Masterthesis

Improving Simulation Performance and Architecture in the SystemC Model of the ParaNut Processor

Author: Felix Wagner

Bahnhofstr. 14
86150 Augsburg
mail@wagner-felix.de

Matrikel Number: 2112927

Course: Master of Applied Research in Computer Science

Supervising Professor: Prof. Dr. Gundolf Kiefer

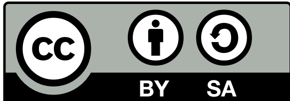
Date: August 9, 2023

Technical University of Applied Sciences
An der Hochschule 1, 86161 Augsburg, Germany
Tel. +49 821 55 86-0
www.hs-augsburg.de

Fakulty for Computer Science
Tel. +49 821 55 86-3450

Copyright © 2023 Felix Wagner

This work is licensed under a Creative Commons "Attribution-ShareAlike 4.0 International" license.



Contents

List of Acronyms & Abbreviations **III**

Abstract **IV**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Purpose of this Thesis . . . 1
1.3 Structure of this Document . . . 2
1.4 Terms and Definitions . . . 2

2 Basics **4**
2.1 The ParaNut Processor . . . 4
2.2 Processor Architecture . . . 4
2.2.1 CPU . . . 5
2.2.2 Memory Unit . . . 7
2.2.3 Interconnect . . . 9
2.2.4 UART . . . 9
2.2.5 GPIO . . . 9
2.2.6 MTimer . . . 9
2.3 Toolchain . . . 10

3 ParaNut Hard- and Software Development **10**
3.1 Eliminating VHDL Hardware Sources . . . 10
3.2 Memory Unit in SystemC . . . 11
3.3 ParaNut Single Core . . . 13
3.4 GCC Compatibility . . . 13

4 Automated Building and Testing **14**
4.1 The Build System . . . 14
4.2 The ParaNut Build System . . . 15
4.3 Designing a New Build System . . . 17
4.3.1 Directory Base . . . 17
4.3.2 Libraries - hal-base.mk . . . 20
4.3.3 Applications - application-base.mk . . . 21
4.3.4 Hardware - sysc-base.mk . . . 21
4.3.5 ParaNut Systems . . . 23
4.3.6 Continuous Integration/Continuous Delivery . . . 24
4.4 Project Architecture . . . 24
4.5 Installation . . . 26
4.6 Synthesis with SystemC . . . 27

5 Evaluation **28**
5.1 Build System 28
5.2 Processor Optimizations 29
5.2.1 Memory Unit 29
5.2.2 Single Core 30

6 Conclusion **32**

Appendices **35**
A EXU Block Diagram 35
B EXU State Machine 36

List of Acronyms & Abbreviations

ALU Arithmetic Logic Unit.

CePU Central Processing Unit.

CI/CD Continuous Integration / Continuous Delivery.

CoPU Co Processing Unit.

CSR Control and Status Register.

EXU Execution Unit.

FPGA Field Programmable Gate Array.

GCC GNU Compiler Collection.

HAL Hardware Abstraction Layer.

HLS High-Level Synthesis.

ICSC Intel Compiler for SystemC.

IFU Instruction Fetch Unit.

ISA Instruction Set Architecture.

LSU Load/Store Unit.

MMU Memory Management Unit.

SIMD Single Instruction Multiple Data.

SMT Simultaneous Multi Threading.

TLS Thread Local Storage.

UART Universal Asynchronous Receiver / Transmitter.

VHDL Very High Speed Integrated Circuit Hardware Description Language.

Abstract

The ParaNut project provides a scalable, and fully RISC-V compatible soft core processor for FPGA based systems. Having been under continuous development for more than a decade, the project has grown significantly in functionality and complexity. The general aim of the master thesis presented in this document is to research and implement methodologies to improve on the existing architecture and technology stack, to enhance maintainability and development workflow. This is achieved through restructuring the project directory and providing a new, centralized build automation based on GNU Make, the SystemC library and the Intel Compiler for SystemC, including verbose targets for building, installing and testing. This work documents the structure of the processor's hardware modules and the dependencies between them. Additionally, it describes the development of a synthesizable memory unit written in SystemC, developed as part of the process of unifying simulation and synthesis sources by providing all modules in synthesizable SystemC.

1 Introduction

1.1 Motivation

The ParaNut project provides a configurable, scalable, and fully RISC-V compatible soft core processor for FPGAs (Field Programmable Gate Arrays) [1]. It is being developed by the *Efficient Embedded Systems Group* at the *Technical University of Applied Sciences Augsburg* and is currently in use in research and education. The Open Source project was created by Prof. Dr. Gundolf Kiefer in 2010 and has since been advanced by students at the university. Most of the ParaNut's hardware is written in SystemC. A complete SystemC model of the hardware can be used to simulate the hardware behavior on a PC. When synthesizing for FPGA systems, some components are described in VHDL (Very High Speed Integrated Circuit Hardware Description Language).

This mix of multiple hardware description languages/methods led to a divergence between simulated hardware and synthesized hardware. A major motivation for transitioning to a SystemC only hardware description is a bug present in the synthesized hardware, that is not reproducible in the simulator and thus hard to analyze and resolve.

Being in development for more than thirteen years, the project has undergone major growth. Most development was focussed on improving or adding single components to the project which led to a diverse and distributed build system. This resulted in diverging coding styles and a lack of detailed documentation for many of the more complex modules included in the project. Many of the older module's test benches have not been updated to include updated features and are partially broken.

In the summer of 2022 developers started transitioning the hardware build process from the Xilinx Vivado HLS (High-Level Synthesis) to the Open Source tool ICSC (Intel Compiler for SystemC). Accordingly, most of the previous build process are obsolete, and the SystemC code must be updated to the tool's requirements. This presents an opportunity to refactor the code to current SystemC and project conventions and document the current hardware architecture and interfaces.

1.2 Purpose of this Thesis

The work presented in this document aims to improve the development and deployment process of the ParaNut project while simultaneously enabling developers to provide specialized hardware modules for improved simulation performance and an inherently modular hardware build procedure.

These improvements are achieved by designing a more centralized build system and restructuring the project's folder architecture to accommodate the components that have been developed so far, as well as components that will be developed in the future.

Whilst describing the design process and concepts behind the changes to the build system and

repository, this work also provides documentation for future developers to reference when updating and extending the system.

Besides the structural and organizational changes briefly described so far, a number of adaptations and improvements in the ParaNut hardware itself have been implemented. These changes are mainly concerned with providing a fully synthesizable memory management unit in SystemC as well as optimizing the ParaNut configurability to provide a more versatile and efficient hardware platform in simulation and on hardware.

1.3 Structure of this Document

This document presents its contents in four chapters. The first, Section 2 "Basics" describes the foundational concepts of the ParaNut processor, its unique attributes, features and structure. The second chapter, Section 3 "ParaNut Hard- and Software Development", describes work conducted on the hardware architecture and functionality of the processor itself, optimizations and refactoring done in order to improve its performance and synthesizability. The third chapter, Section 4 "Automated Building and Testing", explains the process and result of restructuring the project's build system. A fourth chapter covers validating and analyzing the results produced.

1.4 Terms and Definitions

Table 1 lists terms used throughout this document.

Term	Definition
Version 1.1 **v1.1** **Previous Version**	The state of the project repository before the changes described in this document were made.
Version 1.6 **v1.6** **Current Version**	The state of the project repository after the changes described in this document were made.
Developer	Someone who develops the ParaNut project, i.e. is modifying the source files inside the ParaNut repository.
User	Someone who uses a ParaNut project installation, i.e. develops soft- and hardware for a custom ParaNut System without altering the project's sources files
ParaNut System	A specific hardware configuration of a ParaNut Processor and libraries to utilize the hardware in software. May include custom hardware components developed specifically for this system.
Source Tree	The files and their organization, as provided by the ParaNut git repository.
Installation Tree	The files and their organization of a ParaNut installation (created through "make install").
System Tree	The files and their organization of a ParaNut System. This may either be located inside the source tree or in a directory defined by the user.
Project Tree	The files and their organization of ParaNut Project. This can be located anywhere on the user's system and is created using the pn-newproject binary

Table 1: Definition of Terms

A note on the availability of the ParaNut Project

The repository is accessible either on the public GitHub repository: `https://github.com/hsa-ees/paranut` or internally on the university's GitLab server: `https://ti-build.informatik.hs-augsburg.de:8443/paranut_developers/paranut` Version 1.1 is only available on the internal server. Only Version 1.0 and 1.5 have been made public. v1.0 lacks major development included in Version 1.1, like a Memory Management Unit and FreeRTOS support. Version 1.6 adds the synthesizable memory unit, bug fixes and single core option. Git Hashes for reference:

GitLab (internal):	v1.6 (30cfbfc6),	v1.5 (4665bea0),	v1.1 (2a094e4f),	v1.0 (d3eb3c67)
GitHub:	v1.6 N/A,	v1.5 (86a072a),	v1.1 N/A,	v1.0 (65f1057)

2 Basics

2.1 The ParaNut Processor

The ParaNut hard- and software unique among currently available RISC-V soft core processors due to its special concept of parallelization. A detailed description of the concept and design approach used by the developers is published under [4].

Developers and users are able to switch between SMT (Simultaneous Multi Threading) and SIMD (Single Instruction Multiple Data) parallelization during runtime, code sections describing either form of parallelization do not need specific syntax or compiler but may consist of standard RISC-V instructions.

In order to access these and other unique hardware capabilities, a software library called *libparanut* is provided. This library can be used to enable and disable both parallelization modes, to synchronize memory access and to read and write custom CSRs (Control and Status Registers).

2.2 Processor Architecture

The ParaNut hardware is built around the concept of a single CePU (Central Processing Unit) and multiple CoPUs (Co Processing Units). The CePU in this design provides a central, controlling unit. It is composed of an ALU (Arithmetic Logic Unit), a register file, control logic for fetching data and instructions from the central memory bus and interrupt and exception logic. In order to facilitate fast and prioritized memory access, the CePU includes as two ports for communicating with the central memory unit. One read-only port for fetching instructions and a second read/write port for data access. As the design concept of the ParaNut aims at using as little hardware and energy resources as possible, CoPUs are designed to be significantly smaller and less complex than the central CePU. A ParaNut System can be configured to contain one or more CoPUs, each of which may either be of *Capability Level 1* or *Capability Level 2*. The *Capability Levels* differ in their respective hardware size and the parallelization abilities they offer.

Capability Level 1 can only be used for SIMD parallelization. A CoPU of this mode does not feature a dedicated IFU (Instruction Fetch Unit). Instead, when a *Capability Level 1* CoPU is active, it shares the CePU's IFU, effectively running the respective instructions in a vectorized manner. As cores of this *Capability Level* do not need to access the system memory for fetching instructions, they only provide a read/write port for data access and omit the read port for instructions featured by the CePU.

Capability Level 2 CoPUs offer both, support for SIMD and SMT code execution. Compared to a CePU, they lack exception and interrupt hardware, they do however provide a dedicated IFU, enabling them to execute instructions largely independent of the CePU. The hardware responsible for fetching and providing the instructions in a *Capability Level 2* CoPU can be turned off, in which case the CoPU is connected to the IFU of the CePU, resulting in the same

hardware behavior exhibited by a *Capability Level 1* CoPU.

Capability Level 3 describes a full CPU core, it is used to control cores of lesser *Capability Level*. Accordingly, it's IFU can not be turned off and is used to provide instructions to cores running in SIMD mode. It contains full interrupt and exception logic and handles exceptions caused by a CoPU. The CoPU is halted until the incidence is resolved.

Figure 1 illustrates a ParaNut System with two *Capability Level 1* and one *Capability Level 2* CoPUs.

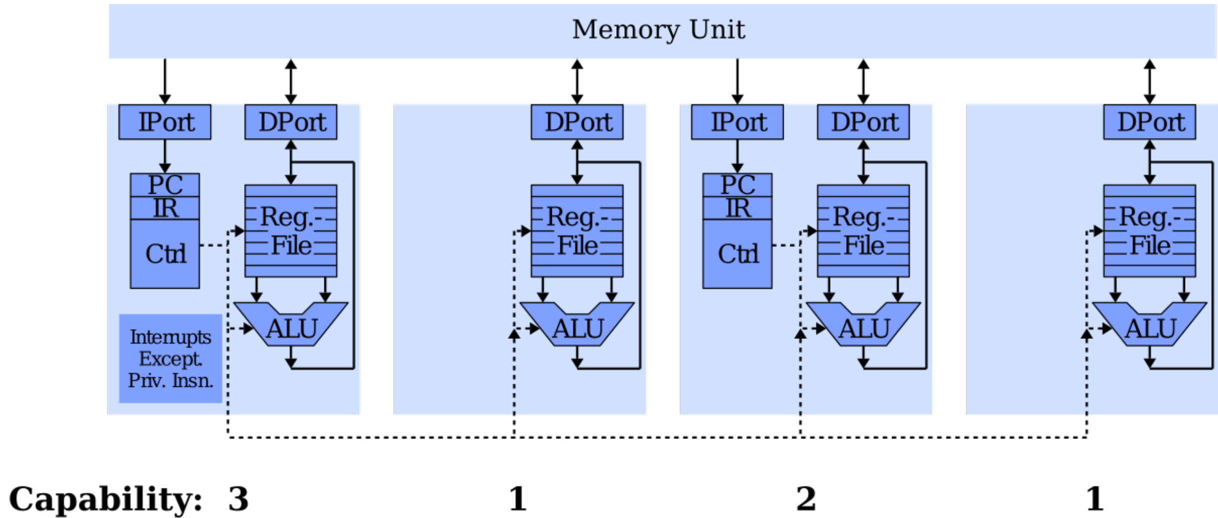


Figure 1: ParaNut Hardware Capability Levels [4]

The following sections describe logical components of the ParaNut processor. As a scalable and highly configurable processor it is key, that some of its components may be activated or deactivated in any given implementation. The following description can only encompass components that are currently present in the ParaNut project, the list is likely to be extended in the future.

Fig. 2 illustrates a top level module representation of a ParaNut System. Each block represents a processor module. Dotted lines indicate, that the modules may be configured to be excluded or included in a given configuration. Blocks in orange are partially written in VHDL as of version 1.1. Red blocks indicate modules that contain implementations in which simulation and hardware differ significantly.

2.2.1 CPU

The CPU module's main task is executing instructions from memory. It contains logic to provide an arithmetic unit as well as a control unit. Any ParaNut System must contain one core of *Capability Level 3*. All others can be configured to be either of *Capability Level 2* or *Capability Level 1*. Thus, this module is present in three expansion levels. Additionally, the arithmetic logic unit of each core may be configured to provide the ISA (Instruction Set Architecture) extensions M and/or A. Note that this is a global configuration and effects all cores.

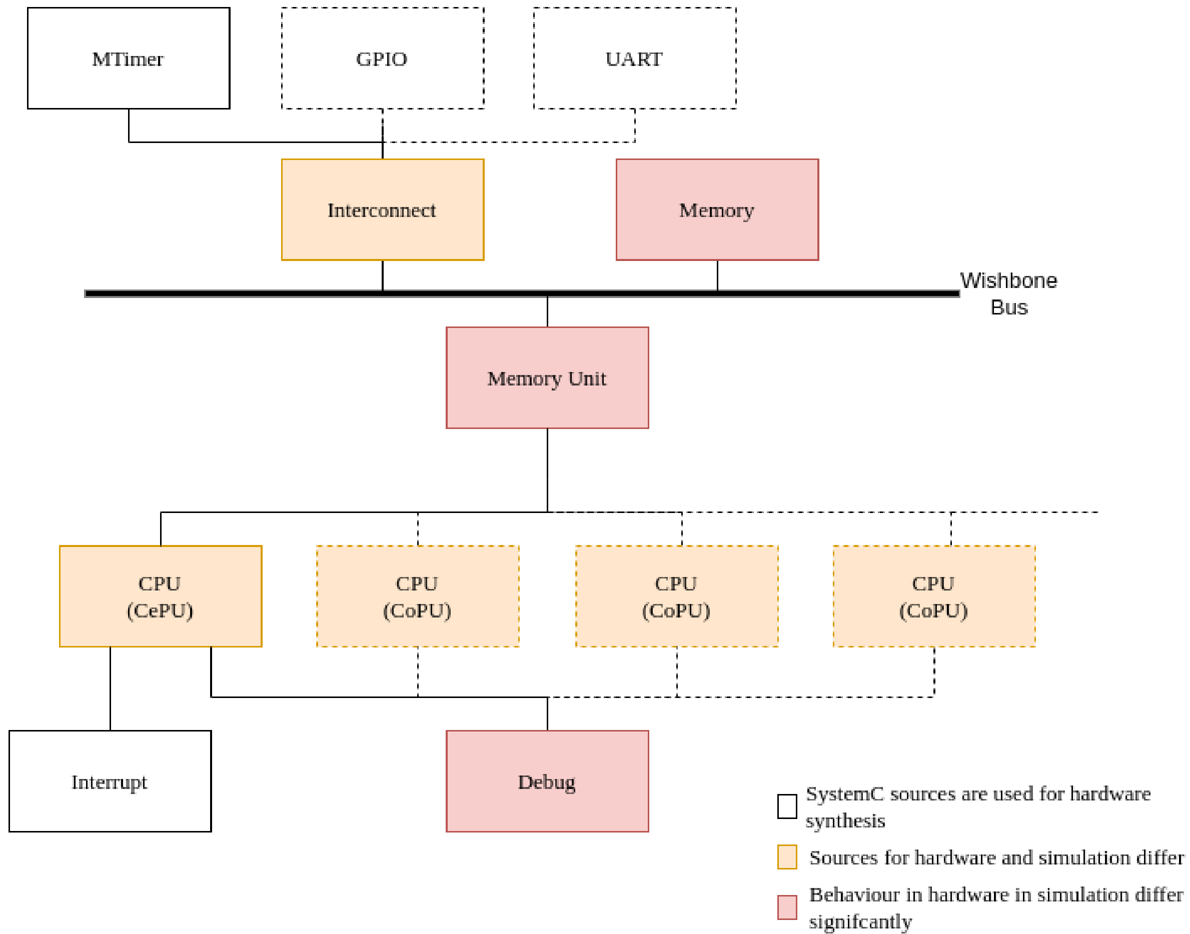


Figure 2: ParaNut Module Architecture

A core is composed of multiple submodules. Figure 1 depicts a schematic representation of CPU's of all three capability levels. Fig. 3 shows a more detailed documentation of the SystemC modules present under the CPU module. The oval "CPU" node represents all the top level module's functions and their connections. A more detailed analysis of a CPU's structure can be found in Section A[1].

The CPUs access the Wishbone Bus through the two submodules LSU (Load/Store Unit) and IFU. The access itself is controlled externally by the Memory Unit. Both, the Memory Unit and the Interconnect module may cause exceptions and have appropriate signals connected to the CPU. Additionally, the CePU can send and receive specific signals to and from the CoPUs. These signals are mainly concerned with exception and interrupt handling, as well as controlling the CoPU's execution mode.

Every CPU contains a set of CSR. Depending on the *Capability Level* of a given CPU, the exact number and set of CSRs differs.

[1]The original file contains meta information to each directed connection between the submodules about the precise signals that are shared. The diagram was created to improve the state of documentation on the central CPU component, the EXU (Execution Unit).

The *MExtension* module, used to add support for the RISC-V "M" Standard Extension for Integer Multiplication and Division, can be activated in a system configuration and is otherwise not present.

There are two state machines implemented in the CPU, one for arithmetic operations and one for controlling the CPU execution.

The EXU state machine is believed to be a possible culprit for the diverging behavior of hardware and software. The state machine too has been analyzed and documented in the form of a diagram (Section B).

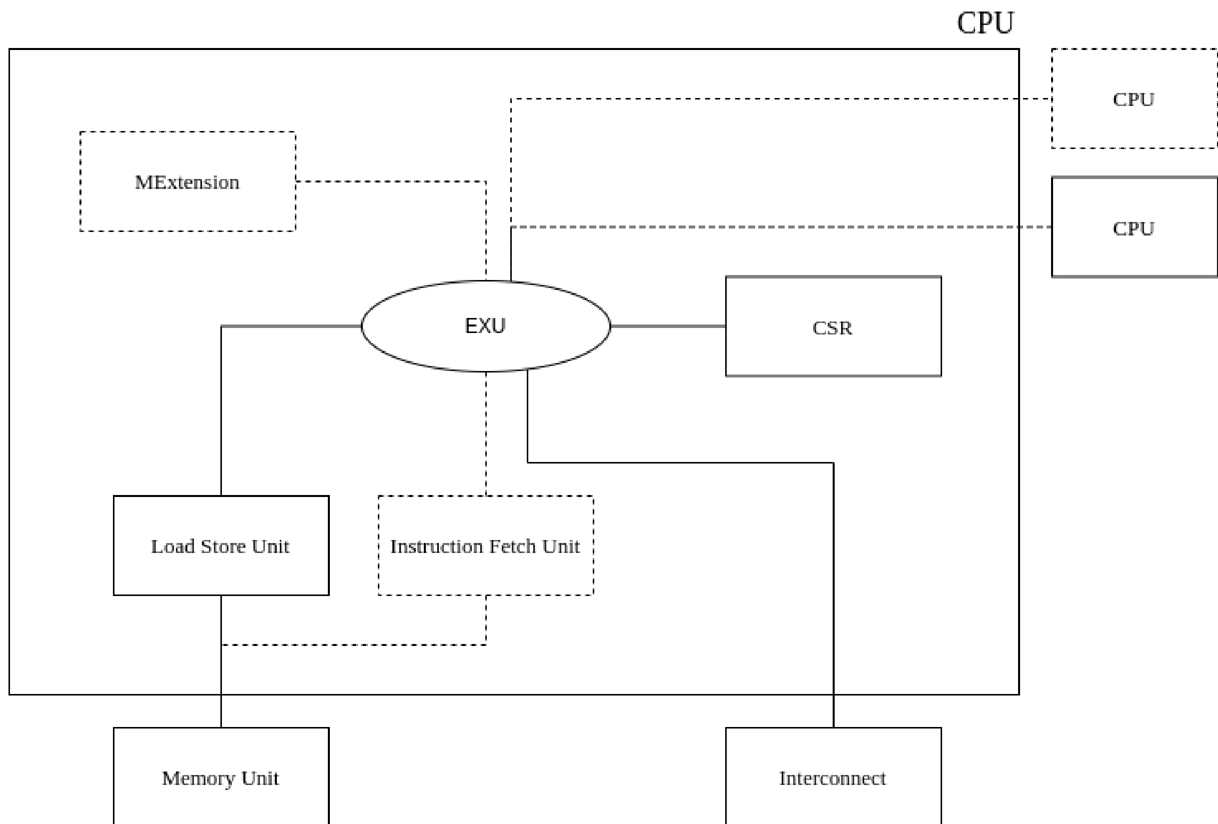


Figure 3: CPU Module Architecture

2.2.2 Memory Unit

The Memory Unit is responsible for granting access to the shared Wishbone Bus and features caching functionalities to accelerate read and write operations on the memory. In order to support Linux on the ParaNut processor, it was extended by a MMU (Memory Management Unit) to support paging. A detailed description of this implementation and the necessary changes and applied concepts can be found in [3].

This module differs in synthesis and simulation. When synthesized for the Zynq platform, the main memory is accessed through the platform's AXI Bus and DDR3 Controller. The MMU, is connected to that AXI Bus through a Wishbone to AXI bridge. In simulation, a memory is

simulated in software that is directly connected to the AXI Bus through the interconnect module. The simulated memory can be configured to exhibit the same behavior as the DDR3 Memory on the Zynq development board.

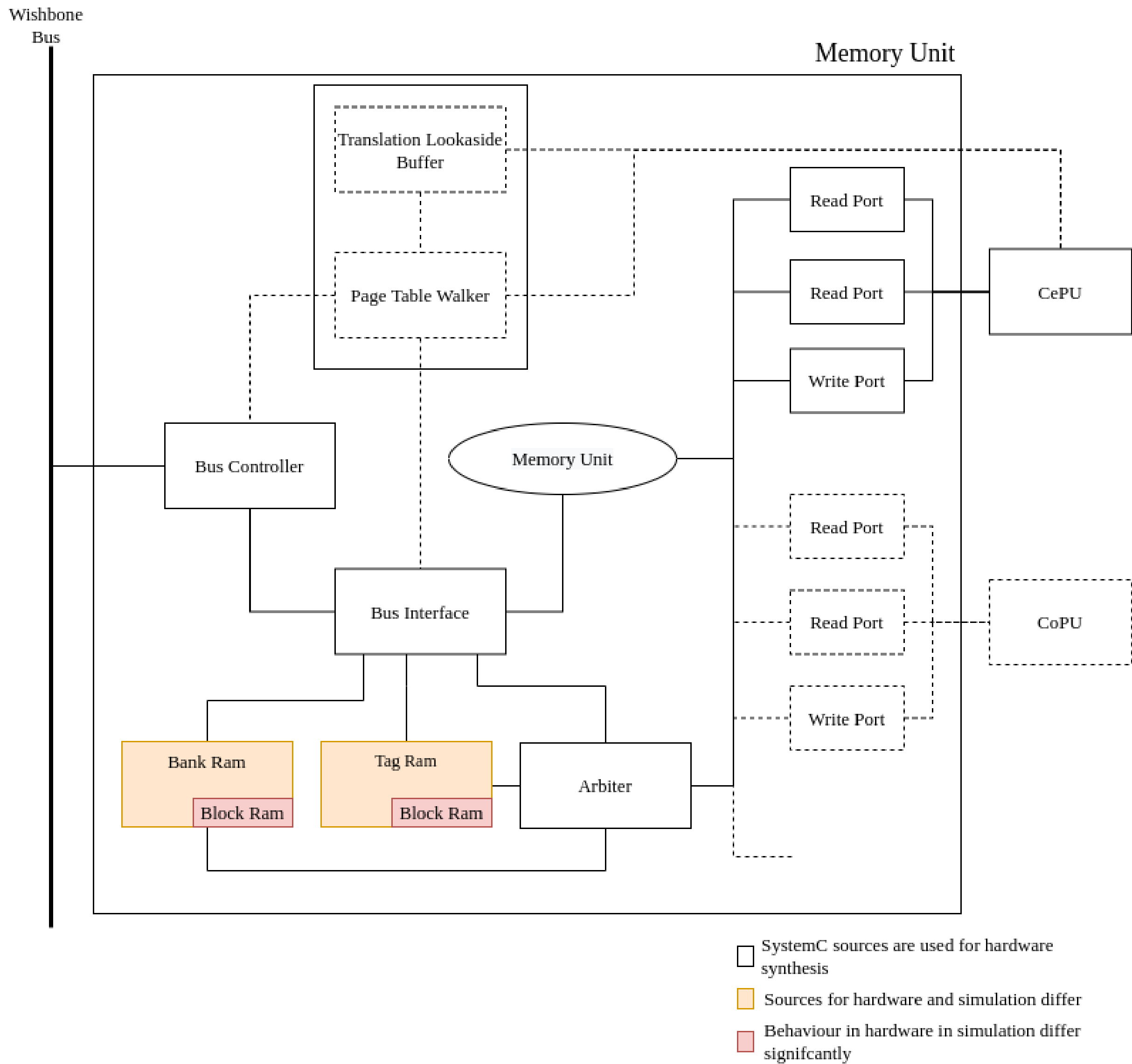


Figure 4: Memory Module Architecture

Fig. 4 illustrates the submodules comprising the ParaNut Memory Module. The number of read and write ports is dependent on the number of CPUs in the system. The two modules *Translation Lookaside Buffer* and *Page Table Walker* form a MMU and are only present when the system is specifically configured to contain this feature. Both, the *Page Table Walker* and the *Bus interface* provide a complete Wishbone interface. Which of these can access the main system bus is controlled by the *Bus Controller*. Memory access requests from read and write ports are arbitrated by the *Arbiter*. When operating in cached mode, the Arbiter delegates the request to the Tag and Bank Ram. If the requested address is not cached, the memory unit will access it throgh the Bus interface directly. In direct access mode, the requests are arbitrated and

routed to the Bus Interface.

Both, the Bank and tag RAM are written in VHDL as of version 1.1, as they have to contain specific syntax for the contained registers to be implemented in block RAM at hardware synthesis. The memory unit contains a Tag and block RAM module for each CPU.

2.2.3 Interconnect

In order ro grant different modules, that internally use the same address space, access to the Wishbone Bus, the hardware includes an interconnect module. It is used to route Wishbone access to the different connected modules and remove the address offset indicating which module is addressed. The memory does not necessitate address translation and is thus directly connected to the bus. The module contains a function call to add peripherals to the system and setting routing rules accordingly. Besides routing Wishbone access, the interconnect module is responsible for prioritizing and routing interrupts.

2.2.4 UART

The UART module, if included in a system, is accessible through a register based interface over the Wishbone Bus. It is currently not available for simulation, as the ParaNut simulator does not provide means to connect virtual or external devices to a wire interface. Besides enabling communication with external UART devices, it is intended to be used for flashing programs to the system memory. The implementation of this is currently in progress. Users can access the UART functionalities through a custom library called *libuart*.

2.2.5 GPIO

Like the UART module, the GPIO module interfaces with the processor logic through its registers, which are accessible over the Wishbone Bus. The widths of its digital in- and output busses is configurable. Reading and writing through software is accomplished with the appropriate, custom library *libgpio*. The module is not available in simulation.

2.2.6 MTimer

In order to provide timer interrupts to the ParaNut, the *MTimer* module was introduced. It is configurable by writing to its registers through the Wishbone Bus. Currently, no library for abstracting access to the timer in software is implemented. The two registers *mtime* and *mtimecmp* have to be written manually. It is directly connected to the CePU through an interrupt signal. Users must implement a custom interrupt handler to process the interrupt. For the ParaNut, the main motivation to add this module was supporting the real time operating system *FreeRTOS* and *Linux*.

2.3 Toolchain

The project's build system relies on several third party tools to compile, synthesize and build its components. The default compiler for compiling SystemC sources is the G++ compiler from the GNU Compiler Collection. The SystemC library used for building the simulator is provided by the Accellera Systems Initiative[2] For FPGA development, currently either the Zybo Z7 or Zybo Z7020 development board are used, both of which feature an FPGA from the Zynq-7000 family. The *Vivado* toolchain provided by Xilinx is used to run high level synthesis for translating the SystemC code to VHDL as well as for hardware synthesis from the mix of generated and written VHDL sources.

To compile software for a ParaNut processor, the *SiFive Freedom RISC-V Tools for Embedded Development*[3] in version 8.3 is used.

The project includes a number of scripts, some of which are using the Linux shell, others depend on Python 3.

3 ParaNut Hard- and Software Development

3.1 Eliminating VHDL Hardware Sources

The ParaNut's hardware description was written in a mix of VHDL and SystemC mainly because not all structures defined in the SystemC standard [5] are supported in the toolchain used for the project's development so far. One major omission from the subset is the ability to dynamically create arrays of modules at elaboration time. This feature is necessary to provide a flexible number of module instances, for example CPUs. For design, verification and simulation purposes, SystemC modules using this concept could be used, as the Accellera SystemC library does support the construct. VHDL sources were necessary to enable these features in hardware.

After Vivado 2020.1 the support for SystemC was discontinued [11].[4] The equivalency of VHDL and SystemC code had to be manually maintained by developers. While unified test benches would have been possible, the project so far did not feature any system to methodically ensure the identical behavior of hardware and simulation sources. At the time of writing, the processor exhibits faulty behavior when using the debugging interface on hardware, while working without issues in simulation. This misbehavior may or may not be caused by differing sources for simulation and synthesis. In any case, it is not yet solved, partly because it is impossible to reproduce the error in the simulator for debugging.

This mix and duplication of modules present in simulation and hardware is significantly harder to maintain, debug and extend, than a project written using a single hardware description language.

[2]The latest release may be found at: `https://www.accellera.org/downloads/standards/systemc` (last accessed 10.07.2023)

[3]Releases of the SiFive Freedom RISC-V toolchain can be found at `https://github.com/sifive/freedom-tools/releases/` (last accessed Jul 12th 2023)

[4]The latest description of the SystemC subset supported by the toolchain is documented in [12].

The official Xilinx statement concerning the discontinuation of SystemC High-Level Synthesis suggests using a third party tool [11]. As major parts of the project are already written in SystemC and the project benefits from the fast, cycle accurate simulator generated from those files, we aim to eliminate all VHDL code from the ParaNut sources.

In order to work towards the goal of setting up a synthesis process based on open source tools, the ICSC[5] was chosen for High-Level Synthesis. The Open Source tool was introduced in 2019 and is intended to synthesize SystemVerilog files from SystemC sources. Its SystemC subset supports more constructs from the SystemC standard [5], including dynamically initialized arrays of modules at elaboration time. Details on how the ICSC is included and used in the build system are discussed in Section 4.3.4.

Due to special syntax present in the project, that was originally required to produce a working model when using Vivado HLS, as well as syntax required by ICSC, *all* SystemC sources must be revised. Major parts of this work, as well as additional resources and findings, are documented in [7]. The modules marked in Fig. 2 required removing any non-synthesizable code and appropriate logic in order to replace it. This work describes the revision of the memory management module, as this was done by the author and the synthesis of large memory has to be treated specially.

Many constructs used in the SystemC sources of version 1.1 are technically synthesizable, however they do not comply with the project's coding conventions. As such, any occurance of the custom types *TWord*, *THalfWord* and *TByte* were replaced with the corresponding *sc_uint<>* type. Methods containing combinational logic are prefixed with *proc_cmb* and implemented registered with the macro *SC_METHOD*. Clocked processes are prefixed with *proc_clk* and registered using the *SC_CTHREAD* macro.

3.2 Memory Unit in SystemC

The *Memory Unit* has been extended multiple times since it was first introduced in the ParaNut processor. Most recently, support for RISC-V privilege level was added to enable Linux support on the ParaNut processor [3]. To facilitate all of the ParaNut's parallelism options, the *Memory Unit* was designed to serve simultaneous memory access requests using two read and one write port per processor core of *Capability Level 2* or higher. *Capability Level 1* cores only require one read and one write port. One of the read ports is connected to the core's IFU, exclusively used to retrieve instructions from the memory, while the combination of another read port and a write port is connected to it's LSU. The accesses for each core are prioritized in the following order:

1. LSU Read Port
2. IFU Read Port
3. LSU Write Port

[5] https://github.com/intel/systemc-compiler, accessed June 15, 2023

Requested memory can either be served from a configurable tag RAM cache, or from the memory connected to the AXI Bus.

Parts of the Memory Unit have been written in VHDL as well as SystemC, where the SystemC was not synthesizable. The respective modules are marked in Fig. 4.

In general, the Memory Unit is one of the most complex modules in the ParaNut hardware. It's original SystemC source consisted of ten modules distributed over three .cpp/.h source files (mmemu, ptw and tlb). The modules are now arrangend in more specific files, to improve readability:

- memory
- memu_arbiter
- block_ram
- memu_bankram
- memu_blockram
- memu_busif
- memu_common
- memu_readport
- memu_tag RAM
- memu_writeport
- memu
- ptw
- tlb

Each file is named after the module it provides.

The original SystemC implantation of the ParaNut processor relies heavily on structs to define interfaces between its components. This improves readability and reduces lines of code. Using structs as *sc_signals* was originally deemed impossible when using either *Vivado* or *ICSC*, with the release January 2023 of the *ICSC*, this feature was added. The feature description however excludes nested structs from being used as records. On requesting this feature on the *ICSC* github page, the developers stated, that implementing such a mechanism was not planned and an excessive effort. The memory unit originally featured such nested structs, which have been replaced by single layer structs on the basis of the following scheme: All signals present in a nested struct are prefixed with the sub-struct's name and directly added to the main record. Methods and functions accessing these values need to be reviewed to ensure functionality. To maintain readability many of these formerly nested structs now contain functions to retrieve data in the original sub struct data type.

The memory unit's cache logic relies on the use of block RAM cells when synthesized for FPGA systems to avoid excessive use of look up tables. The *Vivado Design Suite User Guide: Synthesis*[10] provides information on how to describe various block RAM configurations using VHDL and Verilog. Generating an appropriate Verilog block RAM model from SystemC, when using *ICSC* for high level synthesis, was unsuccessful. The resulting Verilog models are interpreted as registers and synthesized using look up tables. The ICSC provides a functionality for including

Verilog snippets to replace SystemC modules when synthesizing. Accordingly, block RAM was modeled in SystemC for simulation and Verilog for synthesis with Vivado. Future implementations may contain more implementations for other synthesis tools or modules.

The SystemC implementation of the tag RAM and Bankram modules now use these SystemC and Verilog block RAM models. In v1.1, both modules did not provide the same logic in SystemC and VHDL. Instead, the simulation code used simple arrays to model memory, supposedly to improve simulation performance, while VHDL made specific use of the block RAM modules as described by the Vivado documentation.

Additionally to the memory unit discussed so far, members of the ParaNut team developed an alternative memory unit to be used for simulation only. This module can be used to significantly speed up simulation time.

3.3 ParaNut Single Core

Prior to this work, a ParaNut System could contain a minimum of two cores. Parts of the project code already included sections and conditional statements to allow for a single core implementation. When trying to configure a single core system, a major issue was presented by registers and signals that were defined to have a width of *CFG_NUT_CPU_CORES_-LD*, i.e. the binary logarithm of the total number of cores. Examples of such occurrences are *ParaNut CPU enable register (pnce)* or *Hart ID Register (mhartid)* [6]. Some of these signals did not need to be included in the synthesis of a single core system, as they are used to enable communication between cores. Others signals needed to be fixed to a minimal width of 1 bit. To improve readability of affected code sections, the boolean preprocessor variable *CFG_NUT_-SINGLECORE* was introduced.

3.4 GCC Compatibility

The SiFive Freedom RISC-V toolchain release recommended for compiling software for the ParaNut platform contains the GCC (GNU Compiler Collection) 8.3.0. The most recent release of the toolchain provides GCC 10.2[6]. When building the toolchain from the official freedom-tools git [7], the provided GCC version is 12.2.0. Users should be able to use any of these versions to compile software for the ParaNut.

The files syscalls.c (providing ParaNut specific system call implementations) and paranut.ld (the default linker script for ParaNut software) required changes. Previously, the beginning of the heap was located through the *_tbss_end* symbol provided by gcc. With newer GCC versions this symbol is zero per default. Version 1.6 now uses the linker script to locate the heap through a symbol called *__heap_start*, which is both more verbose and robust against changes to the TLS (Thread Local Storage) through GCC.

[6]Freedom Tools Releases: `https://github.com/sifive/freedom-tools/releases` (online, last visited 19.07.2023)

[7]Freedom Tools Repository: `https://github.com/sifive/freedom-tools` (online, last visited 19.07.2023)

This heap location mechanism is used only inside the _sbrk function, which is responsible for allocating memory on the heap. The previous version was not able to allocate any heap when using newer GCC versions. Additionally, its exception handling was flawed. While trying to inform users about the error encountered when trying to allocate heap memory, *printf* was used to print the error. *printf* itself uses *sbrk* to allocate buffer memory, resulting in a recursive exception loop. This issue is solved by replacing *printf* with the more basic *_write* function.

4 Automated Building and Testing

4.1 The Build System

Build systems are an essential part of any software project, as they enable users and developers to easily compile software sources, generate documentation, execute tests and install the generated binaries. They also make up a large portion of all code present in any software project. Up to 31% of code files in a project, according to [9]. Apart from the number of files used to create a build system, maintaining and developing these build automation processes presents a large and complex part of software projects. Any build system should aim to provide the following benefits to a project [8]:

- **Efficiency:** Repetitive and complex tasks should be reduced to simple invocations. These tasks may encompass: Compiling and Linking software sources, building documentation, installing artifacts to a users system and executing tests.
- **Scalability:** In offering simple handles for complex, repetitive tasks, a build system should enable developers to maintain and develop large, complex projects without being knowledgeable in every single component contained in the project. Moreover, it should be easy to add components to an existing project and leverage its existing build system in order to build and integrate them easily.
- **Reproducibility:** As a major concept for software development in general, a build system should enable reproducible and reliable builds on differing systems.
- **Maintainability:** As discussed above, maintaining a build system is often a big part of software project development. It should thus forgo unnecessary complexity and be built with future maintenance work in mind.
- **Minimal dependencies:** Most build systems depend on third party tools and resources like compilers, binaries and libraries. The number of external dependencies should be minimal, improving usability, reducing maintenance effort and error sources.

Makefiles are one way to create and manage a build system. A Makefile is a build script most often used to automate compiling and linking in software projects. It is a script that is based on targets and prerequisites, where each target usually represents a file that is created by executing the commands (most often shell commands) associated with it.

Each target can have a number of prerequisites, which are either files or targets themselves. Upon invoking a target, its prerequisites are analyzed. Before executing the target itself, its prerequisites must be fulfilled. This is the case, if the named file's timestamp is more recent than all its prerequisite's timestamps. This working principle for example ensures, that object files are updated, when their corresponding c files have been modified.[8]

4.2 The ParaNut Build System

Users will mainly interact with the build system to install the ParaNut development environment to their system, define a custom ParaNut configuration, develop software for it and possibly extend it by some hardware component. Most of this interaction is achieved through central Make targets. Users may also make use of the "external" components, FreeRTOS and Linux, to base their software on.

Developers frequently need to interact with single hard or software components for debugging and development purposes. As such, these components must feature targets to selectively build and test them.

Therefore, the Make system must feature targets for a number of different tasks:

- Libraries
  - Building and Linking custom C libraries for the RISC-V architecture
- Applications
  - Cross compiling C applications from C and Assembly sources for the RISC-V architecture
  - Cross compiling Rust applications for the RISC-V architecture
- Hardware
  - Building and running test benches for SystemC hardware modules
  - Generating a SystemC based, cycle accurate hardware simulation
  - Running High-Level Synthesis on SystemC sources
  - Synthesizing for different FPGA platforms
  - Flashing a hardware configuration on to an FPGA
  - Running compiled software in the simulator
  - Running compiled software on hardware
- Documentation

[8] For more in depth information on the principles and functionalities of Makefiles, see `https://www.gnu.org/software/make/manual/make.html`

– Generate documentation from LaTeX sources

– Generate Doxygen documentation from C sources

In v1.1 of the ParaNut, almost every folder contains a Makefile to manage building and testing sources present in that folder and its subfolders. This results in more than 13% of the project's sources constituting Makefile code. Many of these Makefiles depend on artifacts created by other Makefiles in the directory tree.

```
# Configuration options
CROSS_COMPILE ?= riscv64-unknown-elf

CC       := $(CROSS_COMPILE)-gcc
GXX      := $(CROSS_COMPILE)-g++
OBJDUMP  := $(CROSS_COMPILE)-objdump
OBJCOPY  := $(CROSS_COMPILE)-objcopy
GDB      := $(CROSS_COMPILE)-gdb
AR       := $(CROSS_COMPILE)-ar
SIZE     := $(CROSS_COMPILE)-size

ELF = hello_newlib
SOURCES = $(wildcard *.c)
OBJECTS = $(patsubst %.c,%.o,$(SOURCES))
HEADERS = $(wildcard *.h)

CFG_MARCH ?= rv32i

CFLAGS = -O2 -march=$(CFG_MARCH) -mabi=ilp32 -I$(RISCV_COMMON_DIR)
LDFLAGS =  $(CFLAGS) -static -nostartfiles -lc $(RISCV_COMMON_DIR)/startup.S \
$(RISCV_COMMON_DIR)/syscalls.c -T $(RISCV_COMMON_DIR)/paranut.ld

# Software Targets
all: $(ELF) dump
$(ELF): $(OBJECTS)
        $(CC) -o $@ $^ $(LDFLAGS)

%.o:    %.c $(HEADERS)
        $(CC) -c $(CFLAGS) $<

# ParaNut Targets
.PHONY: sim
sim: $(ELF)
        +$(MAKE) -C $(PARANUT)/hw/sim
        $(PARANUT)/hw/sim/pn-sim -t0 $<
```

Figure 5: v1.1 Makefile Script for Simulating a c Program

Figure 5 shows an excerpt from the "hello-newlib" example application inside the *sw* directory. It showcases multiple issues present in the original build automation:

1. It defines all compiler and linker flags and the targets for compiling and linking a binary, although the ones present in this particular software are needed by every software compilation for the ParaNut processor. Put inversely, every Makefile contains identical sections that need to be maintained seperatly

2. Multiple folders are referenced from the project root, which is held in an environment variable $(PARANUT), acquired by sourcing settings.sh. This too decreases maintainability as these paths have to be separately updated in every Makefile.

3. The simulation target depends on the pn-sim binary being built. This is not assured through a prerequisite, but rather through invoking its build target per default. (additionally, the order in which "+$(MAKE) -C $(PARANUT)/hw/sim" and "$(PARANUT)/hw/sim/pn-sim -t0 $<" are executed is not assured by Make)

4. The file in general could contain more detailed documentation

The complexity of the build process is further increased, as the project's sources at version 1.1 contained unique Makefiles for each application, hardware component, test bench and library. This approach of a distributed build system led to a major maintenance cost. Small changes in architecture, variable naming or tooling necessitated manual changes to many independent files. This could easily result in the neglect of a number of components, resulting in the breakage of their build process. The implementation of a server side CI/CD (Continuous Integration / Continuous Delivery) pipeline could help to intercept such errors. However, this would not resolve the issue of an increased maintenance effort compared to a centralized build approach, that would incidentally update all build processes.

4.3 Designing a New Build System

The existing build system was redesigned based on the target list in 4.2. The list is grouped into libraries, applications, hardware and documentation targets. Each of these groups may be treated as an independent subproject whose build system produces distinct artifacts.

One of the most complex targets in the ParaNut project is running an application, that needs to be linked with a ParaNut specific library and compiled for RISC-V, in the hardware simulator (pn-sim), as was already outlined by the old Makefile shown in Figure 5.

As suggested by [2], the build process was analyzed from a top-down perspective. The resulting dependency graph, containing intermediate artifacts is shown in Figure 6. Note that this graph is a schematic representation, as listing all actual sources and artifacts would result in a large, unintelligible graph. It is split into three columns, that visualize the boarders between the three main groupings *Libraries*, *Application* and *Hardware*.

As before, most of the build system's processes can be executed from the main Makefile located in the project root. For example, running *make hello* will trigger the process shown in 6. Aside from basic build and clean targets, it contains test-targets that can be used to test specific project components like the simulator or high level synthesis.

The new system is based on centralized information, thus a single resource containing relevant information is provided to enable sharing information between the subprojects: directory-base.mk.

4.3.1 Directory Base

The file directory-base.mk contains information on where the subprojects, their components, tools and files used by multiple Makefiles are located within the directory. Additionally, it is used to provide information on the structure and location of the currently selected ParaNut System (ParaNut Systems and their function are discussed later in this chapter).

All the paths provided by the file are calculated from the environment variable $PARANUT_HOME[9]

[9] Appending "_HOME" to the project name when defining an environment variable that points to the project installation seems to be an established naming convention (e.g. ICSC_HOME, VIVADO_HOME, ANDROID_-HOME)

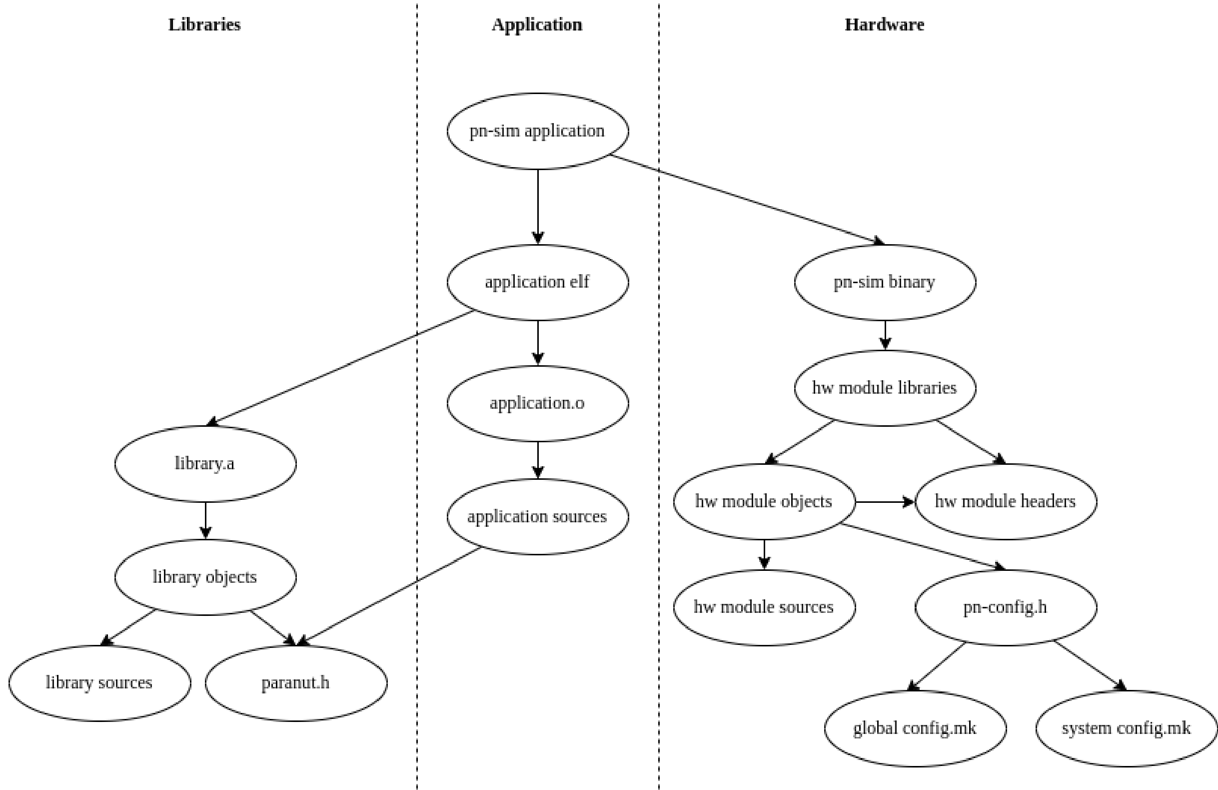


Figure 6: Schematic Dependency Tree for Running an Application in the Simulator

that points to the currently active ParaNut directory. Usually users and developers will source the *settings.sh* in the directory root. Apart from setting *PARANUT_HOME*, sourcing *settings.sh* adds the binaries located in $(PARANUT_HOME)/tools/bin to the user's $PATH environment variable. When the ParaNut is installed on a system, $PARANUT_HOME will point to the installation directory (see Section 4.5).

When making changes to the project architecture, developers should aim to adapt this file first before adapting other build tools to ensure synchronicity between all tools.

Some shell based tools inside the ParaNut project require information on the location of artifacts and components (e.g. pn-newproject, see Section 4.5). The dedicated target *get-variables* can be utilized to provide these to such scripts.

Targets often used within each subproject are centralized as well. The three main subprojects: Libraries, Applications and Hardware each provide a dedicated base-Makefile that contains targets that almost all components inside the subproject need.

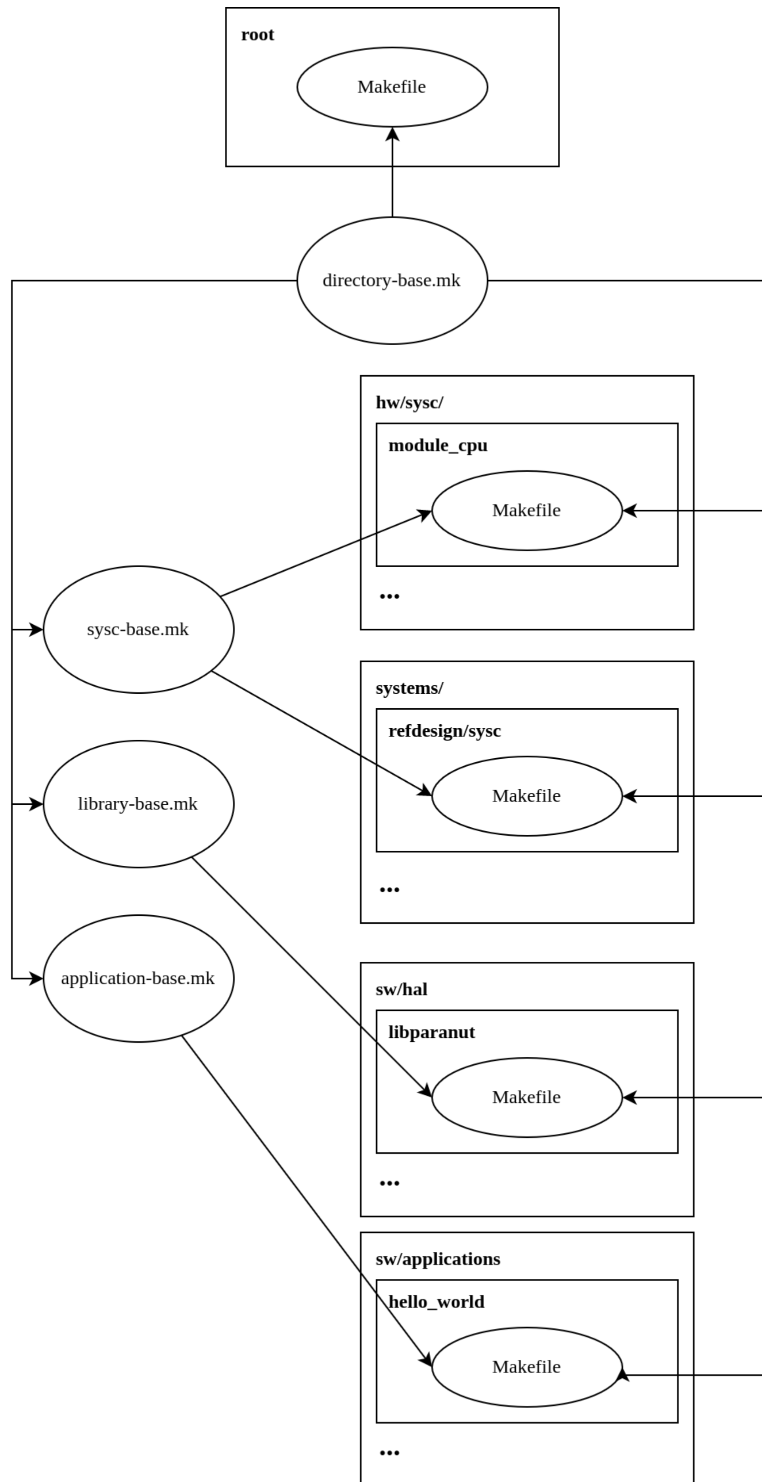


Figure 7: Makefile Inclusion/Inheritance Scheme

Figure 7 shows where each of the four base-Makefiles are included. Thanks to this centralized approach, top level Makefiles almost exclusively contain variable definitions to set source files and configuration. The Makefiles of the components inside a given subproject are structured identically. For example, the Makefile for the *hello_newlib* only contains source and build con-

figuration and the base-Makefile includes directives, as shown in Fig. 8.

```
################# Software Configuration ##########################################

# SOFTWARE_SRC:                Sources that are used for Simulation as well as
#                              synthesis. (including the testbench/sc_main file)
SOFTWARE_SRC ?= hello_newlib.c

# LIBRARY_DEPENDENCIES:        Paranut specific libraries, this software
#                              depends on. Mainly used to make shure the respective
#                              files are built
LIBRARY_DEPENDENCIES ?=

# SOFTWARE_CFLAGS:             Additional compiler flags this software needs
SOFTWARE_CFLAGS ?=

# SOFTWARE_LDFLAGS:            Additional linker flags this software needs
SOFTWARE_LDFLAGS ?=

# SYSTEM:                      Defines the system the software is run on
#                              in simulation as well as on hardware
PN_SYSTEM ?= refdesign


##################################################################################
##### DO NOT EDIT FROM HERE ON ###################################################
##################################################################################

################# Module Makefile ################################################
APPLICATION_DIR:= $(abspath $(CURDIR))
PARANUT_HOME ?= $(CURDIR)/../../..

################# Automatic Name Generation ######################################
# Get Module name from folder name
SOFTWARE_NAME = $(lastword $(subst /, ,$(APPLICATION_DIR)))

################# Directory Makefile Include #####################################
include $(PARANUT_HOME)/directory-base.mk

################# Master Makefile Include ########################################
include $(APPLICATION_BASE_MK)
```

Figure 8: v1.6 Makefile Script for Simulating a c Program

In special cases these files can be extended with unique build targets. The three base-Makefiles for subprojects are discussed in the following sections. All of these Makefiles **must** define the basic targets:

- build
- clean
- help

4.3.2 Libraries - hal-base.mk

Especially in the past year, the number of custom libraries for the ParaNut architecture has increased significantly. While the project originally featured only the *libparanut*, multiple hardware abstraction libraries were added, e.g. for accessing the UART (Universal Asynchronous Receiver / Transmitter) module. Some of these libraries include others, this interdependence was not ensured by the previous build system. The centralized approach allows for easy configuration of a librarie's sources, dependencies and special compiler options. This is accomplished by designing a single Makefile named *hal-base.mk*, that features all commonly used targets for

creating a library artifact. Each library can provide this information in a simple Makefile with verbose variable names for the build information data. The new folder architecture discussed in Section 4.4 enables interdependencies between the different library builds without declaring full paths in each library Makefile. Libraries are built recursively, as one library depends on another library file, its existence and up-to-dateness is checked and built if necessary. The library build process is only dependent on the presence of the riscv-toolchain and a single target from the hardware build process, as this build process creates a paranut-config.h that is needed by some libraries. More information on this particular target may be found in Section 4.3.4.

4.3.3 Applications - application-base.mk

To unify the targets for compiling, linking and running applications, a central Makefile named *application-base.mk* was designed. Most applications can be build by defining only the software source files. This Makefile mainly aims to streamline and unify the development process. While most applications originally only supported a single simulation target: "make sim", this base Makefile defines multiple simulation targets for simulating with instruction log, signal trace or in debug mode. It has two interfaces to other build systems, one for ensuring the presence of necessary ParaNut libraries and one for building the hardware simulator. It relies on the riscv-toolchain, which needs to be installed. The build process per default uses the riscv64-unknown-elf-* binaries. Should another toolchain be required, it can be set by appending CROSS_-COMPILE=<toolchain> to the make call. In future revisions, a target for loading the compiled binary to a hardware implementation of the ParaNut processor will be added.

4.3.4 Hardware - sysc-base.mk

At its core, the ParaNut project provides a highly scalable soft core processor. Accordingly, synthesizing, simulating and testing the project's hardware design are its core features. As with the subprojects discussed so far, the hardware too was extended significantly since the last release. This part of the build system was subject to the biggest changes. Partly this is due to the project architecture decisions described in Section 4.4. Mainly, the build system adaptations regarding the hardware subproject are founded on the aim of eliminating VHDL sources as discussed in Section 3.1. This change included the introduction of the ICSC for High-Level Synthesis. The process of which is detailed in Section 4.6. In version 1.1, all hardware sources were kept in two folders one for VHDL and one for SystemC sources. With a growing number of hardware modules, this architecture was becoming hard to read, therefore hardware sources were grouped into processor modules similar to the modules described in Section 2.2. The modules currently present in the ParaNut project are listed in Figure 9.

```
sysc
├─module_cache
├─module_common
├─module_cpu
├─module_debug
├─module_gpio
├─module_mmemu
├─module_mtimer
├─module_system
└─module_uart
```

Figure 9: ParaNut SystemC Modules

Similar to HAL (Hardware Abstraction Layer) and software, each hardware module provides a simple Makefile, that contains build information. Each module can be considered an autonomous hardware build, as it is intended to provide one or more test benches for assuring function and building a simulator, a top module for building and providing a SystemC library to be included in external modules and a target for High-Level Synthesis. At the time of writing, no target for synthesizing for hardware is implemented, as this feature necessitates working High-Level Synthesis using the ICSC for all modules. This transition is still in progress as discussed in [7].

The module *common* contains sources for configuration, helper functions, macros and functions for simulation. Moreover, it contains a template class for peripherals. Its interface can be used by developers to build additional Wishbone modules for a ParaNut System. Because of its function as a repository for helper classes and functions it also includes sources for simulating the SystemC code when using the Accellera SystemC library instead of the one provided by ICSC. Details on this can be found in Section 4.6

The file *sysc-base.mk* contains two distinct sections one for High-Level Synthesis and one for compiling a simulator. In order to build a simulator for a hardware module, potential dependencies must be resolved. This is done by using a variable defined in the module's Makefile. The variable contains all modules the module depends on in order to be simulated correctly. These interdependencies are resolved recursively, meaning the build process resolves all interdependencies and builds the modules that have no dependencies first, incrementally building libraries for all modules until resolving the prerequisites of the module whose build target was invoked.

The targets and steps to create a test bench binary are the exact same as for creating a simulator, since a test bench in SystemC is a simple simulation with predefined signal states to test the hardware functionality. In order to prevent confusion when being used by users not acquainted with the SystemC simulation principle, two Phony targets are provided for running and building the simulation/test bench: *build-sim, run-sim* and *build-tb, run-tb.* Both targets build and run the same binary. Depending on the intent of the user one may be more intuitive than the other.

Besides the main targets build-sim (building a simulator/test bench) and build-syn (running High-Level Synthesis to build a Verilog source), the sysc-base.mk contains a target for creating the paranut-config.h. This target refers to the *module_ common*, which provides the functionality to parse the config.mk (see 4.3.5) files present in the project root and systems directories into a header file to be used by the hardware build and some libraries and applications.

The modules listed in 9 do not include a central instance intended to build a simulator of the whole ParaNut. *module_ system* does contain the logic to connect the various modules and could theoretically be used to simulate a complete ParaNut System. However, it is intended to simplify Systems that use default components and reduce the number of files required to describe them. Systems using non-default components must omit this module and provide their own logic for connecting them.[10]

4.3.5 ParaNut Systems

A ParaNut System is the collection of a specific hardware configuration, potentially additional hardware modules and the artifacts resulting from building this configuration, needed to simulate or synthesize a respective ParaNut processor. The default ParaNut configuration (number of cores, capability levels, cache size, etc.) is contained in a *config.mk* file in the root directory. A ParaNut System can alter this base configuration by providing a seperate *config.mk* that contains all configuration options that differ from the default. This adaption is a change in comparison to the previous release, where each system contained a custom Makefile and a custom config.mk that redefined all config options. Developers experienced, that some systems were neglected when updating the main config.mk, accidentally breaking these systems' build processes.

Logically, a ParaNut System is a container for at least one hardware module itself, which at its minimum contains the *sc_ main* needed to provide a full processor simulation. This design enables reusing the sysc-base.mk for the simulator build from any desired directory and allows users and developers to design additional hardware for a specific system.

The concept of ParaNut Systems is central to the new build process, as it simplifies simultaneously working with different hardware configurations. A ParaNut System folder provides all artifacts created by the build process, as well as access to its hardware abstraction libraries. The default system to be used when running an application using the *make sim* target inside an application folder is defined in the directory-base.mk. It may however be overwritten inside the application's Makefile, as is necessary for some applications (e.g. Linux) or by the user by adding PN_SYSTEM=<system name> to the call. The concept of ParaNut Systems is again reused when using the ParaNut project's software development workflow, which will be discussed in Section 4.5.

A ParaNut System's folder structure is displayed in Fig. 10. Most importantly it provides the custom config.mk. The *sw* directory contains artifacts from building ParaNut libraries, such as

[10] An example is the System for Linux simulation, which needs to include and connect the simulation-only memory module.

headers and library files split into *lib* and *include* folders. The *hw* folder contains a directory for board specific files that will be needed for synthesizing for different platforms. It also contains a *sysc* folder that contains a SystemC top level module and may hold additional SystemC sources. When the simulator for the System is built the binary is placed into this folder. Moreover, the *sysc* folder contains an *include* directory, that contains the libraries created by building all hardware modules referenced in the Makefile. As these libraries need to be built for each configuration, this allows for retaining preliminary build artifacts for different system configuration, which was not possible with the old build system.

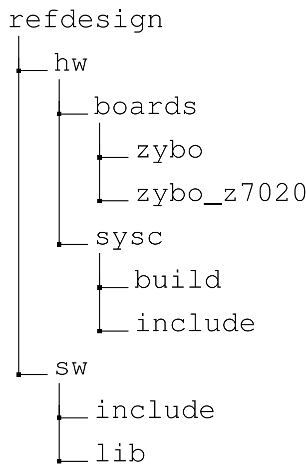


Figure 10: ParaNut System Directory

4.3.6 Continuous Integration/Continuous Delivery

In parallel to the structural improvements described in this document, the ParaNut developer team introduced a Jenkins[11] automation server. It is used to test the ParaNut project on two levels. The top level perspective tests targets contained in the main Makefile. In order to quickly locate an error encountered within this process, each component can be tested separately. Aside from providing information on the project's health, the CI/CD regularly runs the benchmarks used to gage the processor's performance. This allows developers to monitor the impact of changes on the ParaNut's performance.

4.4 Project Architecture

The ParaNut Project was and is largely maintained as a single, repository containing sources for hardware, software, documentation and tools. It's directory structure now mirrors the subproject approach. As of now, the project could easily be split into multiple repositories.

Figure 11 shows the first three levels of the new directory structure of the ParaNut Project. The fourth layer contains the software or hardware modules and components and is not listed for intelligibility. Only *sw*, *hw* and *systems* are deeper than three levels. The structure beneath

[11] urlhttps://www.jenkins.io/, last visited 25.07.2023

hw->sysc can be seen in Fig. 9 and is discussed in Section 4.3.4. *sw* was redesigned from a monolithic design to contain seperate folders for applications, libraries (hal), operating systems (os) and benchmarks/tests (test-applications). The specific applications and libraries are not relevant to this work and will not be listed.

As explained in Section 4.3.4, ParaNut Systems have a special role in the ParaNut project, their basic working principle, as well as their folder structure was already discussed and displayed in Fig. 10.

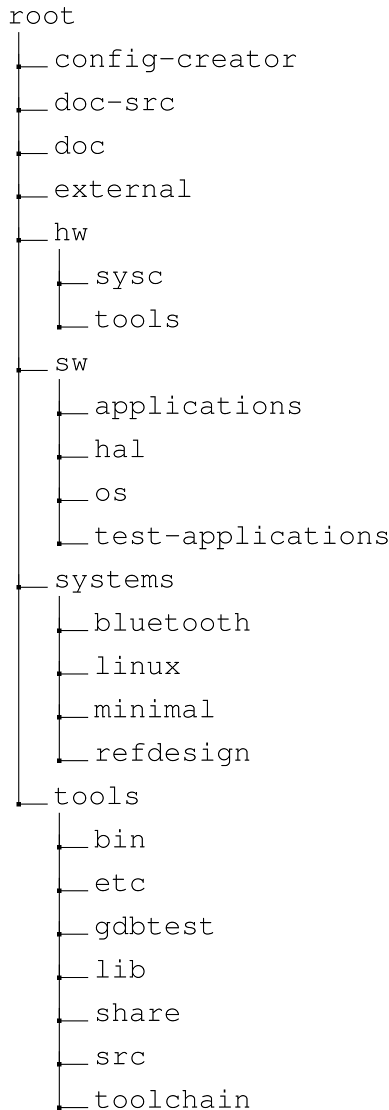


Figure 11: ParaNut Root Directory

In general, the changes to the repository structure can be considered minor, as another level for sorting hardware and software sources into more general modules or categories was added for the most part. These changes would not have been easily possible with the old build system in place. Especially the introduction of the *directory-base.mk* enables a much more flexible structure and

a build process that can easily be adopted to changes. This allows for future improvements to be easily implemented if necessary.

4.5 Installation

The ParaNut's installation feature aims to provide a development environment for users of the project. The installation should provide resources necessary for developing software, like libraries, a simulation environment, as well as tooling for flashing the configured hardware and software to a development board. Therefore, the main Makefile offers a *make install* target. Per default, this target installs necessary sources and artifacts to */opt/paranut*. The target directory may be configured by providing *INSTALL_PREFIX* to the installation call. If the target directory is not owned by the user, the call must be made with *sudo*. Fig. 12 shows the structure of the installation directory, essentially mirroring the structure of the repository, while omitting unnecessary sources.

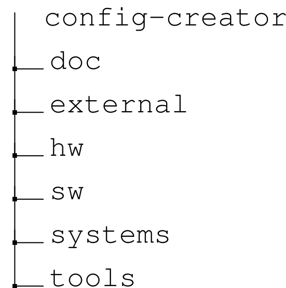


Figure 12: ParaNut Root Directory

Besides already discussed subdirectories, the installed sources contain a folder named *external*. This folder may contain sources from external repositories for use by software components, e.g. the FreeRTOS-Kernel. Users can decide to add these sources to the installation. Currently two software components use external repositories, Linux and FreeRTOS. They can be included by adding *OS_INSTALL="freertos linux"* to the installation call.

Binaries important for the usage are located in *tools/bin*. This folder is added to the $PATH environment variable when sourcing the *settings.sh* contained in the installation root. These binaries include the *pn-config-creator* (used for easily creating ParaNut configurations), *pn-flash* (for flashing hardware) and *pn-newproject*.

pn-newproject is the main access point for users to interact with the installed ParaNut project. It creates a folder containing a copy of the default ParaNut System as defined in 4.3.4. In addition to the sources present in the default ParaNut System, a write protected *config.reference.mk* is added that contains a copy of the main config.mk. It is present in the installation directory for users to reference what the current default configuration encompasses. It also contains a copy of the version.env from the installation directory to track the version of the ParaNut project with which the project directory was created and a special README containing information for the

users.

Having sourced *settings.sh*, the users are enabled to edit the contained *config.mk* to configure their ParaNut System. All artifacts from building this system are automatically stored inside this folder's structure.

4.6 Synthesis with SystemC

Part of the reason for re-organizing the build system and architecture of the project was the need to incorporate High-Level Synthesis using the ICSC (Intel Compiler for SystemC). The build process provided by ICSC is designed around the tool *CMake*. Adopting the first hardware modules for synthesis using this build chain, prooved its complexity. In order to create the final synthesis target, a SystemVerilog file, three shell instructions need to be executed. `make` for creating the cmake script, `cmake` for creating the binary used to generate the SystemVerilog target and `./sc_tool` to generate the final result. In order to streamline the development and deployment process, this procedure needed to be simplified. The build system present in the ParaNut project is centered around *Make*. As this is already part of the build process intended by the developers of the ICSC, it was deemed rational to reduce the toolchain and make cmake obsolete. The steps accomplished by the `cmake` invocation in the ICSC repository are easily reproduced using Make. Two separate cases had to be considered:

- Compiling the sources to produce a simulator
- Synthesizing a single Verilog file

The first case is easily accomplished by compiling a binary from all SystemC sources, where exactly one test bench file is included in the sources, containing a sc_main. When linked with a SystemC library, this produces a hardware simulation. The new build system is intended to provide flexibility regarding the SystemC library used. Developers may either use a library installed from their system's package management system, compiled from the Accellera SystemC sources or the library provided with the ICSC. In order to simplify selecting and configuring the desired library, the *systemc_config.mk* file was added to the ParaNut root directory. The build system is setup to use the system library, if no environment variable *SYSTEMC_HOME* is provided. *SYSTEMC_HOME* may either point to an Accelera[12] or ICSC installation directory. If it points to the ICSC directory, *USE_ICSC=1* must be set for the right configuration to be used.

When using the SystemC library fork contained in the ICSC repository, two functions important to the synthesis and simulation of a ParaNut System are available: *sc_new* and *sc_newarray*. Both functions are needed to dynamically create module instances in SystemC. These functions however are not present in the Accelera implementation of SystemC. An alternative implementation of these is contained in the *module_common* that provides these function calls and im-

[12] Releases and documentation reguarding this library can be found at `https://www.accellera.org/downloads/standards/systemc`

plements their behavior for builds using the Accelera SystemC. An include guard protects the functions from being redefined when using the ICSC library. Some modules, such as the memory management module contain sources that are only used when generating a simulator binary and can not be synthesized for hardware, as the contained functionalities are either unnecessary or already implemented otherwise in hardware. Such sources must be indicated by being provided in the module Makefile's *SIM_SRC* variable. All other sources are provided with *MODULE_SRC* and *TESTBENCH_SRC*. *TESTBENCH_SRC* must contain the name of the file containing the sc_main function. This way multiple test benches can be compiled and executed for a single module by either creating distinct Makefiles or overwriting this variable when calling make.

Reproducing the synthesis process accomplished by the original cmake process necessitates the dynamic generation of a C-file that includes all source files and a compiler configuration string. This is accomplished by utilizing the source information in the hardware module's Makefile and a template file that is parsed using the command line text editor *sed*. In addition, the cmake process creates a folder that contains copies of all involved source files. This is reproduced by creating a "sc_build" folder and recursively copying the module's and its dependencies' sources to that folder. The *sc_tool* file is compiled using gcc and generates a SystemVerilog file when run. The Xilinx Vivado toolchain used for hardware synthesis supports SystemVerilog, its High-Level Synthesis however produces a Verilog output and the general support for special constructs (like block RAM) is better in Verilog. Respectively, a third party tool for transforming the SystemVerilog output to Verilog is used. The tool is called *sv2v* (SystemVerilog to Verilog) and provided by `https://github.com/zachjs/sv2v`. It needs to be manually installed by developers and is a simple command line application. Utilizing Make's prerequisite logic, the full process for generating a Verilog file from SystemC sources can be accomplished by running *make build-verilog*

5 Evaluation

5.1 Build System

The complexity of a build system can be measured in a number of ways. In using Makefiles for automated building and testing, measures such as number of lines, targets, dependencies and indirections[13] can be used. All with the aim of measuring the system's tangibility. [8]

To verify the basic functionality, a peer review was conducted, the findings of which were incorporated into the final design. Users and developers were not provided with additional information, as the review was intended to test the build system's documentation and legibility. With the ParaNut Project, the build system must be evaluated from two points of view, the developer's as well as the user's perspective. Some participants do not work with or on the ParaNut project or have just recently started to, allowing for feedback from a user perspective. The user's interaction with the build system did not change significantly in comparison to v1.1. The steps to install the installation tree, create a new project, and compile and simulate software on it are largely

[13] i.e. instances of features that require the reader to look somewhere else" [8]

identical. The main difference regarding the user perspective being, that artifacts from building and configuring the simulator are now placed in the project tree, rather than the installation tree. A change not noted by test users. Feedback regarding the usability were incorporated into the final release.

The developer's perspective was continuously reviewed, as some developers adopted the build system changes in an early state and gave constant feedback, greatly improving the system's reliability.

Using the CI/CD to run all the project's tests and applications using the new build system enabled fast validation of functionality and reliability.

Concerning build system complexity, the main improvement is the consolidation in base-Makefiles, reducing indirections. When working on a subproject's component, exactly three Makefiles are relevant: The component configuration file, the respective base Makefile and *directory-base.mk*. To change or extend the build process of all components in a subproject, only the subproject's base file must be modified (With v1.1, all the component's Makefiles needed to be changed). Additionally, the Makefile documentation was improved.

5.2 Processor Optimizations

5.2.1 Memory Unit

The memory unit's performance in read/write operations per cycle has remained the same, as can be seen in the performance test results in Table 2.

	Memory Unit v1.1	**Memory Unit v1.6**
sequentially accessing 2048 words		
writing, first run	923,10	933,64
writing, second run	300,00	300,00
reading	200,00	200,00
parallel write and read 2048 words, 4 ports		
writing (adjacent words)	2461,33	2437,7
reading (adjacent words)	300,00	300,00
writing (different sets and banks)	1829,88	1828,52
reading (same words)	501,95	501,90
reading (random words)	350,29	350,29

all figures in clocks per operation

Table 2: Memory performance comparison

Memory simulation complexity is increased through the introduction of multiple block RAM simulations. This affects simulation performance. The runtime of the test bench used to generate the results in Table 2 increased from 11,7s to 19,8s (Average of five runs, using the time shell

tool and accumulating system and user figures). A loss in performance was expected and is acceptable, since this module is primarily intended for synthesis and debugging. Developers recently produced a memory module intended for fast simulation, that can not be synthesised[14]. In simulation, it outperforms both memory units significantly, only taking 1,1s.

Using the default cache/memory configuration for the ParaNut, the memory unit was synthesized. Table 3 shows the default configuration options from config.mk. Table 4 lists the resulting parameters for the different block RAMs generated by the memory unit.

Parameter	Default Value
CFG_MEMU_CACHE_BANKS_LD	2
CFG_MEMU_CACHE_SETS_LD	9
CFG_MEMU_CACHE_WAYS_LD	2

Table 3: Default Cache Parameters

Module Description		**Port A**				**Size**
		Width	Depth	Write	Read	
Cache	**Port A**	32 Bit	2048 Bit	YES	YES	8192 Byte
	Port B	32 Bit	2048 Bit	YES	YES	
tag RAM	**Port A**	25 Bit	2048 Bit	YES	NO	6400 Byte
	Port B	100 Bit	512 Bit	NO	YES	
LruRam	**Port A**	6 Bit	512 Bit	YES	NO	384 Byte
	Port B	6 Bit	512 Bit	NO	YES	

Table 4: Memory Unit block RAM Usage

The synthesis usage report from Vivado confirms that the block RAMs are generated as predicted. Note, that each of the memories is generated once for each core. The default four core system thus contains four of each. When synthesized for the Xilinx ZYNQ 7020 platform, the memory unit itself uses 5720 look up tables. A comparison to the look up table usage from v1.1 can be found in Table 5. The difference between both versions seems to be mainly caused by the Arbiter and tag RAM modules. Both synthesized designs use 14 block RAM tiles and between 1050 and 1090 register.

5.2.2 Single Core

Aside from a reduction in hardware size, the single core ParaNut System is intended for increasing simulation performance in single threaded applications. Table 6 shows simulation times for two such applications. In comparison to the previously smallest system (two cores), the single core system simulation is 47% faster . All tests were conducted on v1.6 of the ParaNut project, other

[14]This memory unit is not part of the v1.6 release and still under development, hence results are preliminary.

Component	VHDL Memory Unit	SystemC Memory Unit
overall	3583	5720
Arbiter	701	1726
tag RAM	243	1052
Bus Interface	783	437

Table 5: Look Up Table Usage Comparison

	Single Core	Dual Core
Time until flaiure	59,47 min	86,48 min
Clock cycles until failure	$240 \cdot 10^{-9}$ (equivalent to 9,6s at 25Mhz)	
Lines of Linux boot messages	29	

Table 7: Linux boot process on different ParaNut Systems

than the number of cores, no parameters were altered. The simulated CoPUs are of capability level 2. Fig. 13 visualizes the measurements from Table 6. The influence of the number of cores on simulation time is approximately linear. The increase of simulation time based on the number of cores can be approximated as a factor, using the following formula:

$T(c) = 0,24 * c + 1$

T: Simulation time factor in comparison to a single core system, c: number of cores

Application	1 CPU	2 CPUs	4 CPUs	8 CPUs
hello_newlib	2,44s	3,35s	4,99s	7,62s
dhrystone (10000 runs)	343,8s	506,1s	663,9s	1123,8s

Table 6: Simulation times of single threaded applications.

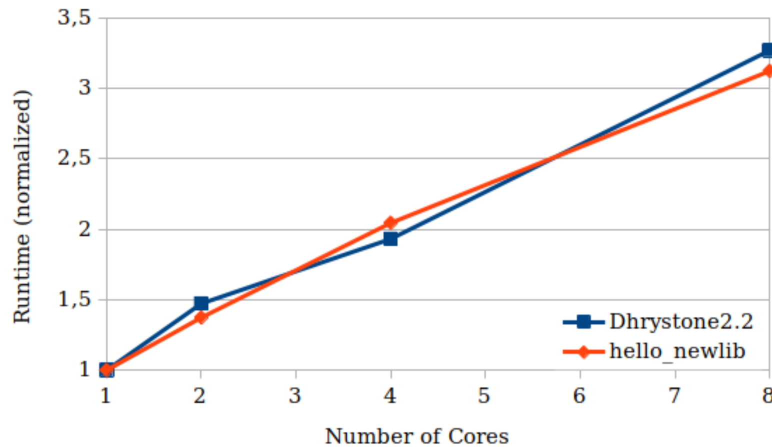


Figure 13: Simulation Performance in Comparison to Number of CPUs

Faster simulation times allow developers to more efficiently debug persistent hard- and software errors. This is especially useful for debugging sophisticated software such as Linux. Table 7

shows a comparison between simulating the Linux boot process on a single and a dual core ParaNut System. Note, that at the time of writing the ParaNut is not capable of running Linux. Developers are working on supporting the operating system. As of now, the boot process aborts due to an unknown error. The single core should help debug this issue.

As a full ParaNut System can not yet be synthesized, the impact of a single core system on the look up table count can not be measured.

6 Conclusion

The work presented in this document is key to enabling the usage of open source tools for synthesizing the ParaNut processor. It successfully implemented a high level synthesis build automation using the Intel Compiler for SystemC and modularized the project architecture. The created build automation is already in use by ParaNut developers and the CI/CD build chain. The interaction between build chain and CI/CD enables fast and continuous validation of development changes, while the centralized build tools reduce maintenance cost for developers.

The project repository was restructured to be more user-friendly and allow future developers to extend the project by placing newly developed components in meaningful and well-defined directories.

The modular architecture can be used by future developers to provide alternative implementations of ParaNut modules. These modules may include simulation models that significantly improve simulation performance, such as a cache-less memory unit, which is already under development. While such modules can speed up simulation significantly, non synthesizable modules may lead to diverging behavior of hardware and simulation. Using a single core ParaNut and default modules for simulating complex, single threaded software is a viable mean for verifying correct behavior in hardware and simulation.

Working towards the goal of synthesizing the system from SystemC only, the memory module was revised to be synthesizable, unfortunately, the Vivado synthesis process still requires ram modules to be written in Verilog to be synthesized correctly from block RAM cells.

In conclusion, the ParaNut project's build system was updated to be more maintainable and incorporate the high level synthesis using the *ICSC*. Future developers will be able to extend the system with targets for hardware synthesis and hardware flashing and develop alternative module implementations for improved performance and/or additional features. The unified codebase and test bench targets on a module base, together with the CI/CD build chain, will make the project more reliable and simplify hardware debugging. Providing a synthesizable memory unit is an essential contribution towards a fully synthesizable ParaNut processor from SystemC.

References

[1] Alexander Bahle, Gundolf Kiefer, Anna Kerstin Pfutzner, Lutz Vollbrach, "The paranut/risc-v processor - an open, parallel, and highly scalable processor architecture for fpga-based systems," 2020. [Online]. Available: `https://ees.hs-augsburg.de/paranut/paranut-paper-ew2020.pdf` (visited on 01/18/2021).

[2] P. Bachmann, "Build from the end," in *PLoP '13: Proceedings of the 20th Conference on Pattern Languages of Programs*, USA: The Hillside Group, Oct. 2013, pp. 1–12, ISBN: 978-1-94165200-8. DOI: `10.5555/2725669.2725694`.

[3] Christian H. Meyer, "A memory management unit for the paranut," 2022.

[4] Gundolf Kiefer, Michael Seider, Michael Schaeferling, "Paranut - an open, scalable, and highly parallel processor architecture for fpga-based systems," *embedded world Conference 2015*, 2015. [Online]. Available: `https://ees.hs-augsburg.de/paranut/paranut-paper-ew2015.pdf` (visited on 01/18/2021).

[5] *IEEE Standard for Standard SystemC Language Reference Manual*, [Online; accessed 15. Jun. 2023], Jan. 2012. DOI: `10.1109/IEEESTD.2012.6134619`.

[6] G. Kiefer, A. Bahle, C. H. Meyer, F. Wagner, and N. Borgsmüller, *The paranut processor - architecture description and reference manual*, `https://ti-build.informatik.hs-augsburg.de:8443/paranut_developers/paranut/-/blob/develop/doc/paranut-manual.pdf`, (Accessed on 07/19/2023), Jun. 2023.

[7] Marco Milenkovic, "Synthese von systemc-code mit open-source-tools," 2023.

[8] D. H. Martin and J. R. Cordy, "On the maintenance complexity of Makefiles," in *WETSoM '16: Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics*, New York, NY, USA: Association for Computing Machinery, May 2016, pp. 50–56, ISBN: 978-1-45034177-6. DOI: `10.1145/2897695.2897703`.

[9] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *ICSE '11: Proceedings of the 33rd International Conference on Software Engineering*, New York, NY, USA: Association for Computing Machinery, May 2011, pp. 141–150, ISBN: 978-1-45030445-0. DOI: `10.1145/1985793.1985813`.

[10] *Vivado design suite user guide: Synthesis*, [Online; accessed 31. Jul. 2023], Jul. 2023. [Online]. Available: `https://docs.xilinx.com/v/u/2019.2-English/ug901-vivado-synthesis`.

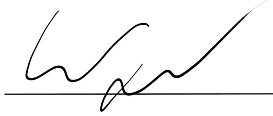
[11] *Vivado/Vitis HLS - SystemC Design Entry for High-Level Synthesis (HLS) is deprecated in 2020.2*, [Online; accessed 15. Jun. 2023], Jun. 2022. [Online]. Available: `https://support.xilinx.com/s/article/73613?language=en_US`.

[12] Xilinx, "Vivado design suite user guide, high-level synthesis," 2021. [Online]. Available: `https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2020_2/ug902-vivado-high-level-synthesis.pdf#nameddest=xApplyingOptimizationDirectives` (visited on 06/15/2023).

Declaration concerning the final project

I, Felix Wagner (Matr. Nr. 2112927), herewith affirm that I have drafted the submitted dissertation independently and have not used any other sources or resources than those declared therein. Sources quoted literally or substantively have been cited according to the acknowledged rules of scientific publication. Furthermore, I declare that the dissertation has not been submitted elsewhere. I have read and understood the explanatory note on fraud in examination procedures at the University of Applied Sciences Augsburg. I affirm that the submitted disserta- tion does not include any plagiarism or any texts or figures created by other persons acting on my instructions.

Augsburg, August 9, 2023



Felix Wagner

A EXU Block Diagram

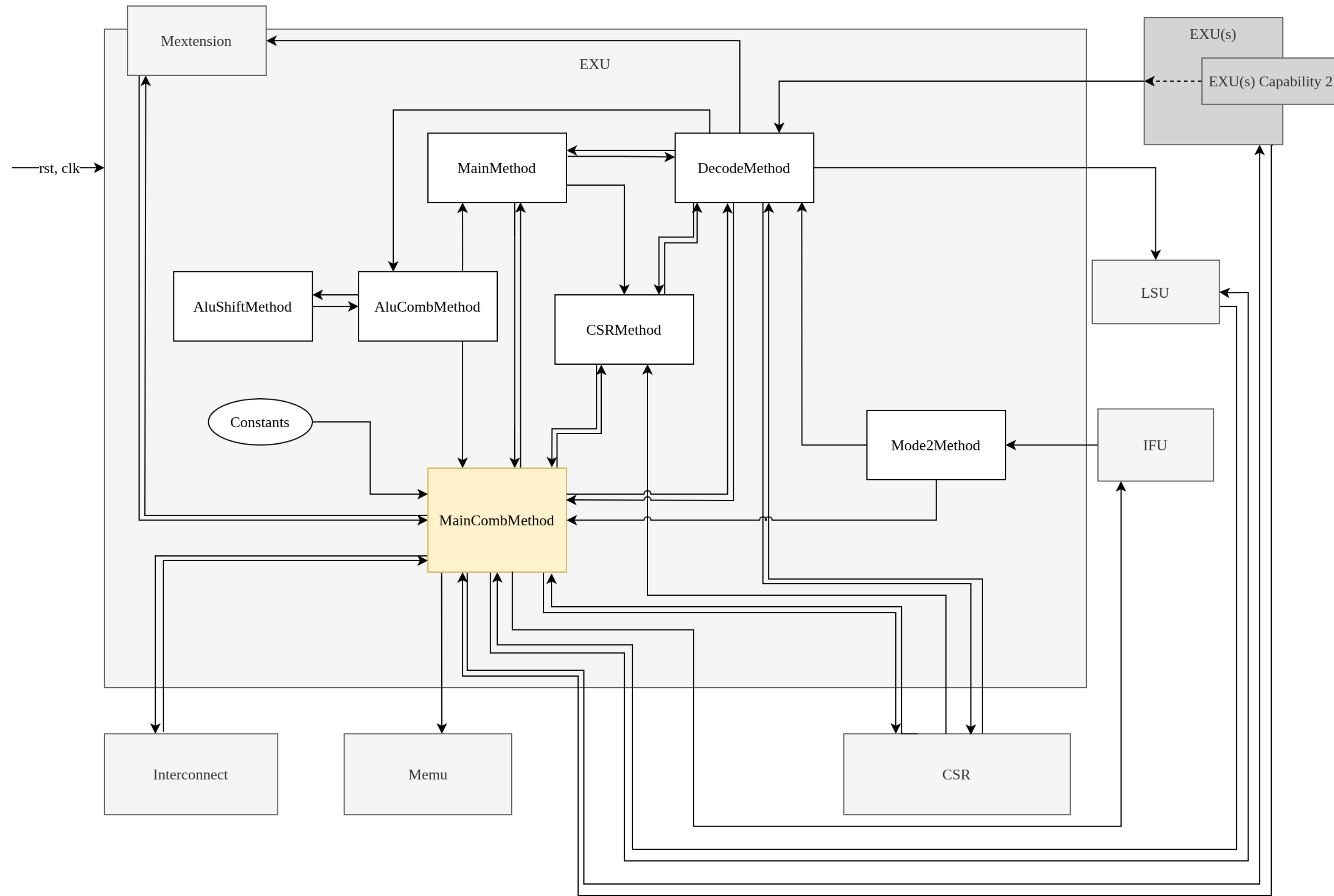


Figure 14: ParaNut Module Architecture

B EXU State Machine

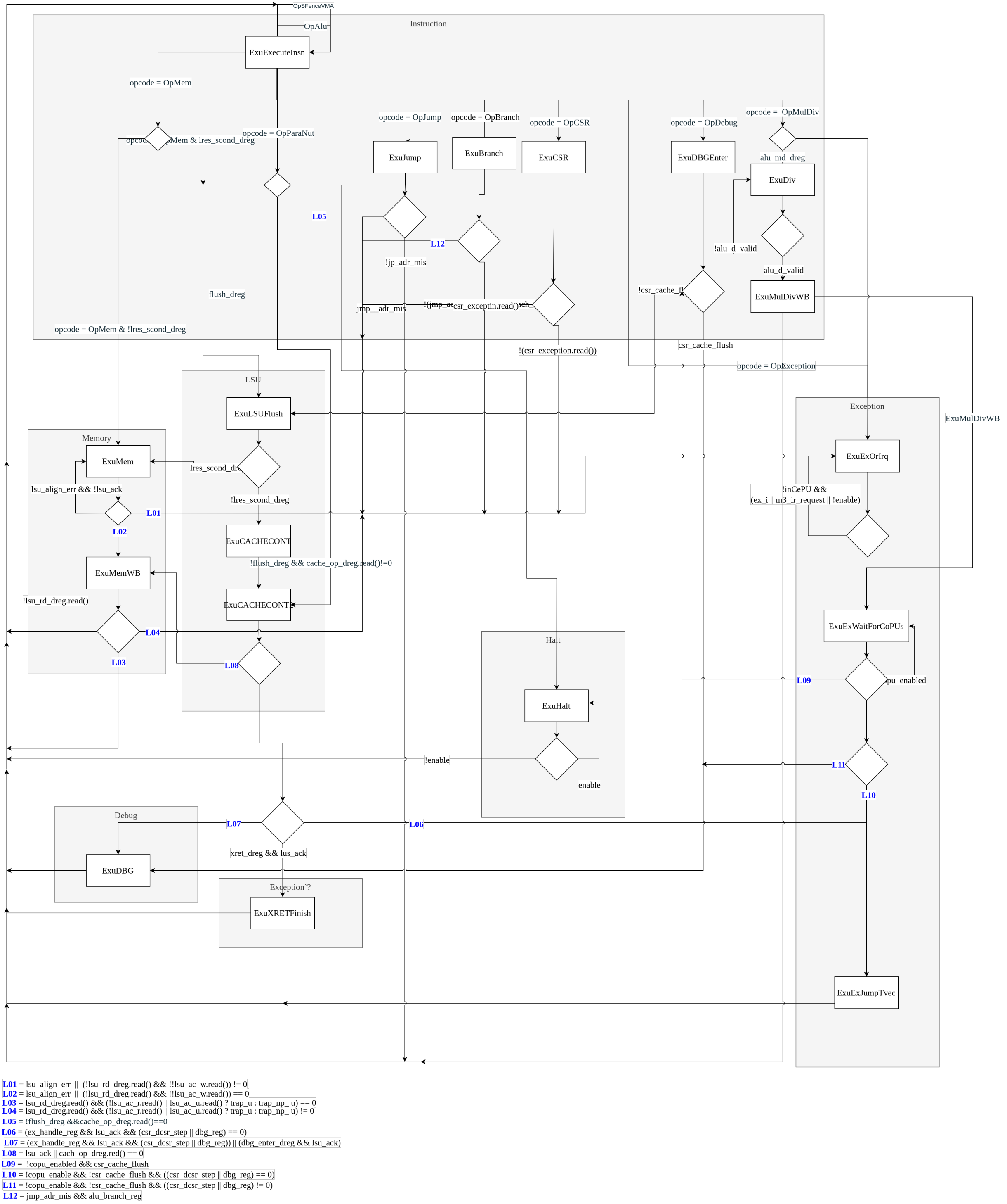


Figure 15: ParaNut EXU State Machine