wread_d);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S99_EXIT;
                end if;
            ----------------------------------------------------------------------
            when S99_EXIT  =>
                KERNEL_CALL.task_done <= '1';
                next_state <= S99_EXIT;
            ----------------------------------------------------------------------

            when others => null;
        end case;
end process HW_TASK_CONTROL;
------------------------------------------------------------------

------------------------------------------------------------------
IP_XOR : entity data_op
------------------------------------------------------------------
port map (
I_CLK    => I_HWT_CLK,
I_RESET  => hwtask_reset_i,
I_RUN    => trigger_op_i,
I_BUFFER => in_buffer_0_d,
O_BUFFER => out_buffer_0_d,
O_DONE   => xor_done_i
);
------------------------------------------------------------------

------------------------------------------------------------------
XOR_DONE_SYNC : process (I_HWT_CLK)
------------------------------------------------------------------
begin
    if rising_edge(I_HWT_CLK) then
        if hwtask_reset_i = '1' then
            xor_done_q  <= '0';
            xor_done_q2 <= '0';
        else
            xor_done_q  <= xor_done_i;
            xor_done_q2 <= xor_done_q;
        end if;
    end if;
end process XOR_DONE_SYNC;
------------------------------------------------------------------

------------------------------------------------------------------
FFST : process (I_HWT_CLK)
------------------------------------------------------------------
begin
    if rising_edge(I_HWT_CLK) then
        if hwtask_reset_i = '1' or KERNEL_RESPONSE.task_reset = '1' then
            state <= S0_READY;
        elsif I_SYSCALL_BLOCK_TASK = '0' then
            state <= next_state;
        end if;
    end if;
end process FFST;
------------------------------------------------------------------
```

Listing A.9: Pipe Hardware Task: LRAM Communication (HDL); back to section 5.1.2.

```
------------------------------------------------------------------------------------------
-- MORE
------------------------------------------------------------------------------------------
hwtask_reset_i <= not (I_HWT_RESET);
------------------------------------------------------------------------------------------
TASK_REGS : process (I_HWT_CLK)
------------------------------------------------------------------------------------------
begin
    if rising_edge(I_HWT_CLK) then
        if (hwtask_reset_i = '1') then
            data_pipe_ret_cookie_q <= 0;
            res_pipe_ret_cookie_q  <= 0;
            wwritten_q <= 0;
            wread_q <= 0;
            out_buffer_0_q <= (others => (others => '0'));
            in_buffer_0_q <= (others => (others => '0'));
            -- More: LRAM Setup (arguments)
            -- More: LRAM Setup (returns)
            -- More: LRAM Setup (data)
            ---
        else
            data_pipe_ret_cookie_q <= data_pipe_ret_cookie_d;
            res_pipe_ret_cookie_q  <= res_pipe_ret_cookie_d;
            wwritten_q <= wwritten_d;
            wread_q <= wread_d;
            out_buffer_0_q <= out_buffer_0_d;
            in_buffer_0_q <= in_buffer_0_d;
            -- More: LRAM Setup (arguments)
            -- More: LRAM Setup (returns)
            -- More: LRAM Setup (data)
        end if;
    end if;
end process TASK_REGS;
------------------------------------------------------------------

-- import kernel response
------------------------------------------------------------------
kernel_to_task_import_response(KERNEL_RESPONSE,
                               I_SYSCALL_VALID,
                               I_SYSCALL_BLOCK_TASK,
                               I_SERVICE_INDEX,
                               I_SERVICE_INDEX_D1,
                               I_SCHEDULER_PROGRESS,
                               I_SCHEDULER_PROCEDURE_ID,
                               I_SYSCALL_SYSCALL_ID,
                               I_SYSCALL_RETURN_ARG,
                               I_PROCEDURE_RETURN_ARG,
                               I_TASK_RUN,
                               I_TASK_RESET);
------------------------------------------------------------------

-- export task call
------------------------------------------------------------------
task_to_kernel_export_call(KERNEL_CALL,
                           O_SYSCALL_THIS_CALL,
                           O_SERVICE_ENABLE_INDEX,
                           O_SERVICE_INC_INDEX,
                           O_SCHEDULER_ENABLE,
                           O_SCHEDULER_RESCHED_REQ,
                           O_SCHEDULER_PROCEDURE_ID,
                           O_KFIFO_STATUS,
                           O_KFIFO_CMD_ID,
                           O_BOUND_TO_KERNEL,
                           O_SYSCALL_SYSCALL_ID,
                           O_SYSCALL_PARAMETERS,
                           O_PROCEDURE_PARAMETERS,
                           O_TASK_DONE);
------------------------------------------------------------------

------------------------------------------------------------------
HW_TASK_CONTROL : process(state,KERNEL_CALL,KERNEL_RESPONSE,
data_pipe_ret_cookie_q,res_pipe_ret_cookie_q,wwritten_q,out_buffer_0_q,
in_buffer_0_q,xor_done_q2,timeout_i,remaining_time_i,wread_q,out_buffer_0_d,
aux_buffer_0_q,aux_buffer_1_q,lram_words_buffer_q,mutex_status_i)
------------------------------------------------------------------
begin
    data_pipe_ret_cookie_d <= data_pipe_ret_cookie_q;
    res_pipe_ret_cookie_d  <= res_pipe_ret_cookie_q;
    in_buffer_0_d <= in_buffer_0_q;
    trigger_op_i <= '0';
    timeout_i <= false;
    remaining_time_i <= 0;
    wwritten_d <= wwritten_q;
    wread_d <= wread_q;
    next_state <= state;
    task_ukernel_sync(KERNEL_CALL,KERNEL_RESPONSE);
    --------------------------------------------------------------
        case state is
        ----------------------------------------------------------------------
            when S0_READY =>
                if KERNEL_RESPONSE.task_run = '1' then
                    next_state <= S1_SUB_DATA_PIPE;
                    task_ukernel_sync(KERNEL_CALL,KERNEL_RESPONSE);
                end if;
            ----------------------------------------------------------------------
            -- More: LRAM Setup State (arguments)
            -- More: LRAM Setup State (returns)
            -- More: LRAM Setup State (data)
            ----------------------------------------------------------------------
            when S1_SUB_DATA_PIPE =>
                polled_kernel_bounded_file_subscribe_local(KERNEL_CALL,KERNEL_RESPONSE,128,
                    data_pipe_flags_q,0,0,data_pipe_ret_cookie_d);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S2_SUB_RES_PIPE;
                end if;
            ----------------------------------------------------------------------
            when S2_SUB_RES_PIPE  =>
                polled_kernel_bounded_file_subscribe_local(KERNEL_CALL,KERNEL_RESPONSE,129,
                    res_pipe_flags_q,0,0,res_pipe_ret_cookie_d);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S3_FETCH_TO_LRAM;
                end if;
            ----------------------------------------------------------------------
            when S3_FETCH_TO_LRAM  =>
                polled_kernel_bounded_lram_write_local(KERNEL_CALL,KERNEL_RESPONSE,
                    data_pipe_ret_cookie_q,(others => '0'),C_WCOUNT,C_WOFFSET,wwritten_d);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S4_LOCK_LMUTEX;
                end if;
            ----------------------------------------------------------------------
            when S4_LOCK_LMUTEX  =>
                hwt_local_mutex_lock(KERNEL_CALL,KERNEL_RESPONSE);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S5_READ_LRAM;
                end if;
            ----------------------------------------------------------------------
            when S5_READ_LRAM  =>
                lram_read_burst(KERNEL_CALL,KERNEL_RESPONSE,in_buffer_0_d,32,512);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S6_UNLOCK_LMUTEX;
                end if;
            ----------------------------------------------------------------------
            when S6_UNLOCK_LMUTEX  =>
                hwt_local_mutex_unlock(KERNEL_CALL,KERNEL_RESPONSE,mutex_status_i);
                if KERNEL_RESPONSE.valid = '1' then
                    next_state <= S7_TRIGGER_OP;
                end if;
            ----------------------------------------------------------------------

            when S7_TRIGGER_OP  =>
```

```
                    trigger_op_i <= '1';
                    next_state <= S8_WAIT_OP;
                ----------------------------------------------------------------------
                when S8_WAIT_OP  =>
                    wait_event_elapsed(KERNEL_CALL,KERNEL_RESPONSE,xor_done_q2,timeout_i,
                        remaining_time_i,C_MAX_WAIT_TIME);
                    if KERNEL_RESPONSE.valid = '1' then
                        next_state <= S9_LOCK_LMUTEX;
                    end if;
                ----------------------------------------------------------------------
                when S9_LOCK_LMUTEX  =>
                    hwt_local_mutex_lock(KERNEL_CALL,KERNEL_RESPONSE);
                    if KERNEL_RESPONSE.valid = '1' then
                        next_state <= S10_WRITE_LRAM;
                    end if;
                ----------------------------------------------------------------------
                when S10_WRITE_LRAM  =>
                    lram_write_burst(KERNEL_CALL,KERNEL_RESPONSE,out_buffer_0_q,32,512);
                    if KERNEL_RESPONSE.valid = '1' then
                        next_state <= S11_UNLOCK_LMUTEX;
                    end if;
                ----------------------------------------------------------------------
                when S11_UNLOCK_LMUTEX  =>
                    hwt_local_mutex_unlock(KERNEL_CALL,KERNEL_RESPONSE,mutex_status_i);
                    if KERNEL_RESPONSE.valid = '1' then
                        trigger_op_i <= '1';
                        next_state <= S12_PUBLISH_FROM_LRAM;
                    end if;
                ----------------------------------------------------------------------
                when S12_PUBLISH_FROM_LRAM  =>
                    polled_kernel_bounded_lram_read_local(KERNEL_CALL,KERNEL_RESPONSE,
                        res_pipe_ret_cookie_q,(others => '0'),C_WCOUNT,C_WOFFSET,wread_d);
                    if KERNEL_RESPONSE.valid = '1' then
                        next_state <= S99_EXIT;
                    end if;
                ----------------------------------------------------------------------
                when S99_EXIT  =>
                    KERNEL_CALL.task_done <= '1';
                    next_state <= S99_EXIT;
                ----------------------------------------------------------------------

                when others =>
                    null;
            end case;
end process HW_TASK_CONTROL;
------------------------------------------------------------------

------------------------------------------------------------------
IP_XOR : entity data_op
------------------------------------------------------------------
port map (
I_CLK    => I_HWT_CLK,
I_RESET  => hwtask_reset_i,
I_RUN    => trigger_op_i,
I_BUFFER => in_buffer_0_d,
O_BUFFER => out_buffer_0_d,
O_DONE   => xor_done_i
);
------------------------------------------------------------------

------------------------------------------------------------------
XOR_DONE_SYNC : process (I_HWT_CLK)
------------------------------------------------------------------
begin
    if rising_edge(I_HWT_CLK) then
        if hwtask_reset_i = '1' then
            xor_done_q  <= '0';
            xor_done_q2 <= '0';
        else
            xor_done_q  <= xor_done_i;
            xor_done_q2 <= xor_done_q;
        end if;
    end if;
end process XOR_DONE_SYNC;
```

```
-------------------------------------------------------------------

-------------------------------------------------------------------
FFST : process (I_HWT_CLK)
-------------------------------------------------------------------
begin
    if rising_edge(I_HWT_CLK) then
        if hwtask_reset_i = '1' or KERNEL_RESPONSE.task_reset = '1' then
            state <= S0_READY;
        elsif I_SYSCALL_BLOCK_TASK = '0' then
            state <= next_state;
        end if;
    end if;
end process FFST;
-------------------------------------------------------------------
```

B. Auxiliary Resources

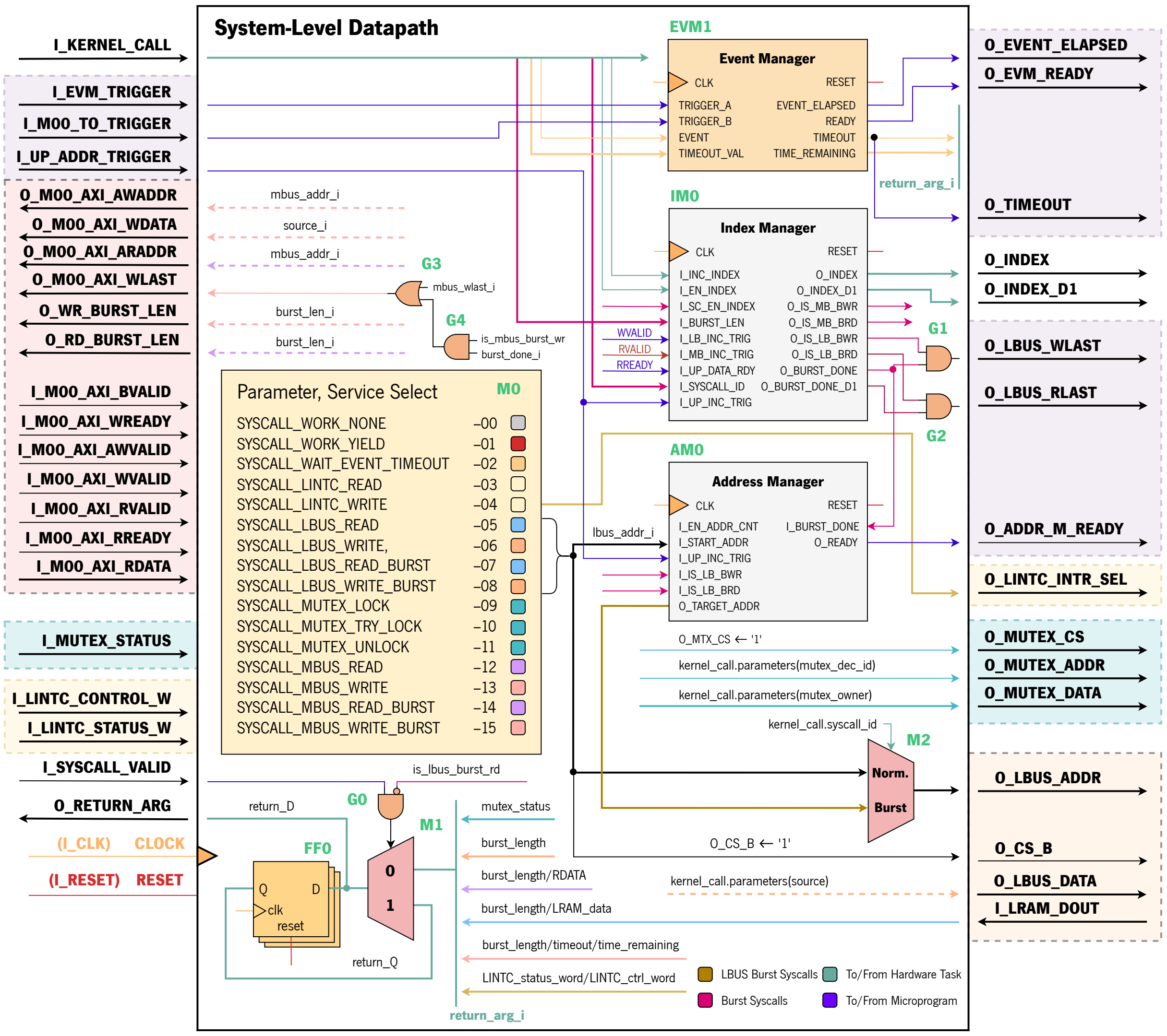


Figure B.1: System-Level Datapath (SLD) RTL Design Internal Architecture (Expanded), back to fig. 4.2.11.

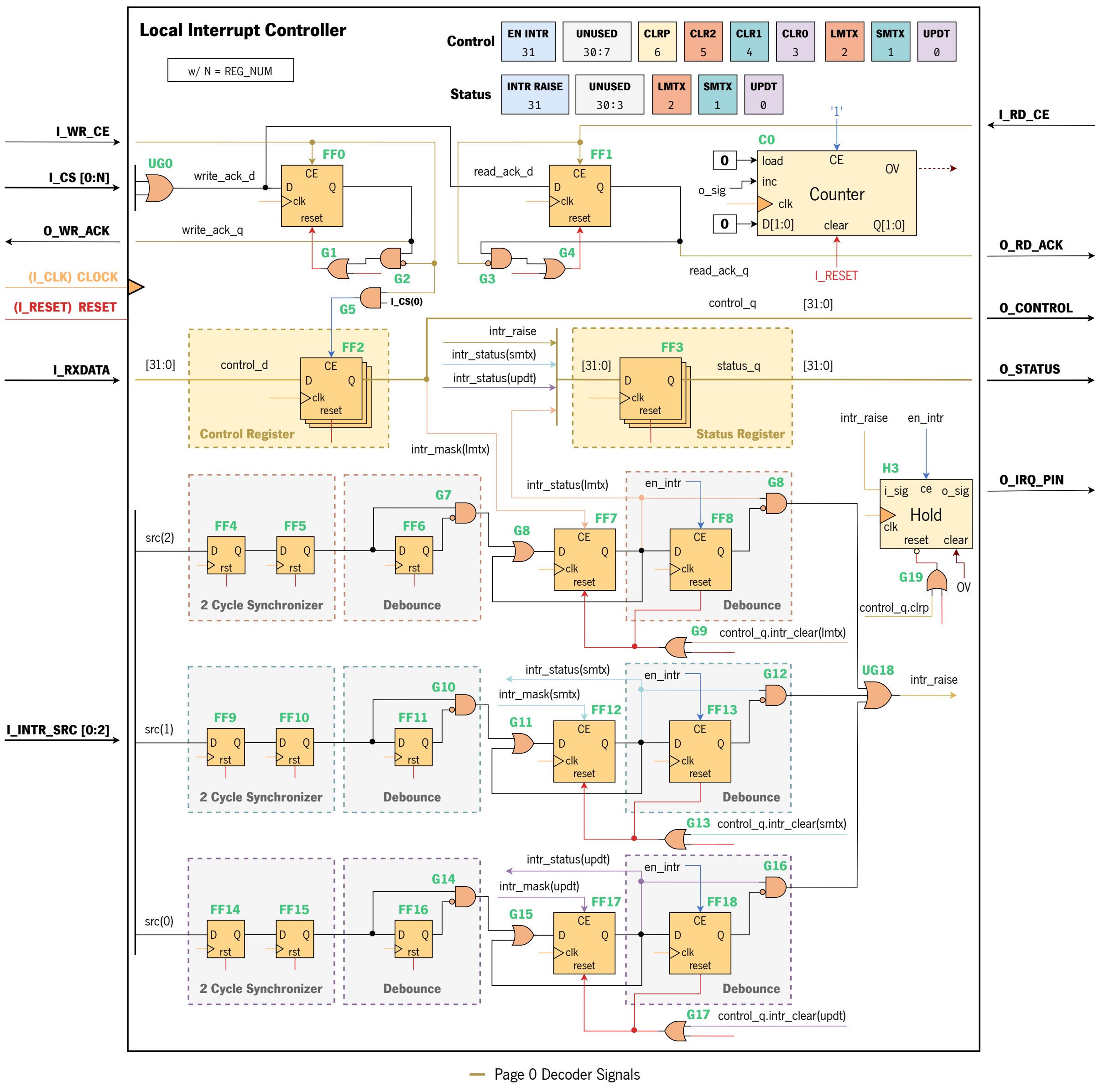


Figure B.2: Local Interrupt Controller RTL Design Internal Architecture (Expanded), back to fig. 4.5.3.

Table B.1: Simplified Microprogram Memory Representation (Part 1), back to fig. 4.2.9.

Microprogram Memory							
Syscall (KL)	**Progress**	**Input Sel.**	**NSF**	**Output Sel.**	**Valid**	**Block**	**Fault**
SYSCALL_WORK_NONE (0)	00	AUX_FALSE	00	NULL	1	0	0
	01	AUX_FALSE	00	NULL	0	1	0
	10	AUX_FALSE	00	NULL	0	1	0
	11	AUX_FALSE	00	NULL	0	1	0
SYSCALL_WORK_YIELD (1)	00	ACC_DEAD	00	KILL_ACC	0	1	0
	01	AUX_FALSE	00	NULL	1	0	0
	10	AUX_FALSE	00	NULL	0	1	0
	11	AUX_FALSE	00	NULL	0	1	0
SYSCALL_WAIT_EVENT_TIMEOUT (2)	00	EM_READY	00	NULL	0	1	0
	01	EVENT_ELAPSED	01	EM_TRIGGER	0	1	0
	10	AUX_FALSE	00	NULL	1	0	0
	11	AUX_FALSE	00	NULL	0	1	0
SYSCALL_LINTC_READ (3)	00	AUX_FALSE	01	NULL	1	0	0
	01	AUX_FALSE	00	NULL	0	1	0
	10	AUX_FALSE	00	NULL	0	1	0
	11	AUX_FALSE	00	NULL	0	1	0
SYSCALL_LINTC_WRITE (4)	00	IRQ_RAISE	00	NULL	0	1	0
	01	AUX_FALSE	00	NULL	1	0	0
	10	AUX_FALSE	00	NULL	0	1	0
	11	AUX_FALSE	00	NULL	0	1	0
SYSCALL_LBUS_READ (5)	00	LBUS_RD_ACK	00	NULL	0	1	0
	01	AUX_FALSE	00	NULL	1	0	0
	10	AUX_FALSE	00	NULL	0	1	0
	11	AUX_FALSE	00	NULL	0	1	0
SYSCALL_LBUS_WRITE (6)	00	LBUS_WR_ACK	00	LBUS_WR_CE	0	1	0
	01	AUX_FALSE	00	NULL	1	0	0
	10	AUX_FALSE	00	NULL	0	1	0
	11	AUX_FALSE	00	NULL	0	1	0
SYSCALL_LBUS_READ_BURST (7)	00	ADDR_READY	00	ADDR_TRIGGER	0	1	0
	01	LBUS_RD_ACK_RLAST	01	NULL	0	1	0
	10	AUX_FALSE	00	NULL	1	0	0
	11	AUX_FALSE	00	NULL	0	1	0

(KL) Kernel-Level

🟨 Safeguard Correction

Table B.2: Simplified Microprogram Memory Representation (Part 2), back to fig. 4.2.9.

Microprogram Memory							
Syscall (KL)	**Progress**	**Input Sel.**	**NSF**	**Output Sel.**	**Valid**	**Block**	**Fault**
SYSCALL_LBUS_WRITE_BURST (8)	00	ADDR_READY	00	ADDR_TRIGGER	0	1	0
	01	LBUS_WR_ACK_WLAST	01	LBUS_WR_CE	0	1	0
	10	AUX_FALSE	00	NULL	1	0	0
	11	AUX_FALSE	00	NULL	0	1	0
SYSCALL_MUTEX_LOCK (9)	00	NOT_LOCKED_A	00	LBUS_RD_CE	0	1	0
	01	AUX_TRUE	10	LBUS_WR_CE	0	1	0
	10	LOCKED_B	00	LBUS_RD_CE	0	1	0
	11	AUX_FALSE	00	NULL	1	0	0
SYSCALL_MUTEX_TRY_LOCK (10)	00	NOT_LOCKED_A	00	LBUS_RD_CE	0	1	0
	01	AUX_TRUE	10	LBUS_WR_CE	0	1	0
	10	LOCKED_B	11	LBUS_RD_CE	0	1	0
	11	AUX_FALSE	00	NULL	1	0	0
SYSCALL_MUTEX_UNLOCK (11)	00	NOT_LOCKED_A	11	LBUS_RD_CE	0	1	0
	01	LOCKED_B	11	LBUS_RD_CE	0	1	0
	10	FREE	00	LBUS_WR_CE	0	1	0
	11	AUX_FALSE	00	NULL	1	0	0
SYSCALL_MBUS_READ (12)	00	M_AXI_ARREADY	00	M_AXI_ARVALID	0	1	0
	01	M_AXI_RVALID_RLAST	01	M_AXI_RREADY	0	1	0
	10	AUX_FALSE	00	NULL	1	0	0
	11	AUX_FALSE	00	NULL	0	1	0
SYSCALL_MBUS_WRITE (13)	00	M_AXI_AWREADY	00	M_AXI_AWVALID	0	1	0
	01	M_AXI_WREADY_WLAST	01	M_AXI_WVALID	0	1	0
	10	M_AXI_BVALID_TIMEOUT	10	M_AXI_BREADY_TRIGGER	0	1	0
	11	AUX_FALSE	00	NULL	1	0	0
SYSCALL_MBUS_READ_BURST (14)	00	M_AXI_ARREADY	00	M_AXI_ARVALID	0	1	0
	01	M_AXI_RVALID_RLAST	01	M_AXI_RREADY	0	1	0
	10	AUX_FALSE	00	NULL	1	0	0
	11	AUX_FALSE	00	NULL	0	1	0
SYSCALL_MBUS_WRITE_BURST (15)	00	M_AXI_AWREADY	00	M_AXI_AWVALID	0	1	0
	01	M_AXI_WREADY_WLAST	01	M_AXI_WVALID	0	1	0
	10	M_AXI_BVALID_TIMEOUT	10	M_AXI_BREADY_TRIGGER	0	1	0
	11	AUX_FALSE	00	NULL	1	0	0

(KL) Kernel-Level

🟨 Safeguard Correction

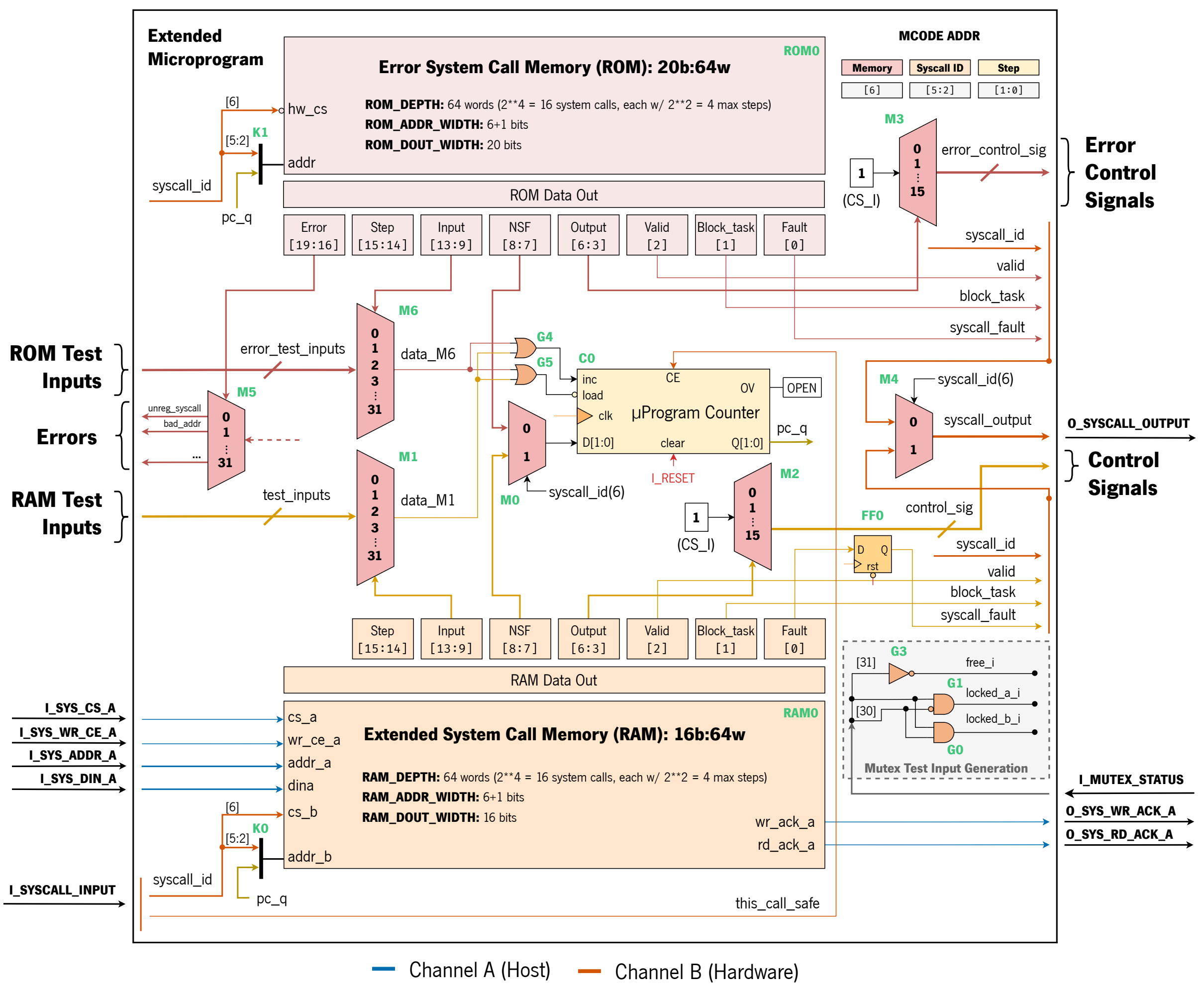


Figure B.3: Extended Microprogram for ROM-RAM Execution: RTL Design Internal Architecture (Simplified).

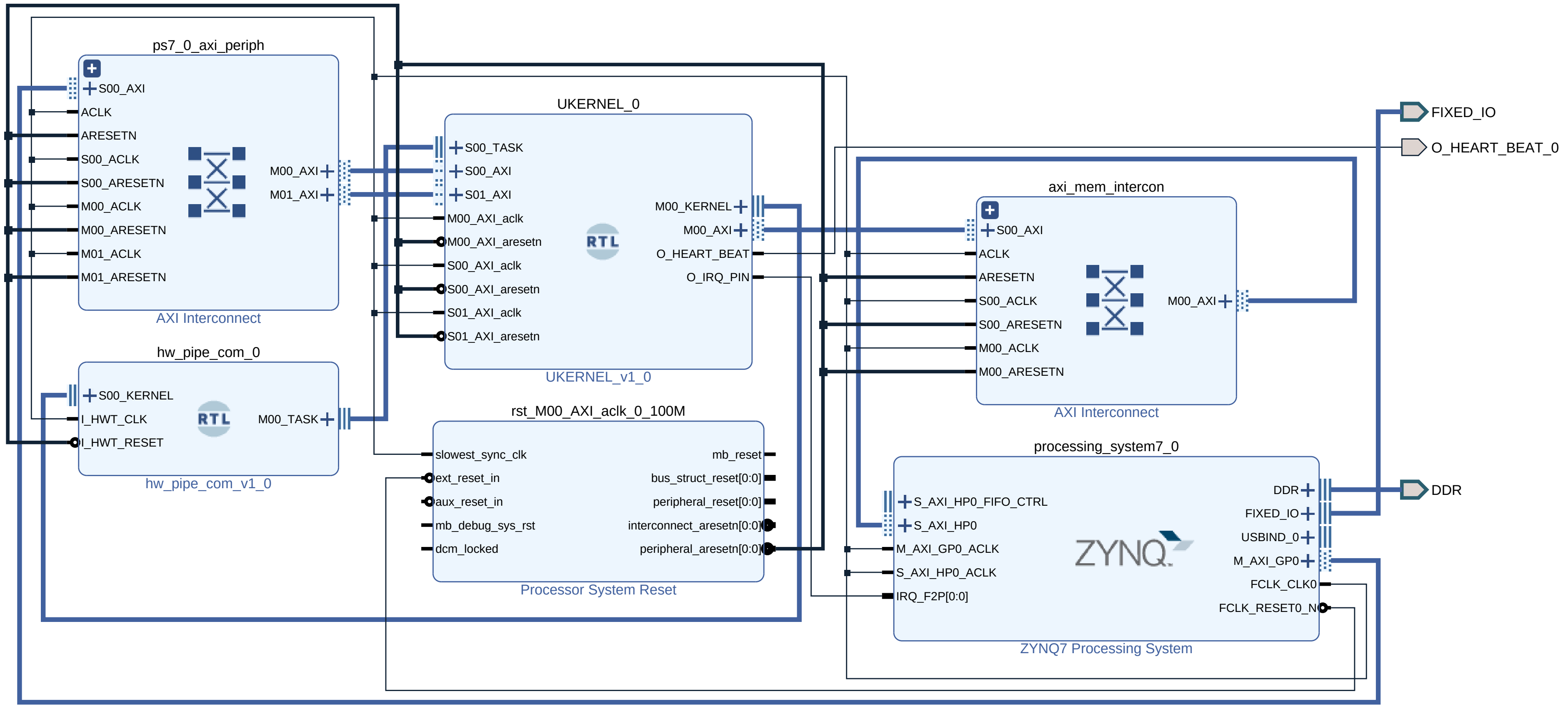


Figure B.4: Pipe Hardware Task Block Design using the Zybo Z7-10 platform.

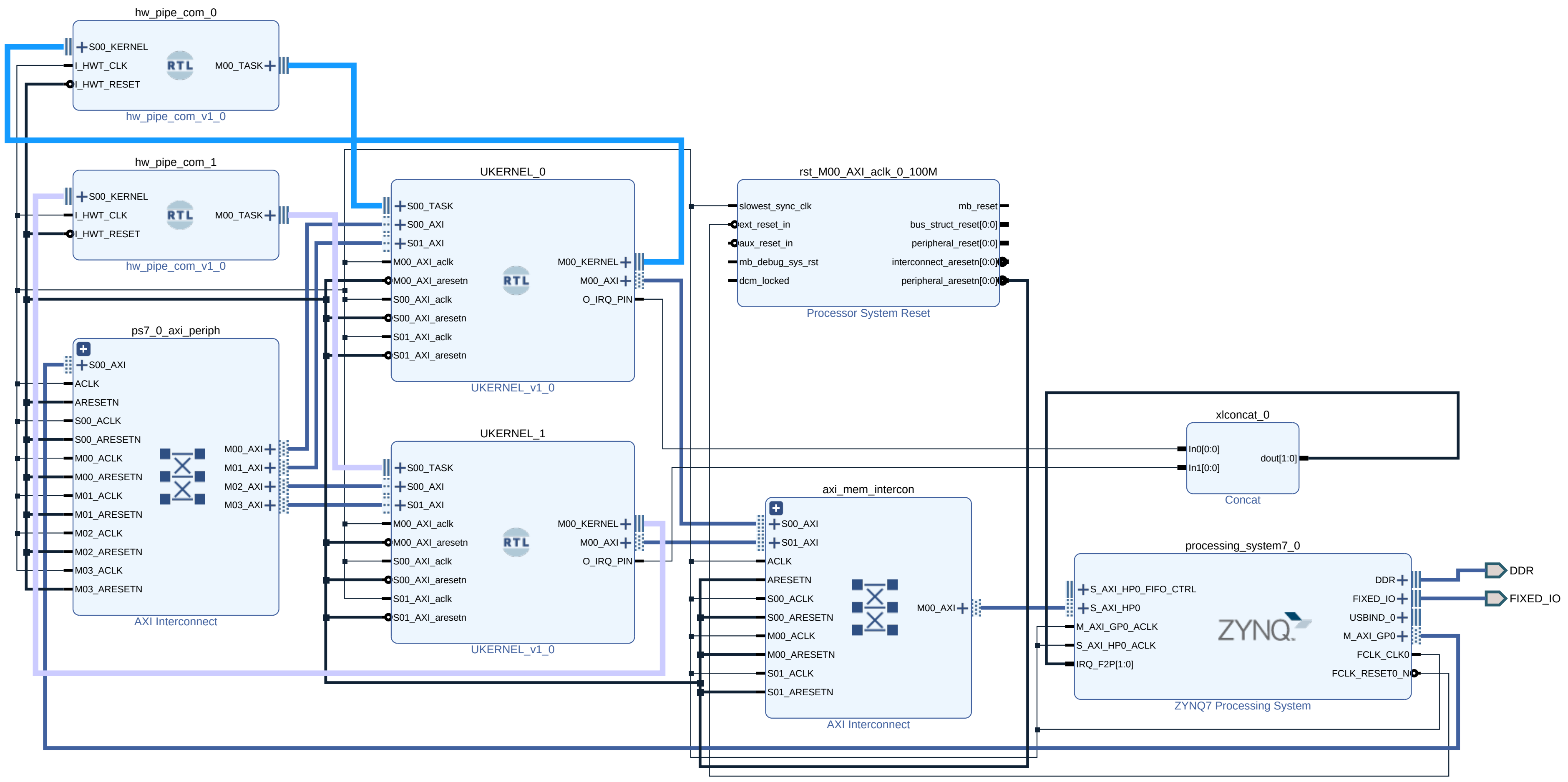


Figure B.5: Dual-Kernel Dual-Task Scheme Block Design using the Zybo Z7-10 platform.

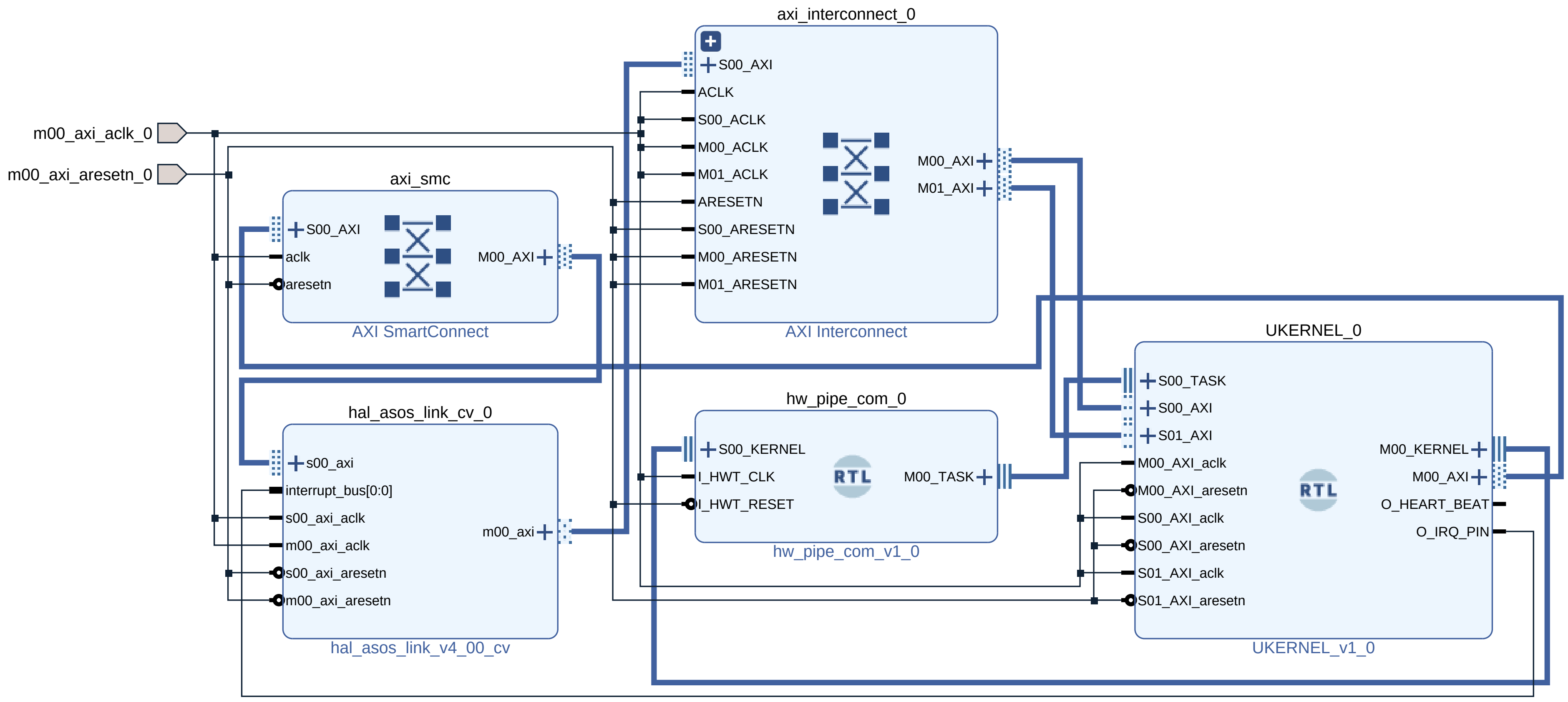
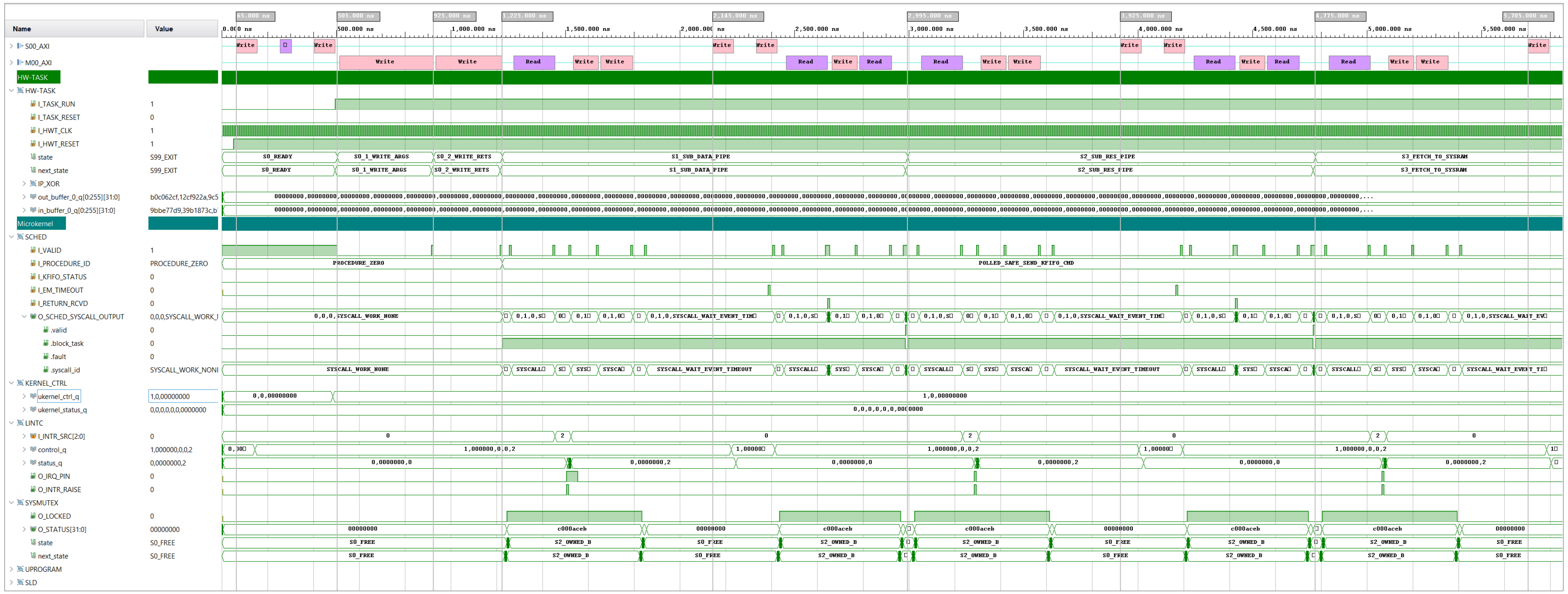


Figure B.6: HAL-ASOS Link IP Full Simulation: Block Design.



```
Name                              Value
> S00_AXI
> M00_AXI
HW-TASK
v HW-TASK
    I_TASK_RUN                    1
    I_TASK_RESET                  0
    I_HWT_CLK                     1
    I_HWT_RESET                   1
    state                         S99_EXIT
    next_state                    S99_EXIT
  > IP_XOR
  > out_buffer_0_q[0:255][31:0]   b0c062cf,12cf922a,9c5
  > in_buffer_0_q[0:255][31:0]    9bbe77d9,39b1873c,b
Microkernel
v SCHED
    I_VALID                       1
    I_PROCEDURE_ID                PROCEDURE_ZERO
    I_KFIFO_STATUS                0
    I_EM_TIMEOUT                  0
    I_RETURN_RCVD                 0
  v O_SCHED_SYSCALL_OUTPUT        0,0,0,SYSCALL_WORK_I
      .valid                      0
      .block_task                 0
      .fault                      0
      .syscall_id                 SYSCALL_WORK_NONI
v KERNEL_CTRL
  > ukernel_ctrl_q                1,0,00000000
  > ukernel_status_q              0,0,0,0,0,0,0000000
v LINTC
  > I_INTR_SRC[2:0]               0
  > control_q                     1,000000,0,0,2
  > status_q                      0,0000000,2
    O_IRQ_PIN                     0
    O_INTR_RAISE                  0
v SYSMUTEX
    O_LOCKED                      0
  > O_STATUS[31:0]                00000000
    state                         S0_FREE
    next_state                    S0_FREE
> UPROGRAM
> SLD
```

Figure B.7: Pipe Communication Hardware Task (SYSRAM w/ *XOR* Encryption): Overview (First Half Zoomed).

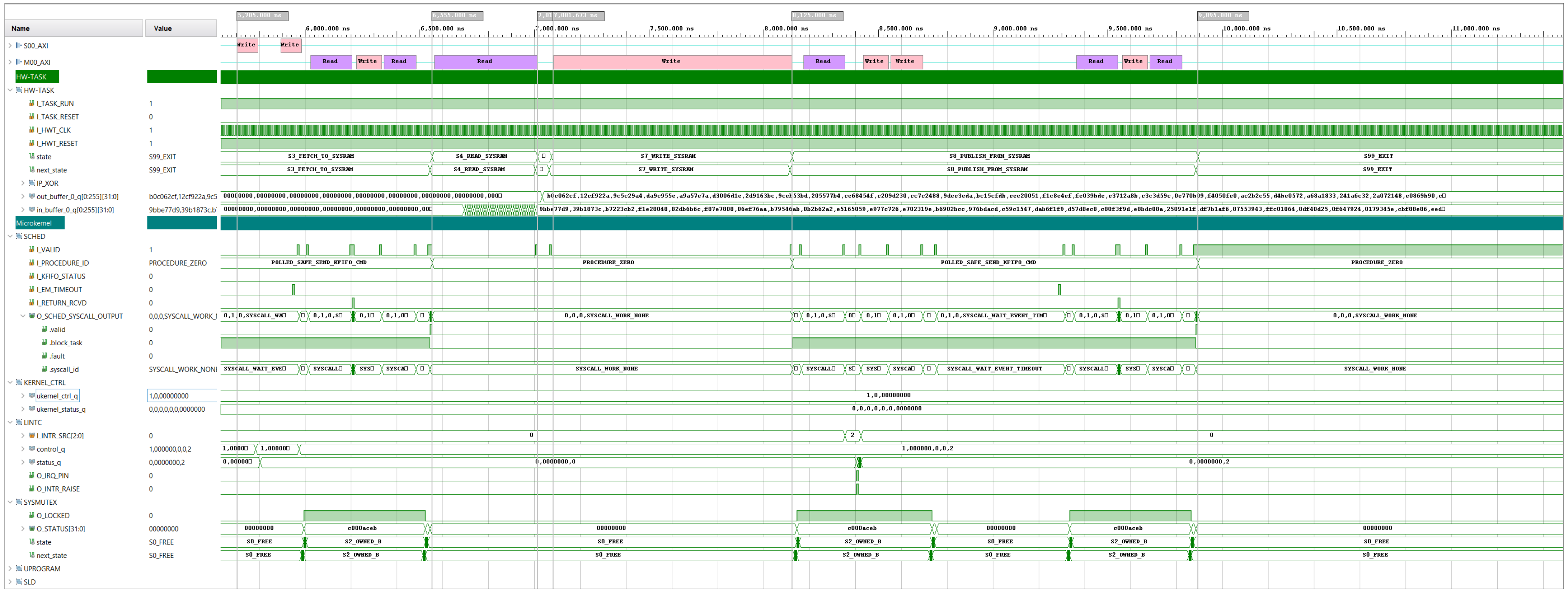

```
Name                          Value
> S00_AXI
> M00_AXI
HW-TASK
v HW-TASK
    I_TASK_RUN                1
    I_TASK_RESET              0
    I_HWT_CLK                 1
    I_HWT_RESET               1
    state                     S99_EXIT
    next_state                S99_EXIT
  > IP_XOR
  > out_buffer_0_q[0:255][31:0]   b0c062cf,12cf922a,9c5
  > in_buffer_0_q[0:255][31:0]    9bbe77d9,39b1873c,b
Microkernel
v SCHED
    I_VALID                   1
    I_PROCEDURE_ID            PROCEDURE_ZERO
    I_KFIFO_STATUS            0
    I_EM_TIMEOUT              0
    I_RETURN_RCVD             0
  v O_SCHED_SYSCALL_OUTPUT    0,0,0,SYSCALL_WORK_I
      .valid                  0
      .block_task             0
      .fault                  0
      .syscall_id             SYSCALL_WORK_NONI
v KERNEL_CTRL
  > ukernel_ctrl_q            1,0,00000000
  > ukernel_status_q          0,0,0,0,0,0,0000000
v LINTC
  > I_INTR_SRC[2:0]           0
  > control_q                 1,000000,0,0,2
  > status_q                  0,0000000,2
    O_IRQ_PIN                 0
    O_INTR_RAISE              0
v SYSMUTEX
    O_LOCKED                  0
  > O_STATUS[31:0]            00000000
    state                     S0_FREE
    next_state                S0_FREE
> UPROGRAM
> SLD
```

Figure B.8: Pipe Communication Hardware Task (SYSRAM w/ *XOR* Encryption): Overview (Second Half Zoomed).

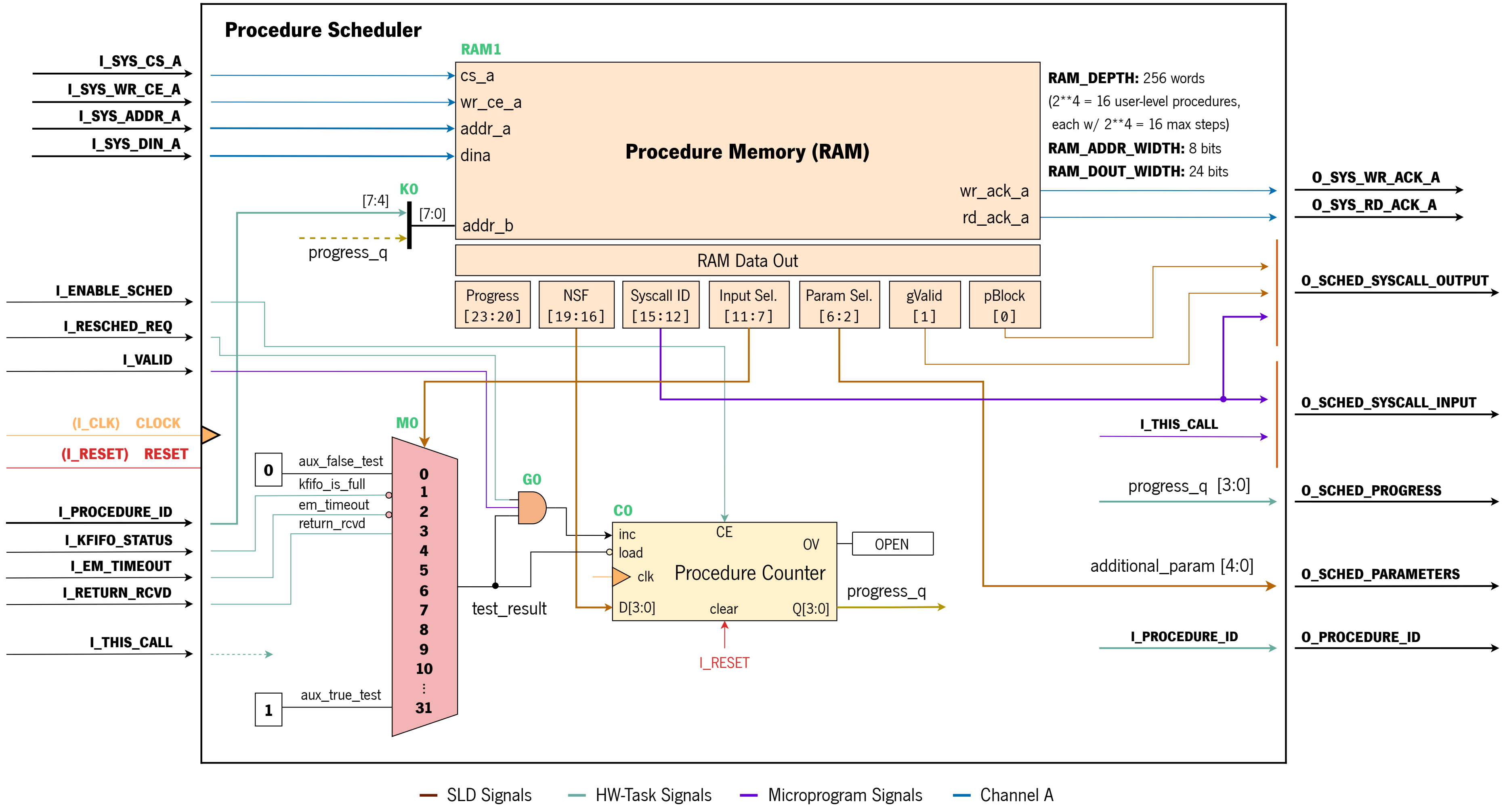


Figure B.9: Updatable Scheduler RTL Design Internal Architecture.

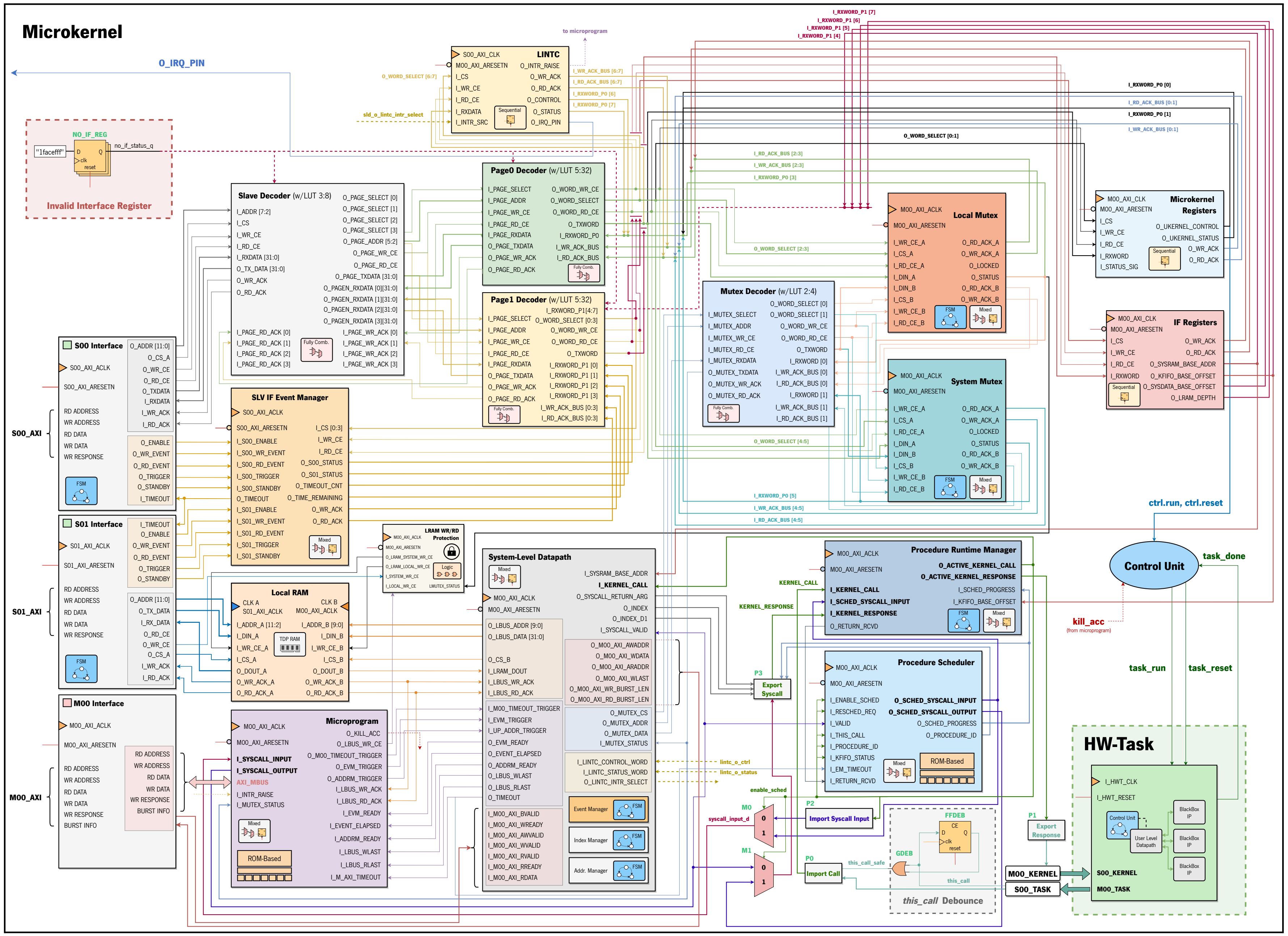


Figure B.10: (Native) Microkernel Internal Architecture Overview, back to fig. 4.1.1.

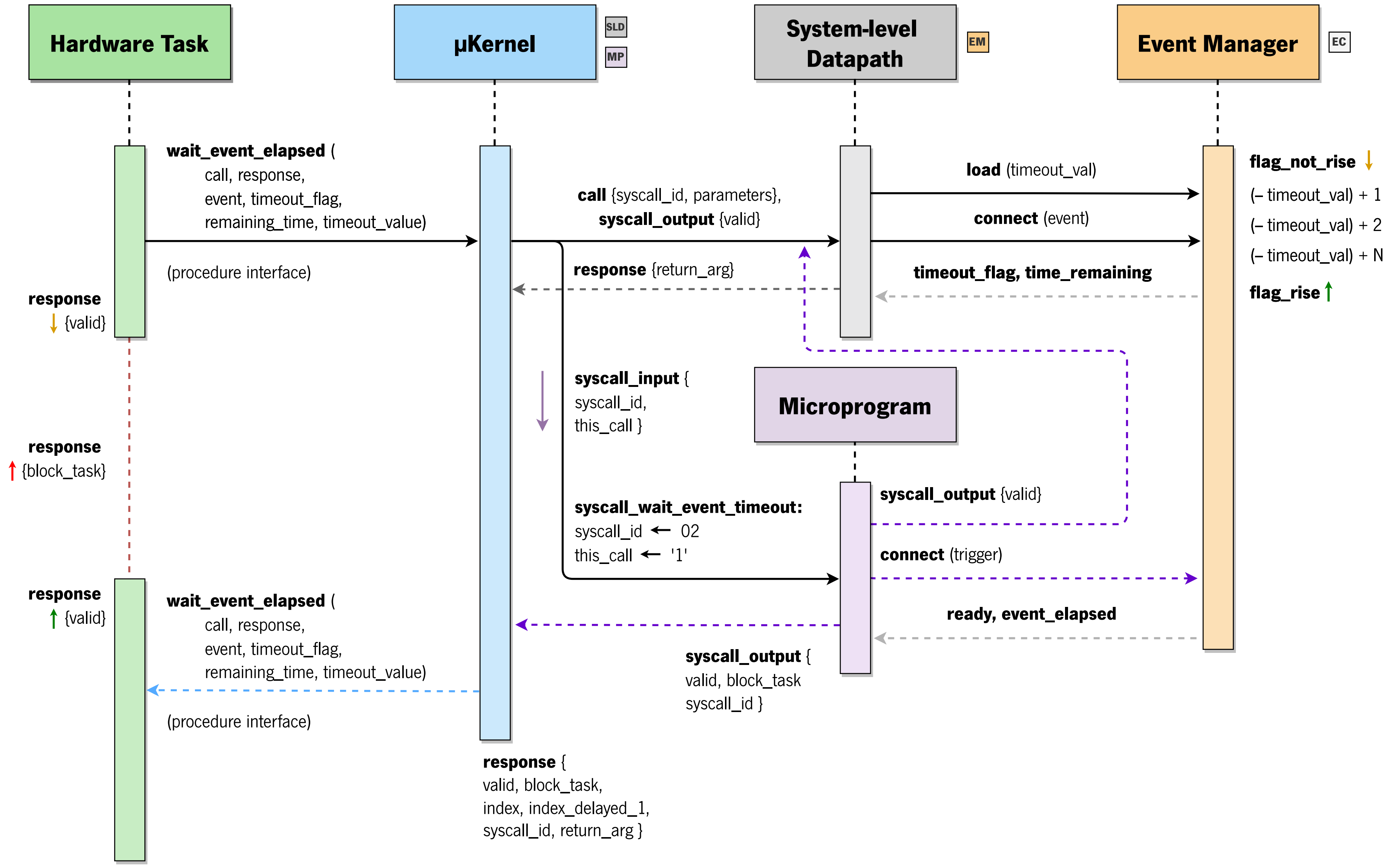


Figure B.11: User-Level Procedure – Wait Event Elapsed: Sequence Diagram, back to fig. 4.2.19.

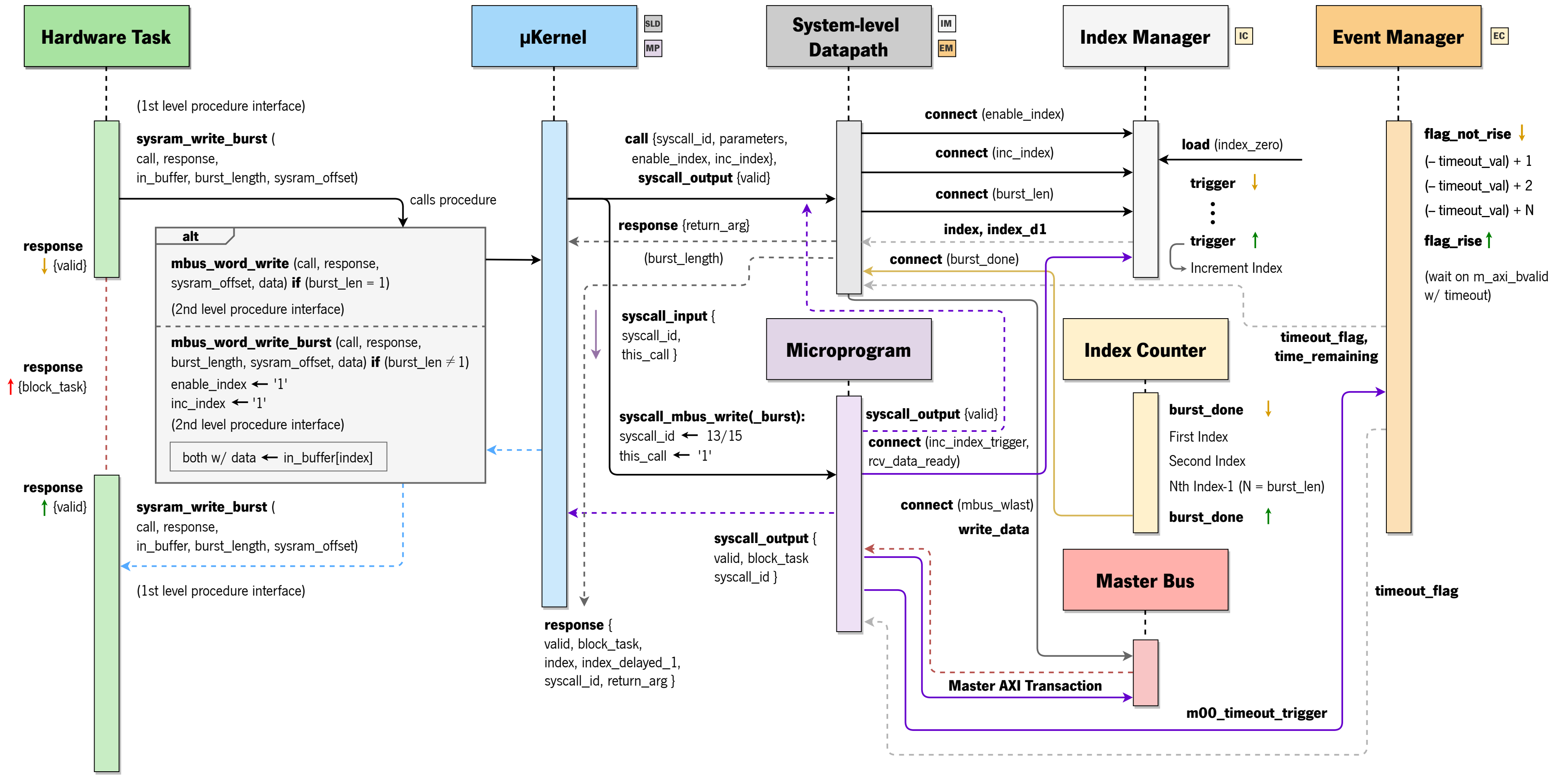


Figure B.12: User-Level Procedure – Unsafe SYSRAM Write Burst: Sequence Diagram (Simplified), back to fig. 4.2.21.

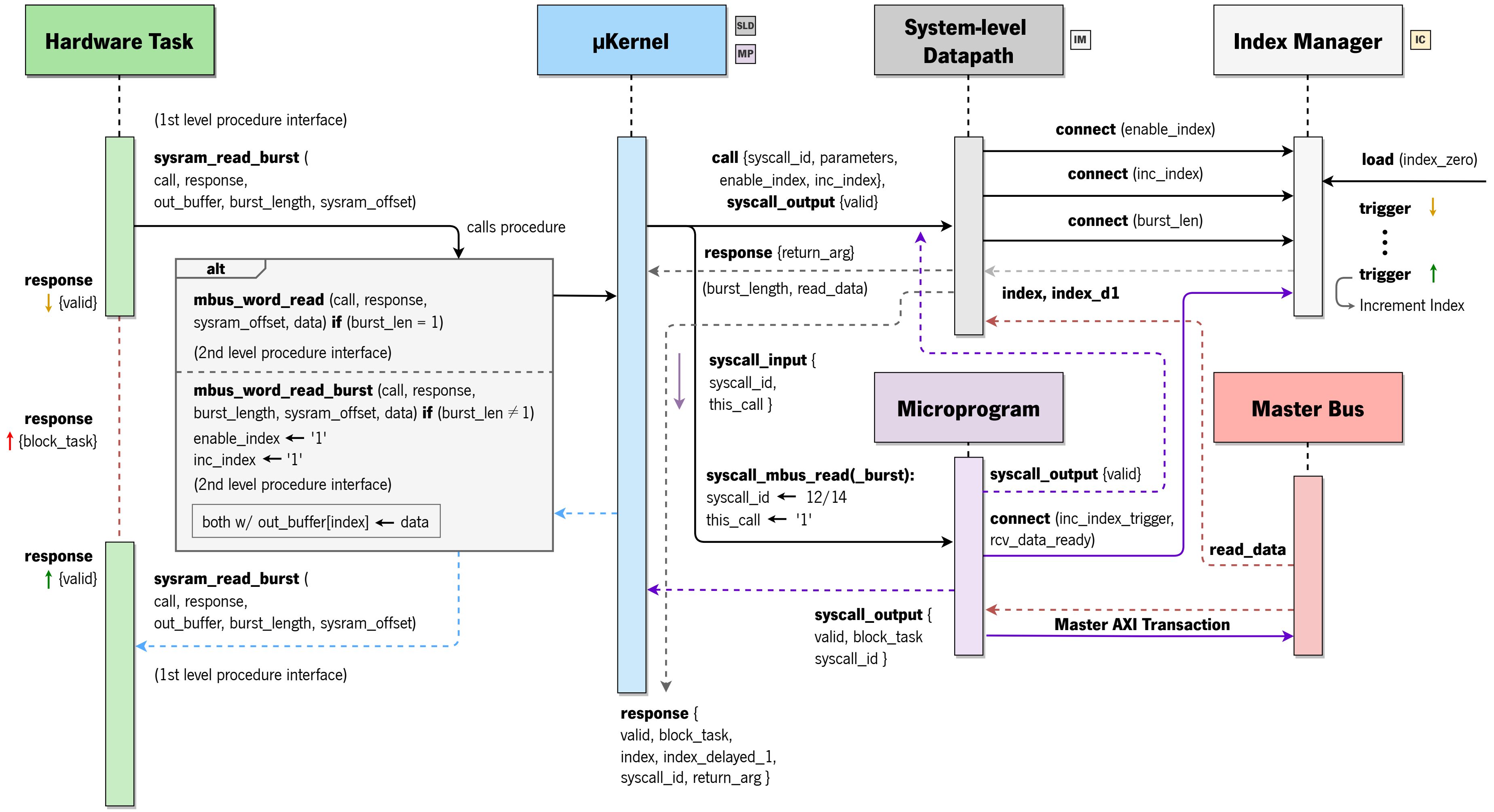

Hardware Task
µKernel
SLD
MP
System-level Datapath
IM
Index Manager
IC
(1st level procedure interface)
sysram_read_burst (call, response, out_buffer, burst_length, sysram_offset)
calls procedure
response ↓ {valid}
alt
mbus_word_read (call, response, sysram_offset, data) if (burst_len = 1)
(2nd level procedure interface)
mbus_word_read_burst (call, response, burst_length, sysram_offset, data) if (burst_len ≠ 1)
enable_index ← '1'
inc_index ← '1'
(2nd level procedure interface)
both w/ out_buffer[index] ← data
response ↑ {block_task}
response ↑ {valid}
sysram_read_burst (call, response, out_buffer, burst_length, sysram_offset)
(1st level procedure interface)
call {syscall_id, parameters, enable_index, inc_index}, syscall_output {valid}
response {return_arg}
(burst_length, read_data)
syscall_input { syscall_id, this_call }
syscall_mbus_read(_burst): syscall_id ← 12/14 this_call ← '1'
syscall_output { valid, block_task syscall_id }
response { valid, block_task, index, index_delayed_1, syscall_id, return_arg }
connect (enable_index)
connect (inc_index)
connect (burst_len)
index, index_d1
load (index_zero)
trigger ↓
trigger ↑
Increment Index
Microprogram
Master Bus
syscall_output {valid}
connect (inc_index_trigger, rcv_data_ready)
read_data
Master AXI Transaction

Figure B.13: User-Level Procedure – Unsafe SYSRAM Read Burst: Sequence Diagram (Simplified), back to fig. 4.2.21.

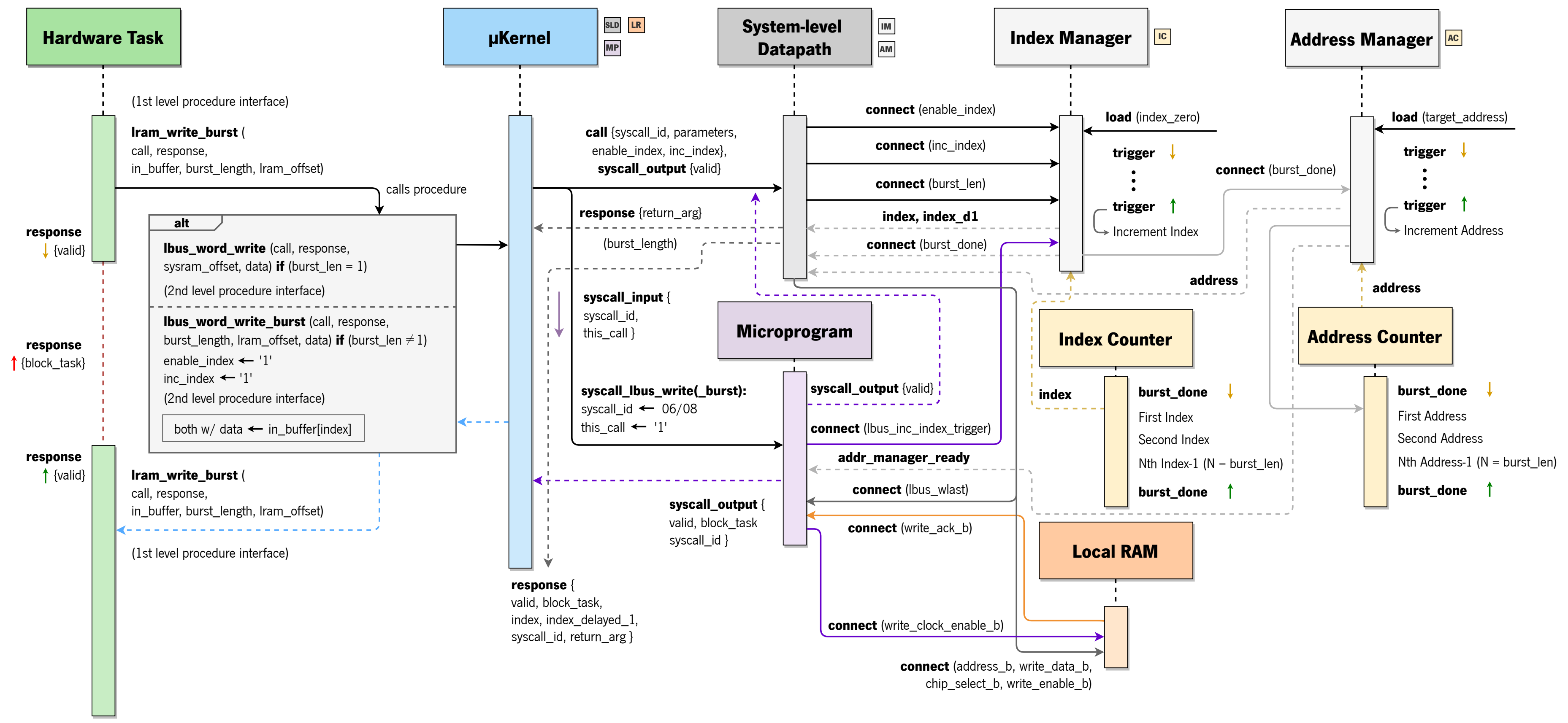


Figure B.14: User-Level Procedure – Unsafe LRAM Write Burst: Sequence Diagram (In-Depth), back to fig. 4.2.23.

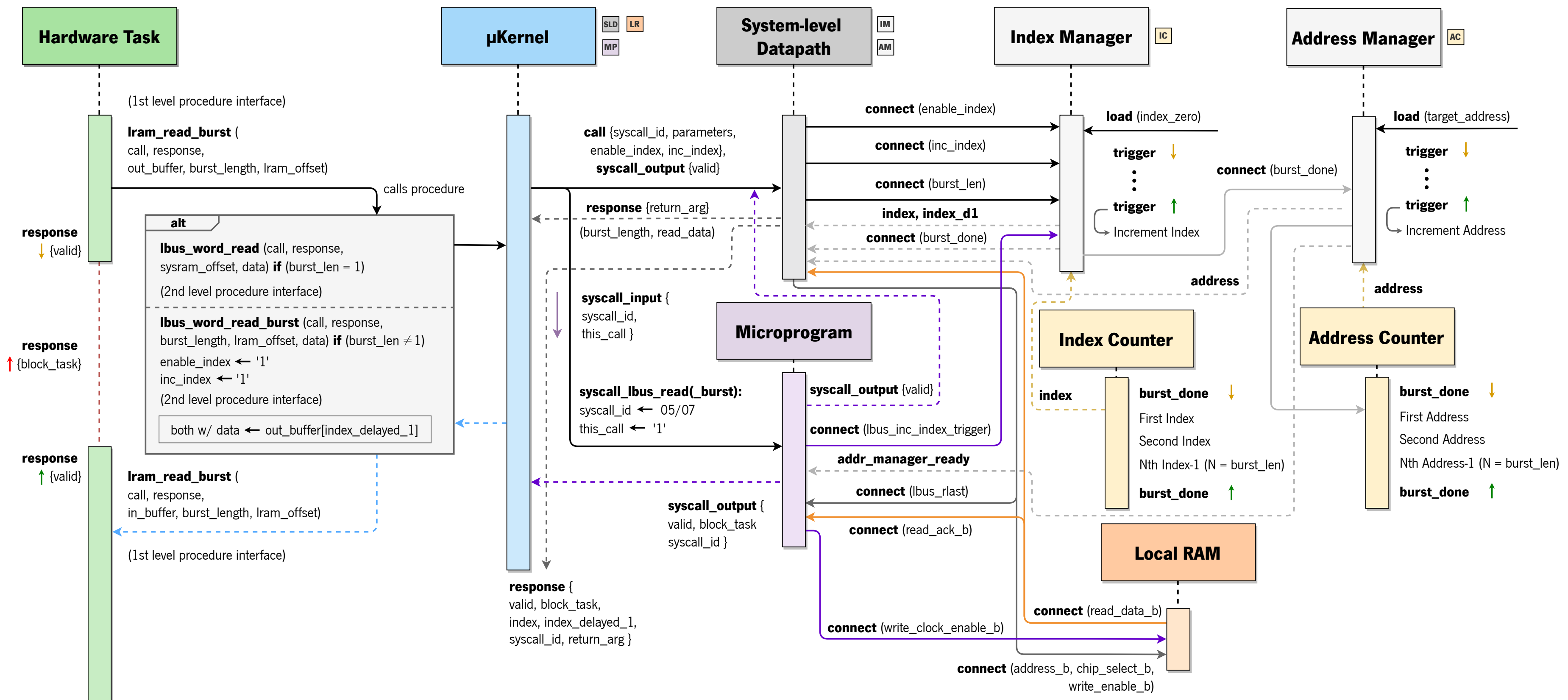


Figure B.15: User-Level Procedure – Unsafe LRAM Read Burst: Sequence Diagram (In-Depth), back to fig. 4.2.23.