The Trireme Platform: Getting Started Guide

Computer Architecture & Embedded Systems (CAES) Laboratory
Secure, Trusted, and Assured Microelectronics (STAM) Center, Arizona State University

1 Introduction

The Trireme Platform contains all the tools necessary for register-transfer level (RTL) architecture design space exploration. The platform includes RTL, example software, a toolchain wrapper, and configuration graphical user interface (GUI). All parts of the platform are open-source and available for download at https://www.trireme-riscv.org/index.html The platform is designed with a high degree of modularity. It provides highly-parameterized, composable RTL modules for fast and accurate exploration of different RISC-V based core complexities, multi-level caching and memory organizations. The platform can be used for both RTL simulation and FPGA based emulation. The hardware modules are implemented in synthesizable Verilog using no vendor-specific blocks. The platform's RISC-V compiler toolchain wrapper is used to develop software for the cores. A web-based system configuration (GUI) can be used to rapidly generate different processor configurations. The interfaces between hardware components are carefully designed to allow processor subsystems such as the cache hierarchy, cores or individual pipeline stages, to be modified or replaced without impacting the rest of the system. The platform allows users to quickly instantiate complete working RISC-V multi-core systems with synthesizable RTL and make targeted modifications to fit their needs.

The code base can be accessed through the GitHub repository. The code base directory structure is show in Figure 5.

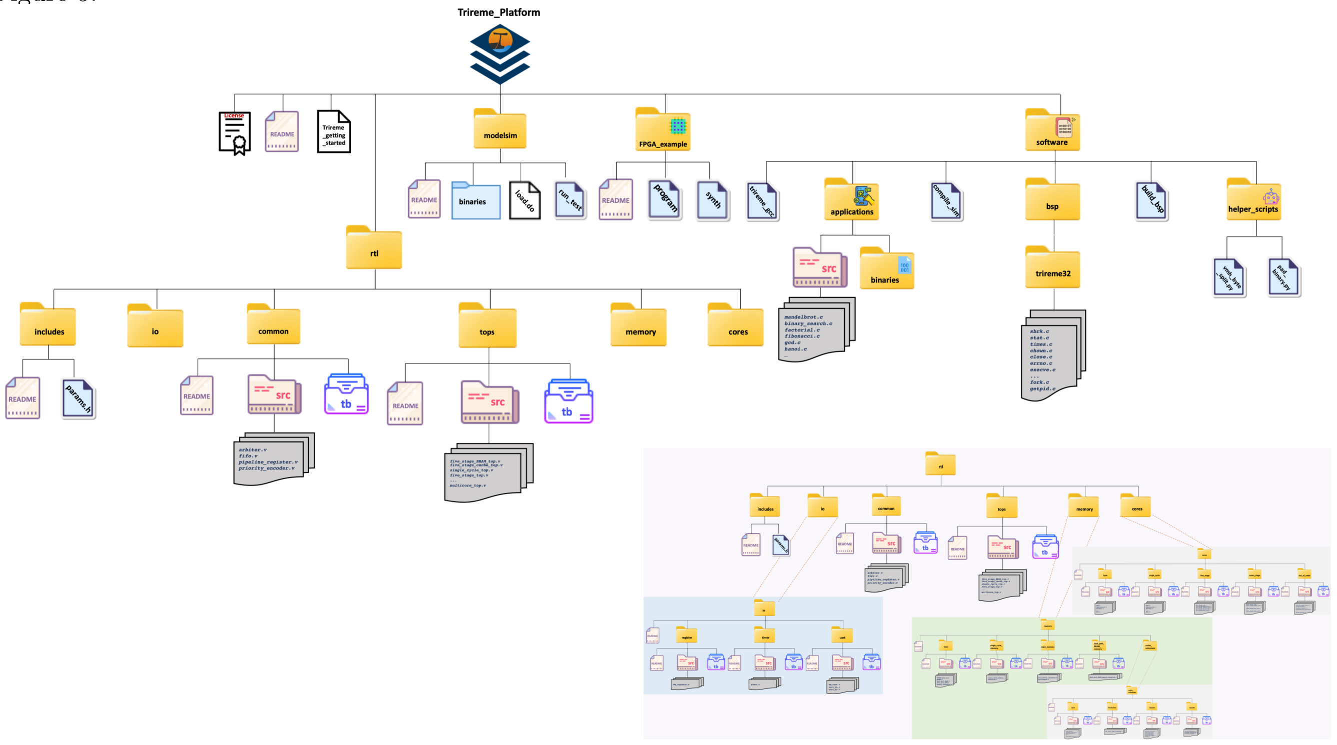

Figure 1: The complete Trireme folder structure.

2 Quick Start

The single-cycle core with FPGA BRAM memory is a good demonstration system to introduce yourself to the Trireme Platform. The single_cycle_BRAM_top module is in the rtl/tops/src directory. Test benches for the whole processor system are in the rtl/tops/tb directory. There are several tests with the name tb_single_cycle_BRAM_top_PROGRAM. Each test bench simulates the execution of a different test program.

The Trireme Platform includes scripts to run simulations in Modelsim from the command line. To use the scripts in the modelsim directory, install Modelsim Altera Starter Edition. The scripts should work with any version of the Altera Starter Edition of Modelsim. We have tested it with version 10.3d released with Quartus II 15.0. To run an example test bench, navigate to the modelsim directory and run:

```
./run_test tb_single_cycle_BRAM_top_gcd
```

If your environment is set up correctly, you should see output similar to the following:

```
# vsim -voptargs="+acc" -batch -quiet tb_single_cycle_BRAM_top_gcd -do "run -all; quit"
# Start time: 16:13:49 on Sep 04,2022
# ** Note: (vsim-3812) Design is being optimized...
# //  Questa Intel Starter FPGA Edition-64
# //  Version 2021.2 linux_x86_64 Apr 14 2021
# //
# //  Copyright 1991-2021 Mentor Graphics Corporation
# //  All Rights Reserved.
# //
# //  QuestaSim and its associated documentation contain trade
# //  secrets and commercial or financial information that are the property of
# //  Mentor Graphics Corporation and are privileged, confidential,
# //  and exempt from disclosure under the Freedom of Information Act,
# //  5 U.S.C. Section 552. Furthermore, this information
# //  is prohibited from disclosure under the Trade Secrets Act,
# //  18 U.S.C. Section 1905.
# //
#
# run -all
#
# Run Time (cycles):          365
#
# tb_single_cycle_BRAM_top (Greatest Common Denominator) --> Test Passed!
#
#
# ** Note: £stop    : ../rtl/tops/tb/tb_single_cycle_BRAM_top_gcd.v(148)
#    Time: 841 ns  Iteration: 0  Instance: /tb_single_cycle_BRAM_top_gcd
# Break in Module tb_single_cycle_BRAM_top_gcd at
#   ../rtl/tops/tb/tb_single_cycle_BRAM_top_gcd.v line 148
# quit
# End time: 16:13:50 on Sep 04,2022, Elapsed time: 0:00:01
```

3 Core Description

3.1 Single Cycle

The single cycle core combines the base **fetch_issue**, **fetch_receive**, **decode**, **execute**, **memory_issue**, **memory_receive**, and **writeback** modules into simple in-order RV32I core. A dedicated control module is used to generate control signals for each submodule in the core. Figure 1 depicts the single cycle core.

The **single_cycle_top**, **single_cycle_BRAM_top**, and **single_cycle_cache_top** modules each use the single cycle core with asynchronous SRAM memory, FPGA BRAM memory, and a cache hierarchy respectively. Several test benches, with different test programs, are included for each top module.

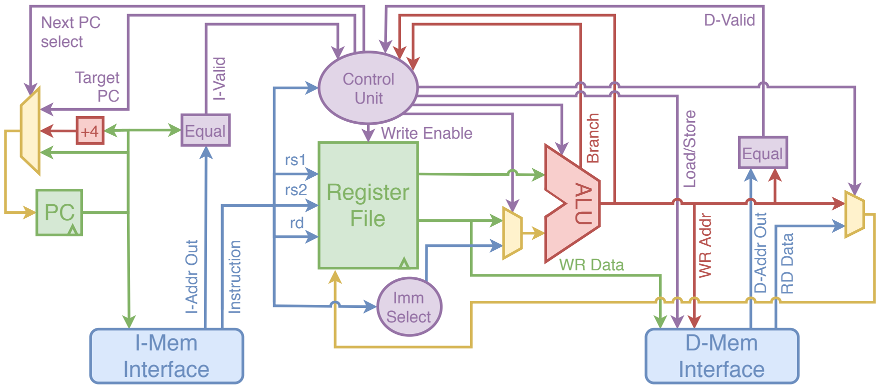

Figure 1: A block diagram of the single cycle core.

To run a top level test bench, execute the run_test script from the single_cycle directory:

```
./run_test tb_single_cycle_BRAM_top_gcd
```

3.2 Five Stage

The Five Stage Core is an RV32i core and includes Fetch, Decode, Execution, Memory, and Writeback stages. The fetch and memory stages each take a single cycle, i.e the instruction and data memory reads must happen combinationally or the core will stall. If BRAM memories or a cache hierarchy are used, the core will stall every other cycle while the BRAM or cache is read. The seven stage core prevents stalls with an additional pipeline stage between issue and receive stages. Figure 2 depicts the five stage core.

The **five_stage_top**, **five_stage_BRAM_top**, and **five_stage_cache_top** modules each use the five-stage core with asynchronous SRAM memory, FPGA BRAM memory, and a cache hierarchy respectively. Several test benches, with different test programs, are included for each top module.

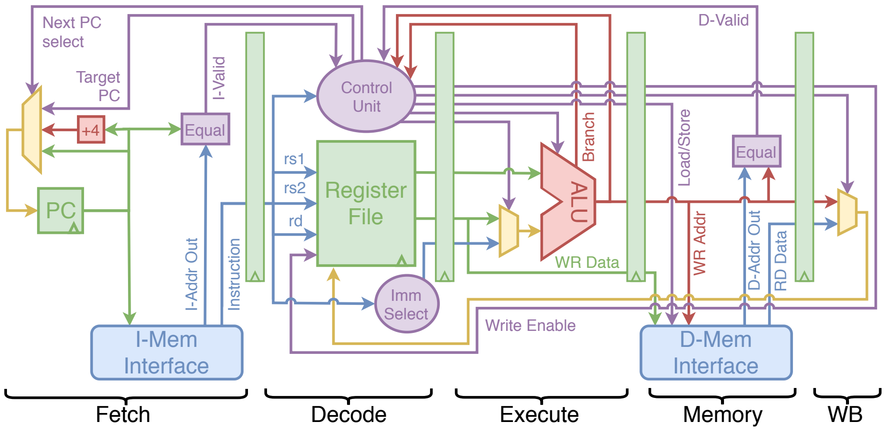

Figure 2: A block diagram of the five stage pipelined core.

To run a top level test bench, execute the run_test script from the five_stage directory:

```
./run_test tb_five_stage_BRAM_top_gcd
```

3.3 Seven Stage

The Seven Stage Core supports both RV32i and RV64i configurations. The 32-bit and 64-bit configurations are selectable with the DATA_WIDTH parameter in the top module or simulation test bench. Figure 3 depicts the seven stage core.

The seven stages include Fetch-Issue, Fetch-Receive, Decode, Execution, Memory-Issue, Memory-Recieve, and Writeback. Using two stages for fetch and memory allows FPGA BRAMs to be accessed without pipeline stalls.

The **seven_stage_BRAM_top**, and **seven_stage_cache_top** modules each use the seven-stage core with FPGA BRAM memory and a cache hierarchy respectively. Several test benches, with different test programs, are included for each top module.

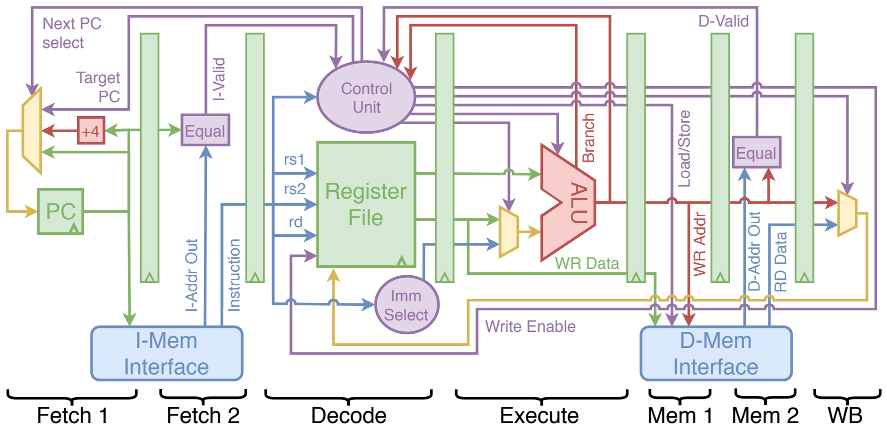

Figure 3: A block diagram of the seven stage pipelined core.

To run a top level test bench, execute the run_test script from the seven_stage directory:

```
./run_test tb_seven_stage_BRAM_top_gcd
```

3.4 Seven Stage Privilege

The Seven Stage Privilege Core supports RV64im and a large subset of the RISC-V privilege specification. Supported privilege modes include M (machine), S (supervisor), and U (user). The ECALL instruction traps as well as timer and software interrupts are supported. Future versions of this core will continue to build support for additional privilege specification features, including virtual memory.

The **seven_stage_priv_BRAM_top** module used the FPGA BRAM memory to build a complete processor system. Several test benches, with different test programs, are included for the **seven_stage_priv_BRAM_top** module.

To run a top level test bench, execute the run_test script from the seven_stage_privilege directory:

```
./run_test tb_seven_stage_priv_BRAM_top_gcd
```

4 Simulating the Trireme Platform with Modelsim

The Trireme Platform includes scripts to simulate the included test benches with Modelsim from the command line. Simulations can also be performed with the Modelsim GUI. The remiander of this section assumes you have Modelsim installed and are comfortable using it.

4.1 Adding Modelsim to your PATH

The included scripts assume that your Modelsim installation is included in your $PATH variable. To add Modelsim to your $PATH variable, execute the following command or add it to your bashrc file.

```
export PATH=$PATH:$HOME/intelFPGA_lite/18.1/modelsim_ase/bin
```

Listing 7: Adding Modelsim to your $PATH variable.

Note that the command above assumes you have the Altera Starter Editoin v18.1 of Modelsim installed in the default location (your home directory). If you have a different version of Modelsim or if you have changed the install location, update the command to include the correct path to the installation's bin directory.

4.2 Compiling Modules in Modelsim

A TCL script is used to compile a test bench and the modules it instantiates. The provided load.do script in the modelsim directory will compile every module and test bench in the Trireme Platform with the appropriate arguments.

By default the TCL variable comple_arg is added to each compile command to provide compiler arguments common to each module. Currently this variable is used to define a Verilog macro that points modules to an include file with more Verilog parameters. If you wish to simulate custom modules with the provided scripts, add the following TCL command to modelsim/load.do to compile the modules and add them to the modelsim library named "work".

```
vlog -quiet $compile_arg relative/path/to/your/modules
```

Listing 8: Compie a custom module in the load.do script

4.3 Simulating Modules

You can simulate any test bench in the Trireme Platform with the run_test script in the modelsim directory. The run_test script must be executed from the modelsim directory. When the run_test script is executed, first, it will execute the load.do TCL script to compile all of the modules. Then the script will simulate all of the test benches entered as command line arguments.

Example usage of the script is shown in the listing below:

```
 # Go to the modelsim directory in the Trireme Platform
 cd modelsim
 # Simulate the tb_single_cycle_top_gcd test bench
./run_test tb_single_cycle_top_gcd
 # Simulate several test benches, one after the other.
./run_test tb_arbiter tb_fifo tb_priority_encoder tb_pipeline_register
```

Listing 9: Example usage of the run_test script

4.4 Including .vmh Files

Make sure to include any necessary memory image (.mem or .vmh) files in the modelsim/binaries directory before simulating a test bench. Modelsim assumes file paths given the to the $readmemh system task are relative to the location Modelsim was launched from. To simulate modules in the Trireme Platform, Modelsim should always be launched from the modelsim directory. The default memory image paths in the included test benches are relative to the modelsim directory.

5 Writing Software for The Trireme Platform

As a complete design space exploration platform, The Trireme Platform provides several example programs and tools to compile them. After a program is written and compiled, it can be executed on the hardware architecture. Currently software must be executed "bare-metal" without an operating system. The Trireme Platform supports a limited subset of the C standard library. Complete standard library support is a work in progress. Dynamic memory allocation is supported with "malloc" and "free" functions. Other standard library features, such as file I/O, have not been implemented yet. Programming architectures with multiple HARTs are supported with multiple "main" functions for each HART. HART (core) 0 executes the "main" function. Additional HARTs in the system will execute a "hartN_main" function where "N" is the HART number. HARTs that fetch instructions from the same memory can be grouped into a single C file with multiple main functions. Example programs are included in the software/applications/src directory.

6 Compiling Software for The Trireme Platform

This section describes how to compile software for The Trireme Platform and assumes you already have the GNU RISC-V tool-chain installed.

6.1 Compiling Software

Compiling software for the Trireme platform is accomplished through the compiler script "trireme_gcc", which is found within the software directory of the Trireme Platform. "trireme_gcc" is a *wrapper* around the RISC-V GCC (GNU Compiler Collection) . As such, the script provides the same interface one would expect from the GCC program.

The Trireme Platform executes software on "bare-metal" meaning an operating system is not used to set up the execution environment for a program. Instead, the compiled binary must include any environment setup necessary to execute the program properly. Programs compiled with trireme_gcc will automatically have the necessary environment setup needed to run on the Trireme platform.

The compiler script supports all arguments that are supported by the GCC program. These arguments are simply forwarded to GCC for compilation, assembly, and linking. In addition, the script allows the user to specify arguments to control compilation for the Trireme bare-metal environment. Executing `./trireme_gcc` or `./trireme_gcc --help` lists the various compiler script specific arguments. For convenience, GCC and compiler script arguments can be intermixed without disturbing GCC invocation.

The Trireme compiler script arguments fall into three categories: memory layout, libraries, and output files. The majority of arguments belong to the memory layout category, which controls the structure of the compiled program. Before using the compiler script, it's important to know where your tool chain is installed on your machine. By default, the Trireme compiler script assumes the RISC-V tool chain is installed in "/opt/riscv". This was specified when the tool chain configuration script was invoked. A different prefix can be specified through the `--toolchain-prefix` argument.

–start-addr Users can set the start address of the program for a Trireme processor by specifying the `--start-addr`. The start address must be in decimal and aligned on a word boundary (i.e., a multiple of 4). If the Trireme compiler script is invoked without specifying this argument, the default start address of 0x0 will be used. For compilation targeting multiple cores, it's important to note that each core will share the same start address.

–stack-addr The program stack address is specified through the `--stack-addr` argument. The stack address is specified in decimal. The stack address must be a multiple of 16 Bytes. In RISC-V, the stack grows from high addresses to low addresses; therefore, this address should take into account the stack size. Trireme does not currently have run-time protection for stack overflow. Therefore, to minimize possible corruption of program data, the stack address should be selected such that there is enough of a gap between the stack end address and the stack start address. For both single core and multi-core configurations, the `--stack-addr` corresponds to the core 0 starting address for the stack. The default value of this argument is 2048. For multi-core configurations, each core is assigned its own stack segment. The compiler script uses the `--stack-size` argument to determine stack start addresses for each core after core 0. Starting from core 0's stack address, each successive core stack address is offset by stack-size bytes from the previous core's stack start address. Values for this argument should be a multiple of 16 and in decimal. The default value for this argument is 2048.

–heap-size The Trireme compiler script places the heap segment at the end of the data section. This cannot be changed by the user. However, the user can specify the size of the heap segment with the `--heap-size` argument. Setting this argument to 0 will tell the compiler script that no heap is needed. Values to this argument should be a multiple of four and in decimal. The default value for this argument is 0.

–num-cores To target a multi-core design, `--num-cores` option is provided. If the number of cores specified is more than one, the compiler script will expect the entry points defined for each core. Please refer to section 5 for more details. By default, the number of cores is set to 1.

–ram-size The main memory size of the target platform is specified through the `--ram-size` as a decimal value argument. By default this value is set to 2048. It is important that this value be large enough to contain all the data and text of the program.

–link-libgloss Trireme implements a Board Support Package (BSP) which acts as a low level Operating System Support Layer. Trireme BSP is in the form of a library that can be optionally linked to programs with the `--link-libgloss` flag. Linking the BSP enables support for the standard C library, including printing to stdout (e.g, puts, printf) and dynamic memory allocation (e.g, malloc, free). Trireme BSP only supports a subset of the POSIX system calls. As such, not all programs can link successfully for the Trireme platform. Before using this flag, the Trireme BSP must be built; please refer to section 6.1.1 for more details.

–{vmh, dump, raw-binary} By default, the compiler script will only generate a ".elf" file. Three other output files supported are ".vmh", ".bin" and ".dump". The ".vmh" file can be used to initialize memory in simulation or FPGA's BRAMs in synthesis. Specifying the argument `--vmh <file path>` will generate a vmh file at the provided file path. The ".dump" file contains contents of an objectdump disassembly which can be useful for debugging. Specify `--dump <file path>` to generate a ".dump" file at the given file path. Specify `--raw-binary` to create a raw binary. The elf headers and metadata are removed from this binary so that it can be directly executed when uploaded to an FPGA implementation with a bootloader program. All three arguments can be specified simultaneously.

–trireme-lib-path The Trireme compiler script can be invoked from directories other than "software". If the compiler script runs from another directory, the Trireme library path must be specified using the `--trireme-lib-path`. By default this points to the "software/library" directory where the BSP will be installed after being built. Note that this is only necessary if `--link-libgloss` is used.

Example 1: To compile a single core gcd.c program in the "software/applications/src" directory with an initial stack pointer of 4096, ram size of 4 KiB, vmh and raw binary output run:

```
./trireme_gcc applications/src/gcd.c -o gcd \
        --stack-addr 4096 \
        --ram-size 4096 \
        --vmh gcd.vmh \
        --raw-binary gcd.bin
```

Listing 10: Compiling a program for a single core processor

Example 2: To compile a dual-core program named gcd_fibonacci.c from the "software/applications/src" directory and generate vmh and raw binary files run the command below. This command will set the stack pointer of core 0 to 4096 and set the stack pointer of core 1 to 8192.

```
./trireme_gcc applications/src/gcd_factorial.c  \
        -o gcd_factorial \
        --stack-addr 4096 \
        --ram-size 4096  \
        --vmh gcd_factorial.vmh \
        --raw-binary gcd_factorial.bin \
        --num-cores 2
```

Listing 11: Compiling a program for a multi-core processor

After a successful compilation, the Trireme compiler wrapper will output a “compilation summary”. Much of the information within the summary aims to prevent possible runtime issues such as stack/heap corruption, instruction memory corruption and irregular program behavior. As shown in listing 12, the compiler wrapper displays the start address of the processor, stack address alignment for each core. In addition to heap size and heap address alignment checks, the compiler wrapper calculates the final program size and provides the recommend main memory size suitable for simulation and FPGA deployment.

```
trireme: compilation summary:
 program path: gcd
 start address: 0 (hex: 0x0000)
 start address on word boundary? yes
 number of cores: 1
 program size: 440 bytes (440.00 B)
 core stack size: 2,048 bytes (2.00 KiB)
 core 0 stack start address: 4,096 (hex: 0x1000)
 core 0 stack end address: 2,052 (hex: 0x0804)
 core 0 stack start address on word boundary? yes
 core 0 stack end address on word boundary? yes
 total stack size: 2,048 bytes (2.00 KiB)
 heap start address: 440 (hex: 0x01b8)
 heap end address: 440 (hex: 0x01b8)
 heap start address on word boundary? yes
 heap end address on word boundary? yes
 heap size: 0 bytes (0.00 B)
 free space size (stack <--> heap gap): 1,612 bytes (1.57 KiB)
 heap region overlaps stack region? no
 memory image total size: 4,096 bytes (4.00 KiB)
 user selected main memory size: 4,096 bytes (4.00 KiB)
 recommended main memory size: 4,096 bytes (4.00 KiB) (bits: 12)
```

Listing 12: Compilation summary output

The compiler wrapper performs several checks on the compiled program. Compilation summary output shows the result of checks in the form of “yes” and “no” answers. The “yes” and “no” answers are colored green for “no action required” and red for “requires attention”. As shown in 12, all checks passed for the gcd program. Although not shown here, the text color of other lines will change to “red” to signify user intervention is required.

6.1.1 Building The Trireme BSP

The Trireme toolchain features low level Operating System support for Bare-metal environments in the form of a board support package (BSP). Low level Operating System Support is implemented as a set of procedures implemented to intercept select POSIX system calls. This allows your application to utilize LibC procedures such as `malloc`. Bare-metal Operating System support can be enabled using the `--link-libgloss` argument.

Building the Trireme BSP is a prerequisite for using the `--link-libgloss` flag. The build process is automated with the help of the “build_bsp” script found within the “software” directory. BSPs are located in the “software/bsp/” directory. Currently only the trireme32 BSP is included. To build a BSP, execute the “build_bsp” script with the BSP name as the first argument. The BSP name is the same as the sub-directory name within the “bsp” directory. Listing 13 shows how to build a BSP using the script.

```
./build_bsp --build trireme32
```

Listing 13: Building the Trireme rv32i Board Support Package

At the end of the build process, the BSP build script will automatically install the BSP library within the "software/lib" directory.

Example 3: Compiling an example program which uses `malloc`

```
./trireme_gcc applications/src/mem.c \
--link-libgloss libnosys_briscv32 \
--ram-size 8192 \
--stack-addr 8192 \
--heap-size 1024
```

7 Trireme Platform Download Folder Structure

The complete Trireme Platform code base can be accessed through its GitHub repository. The code base directory structure is show in the figure below. A concerted effort was made to have README in each subfolder to ease the understanding and usage of hardware modules and the supporting software ecosystem.

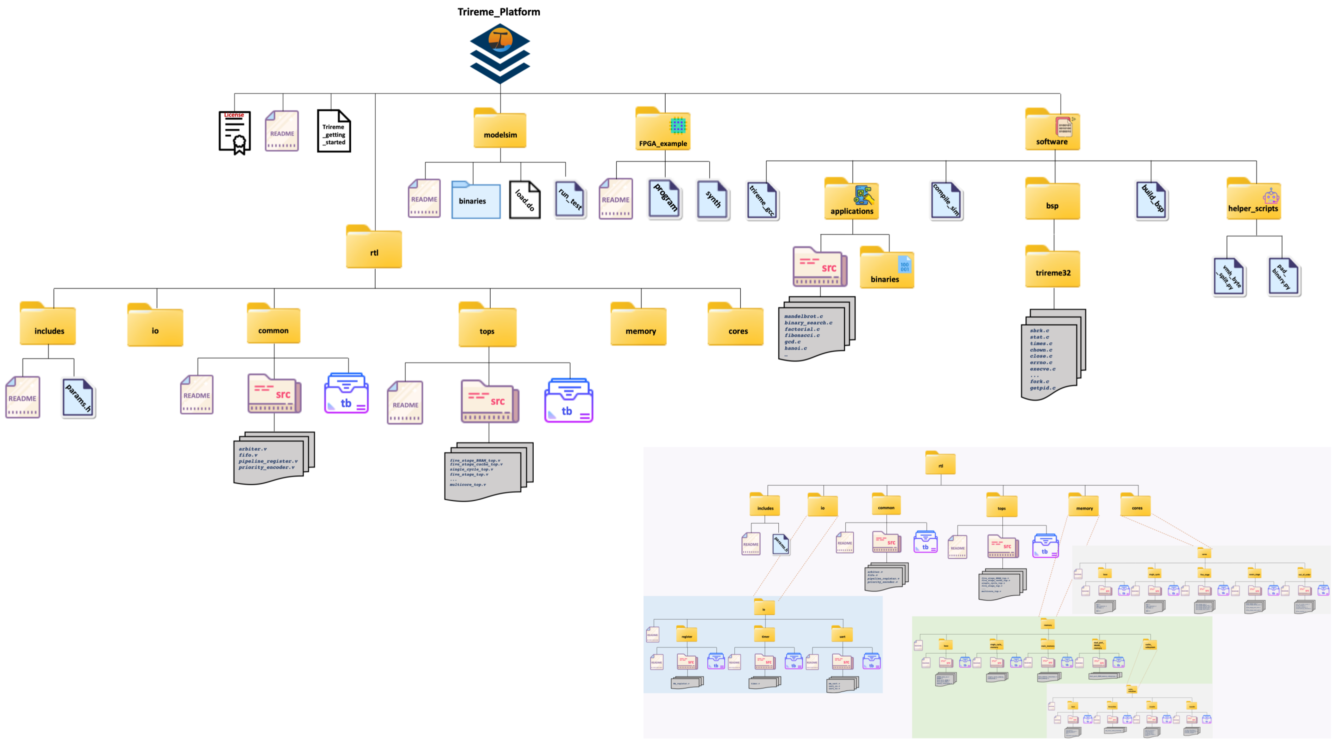

Figure 5: The complete Trireme folder structure.

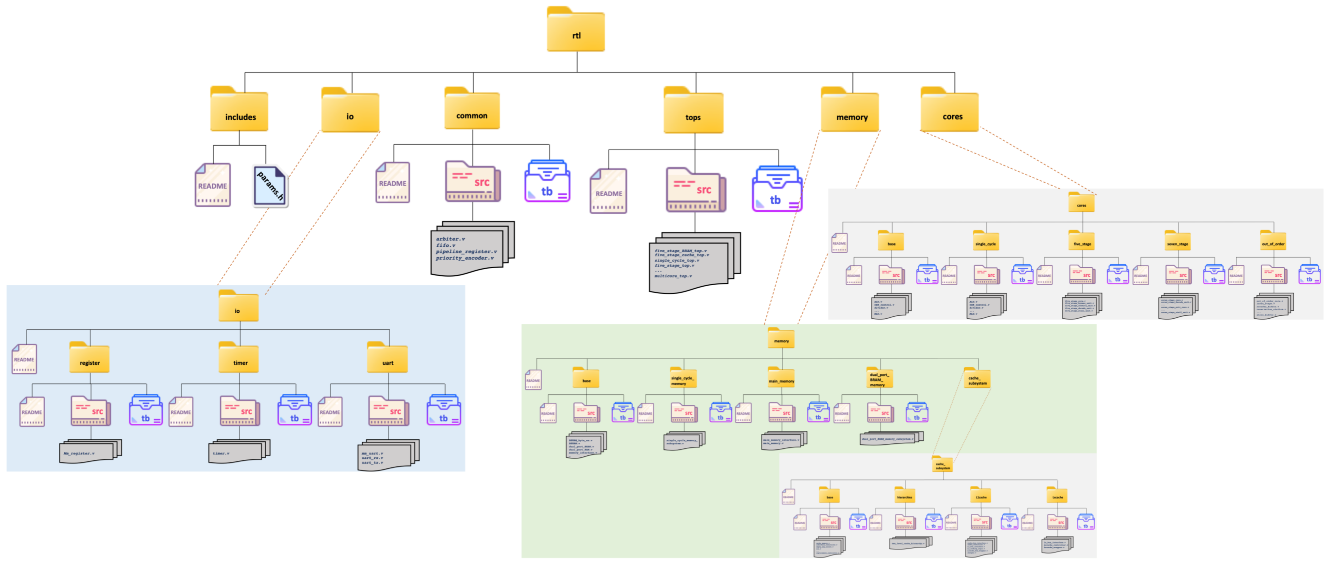

Figure 6: The Trireme hardware module (RTL) folder structure.

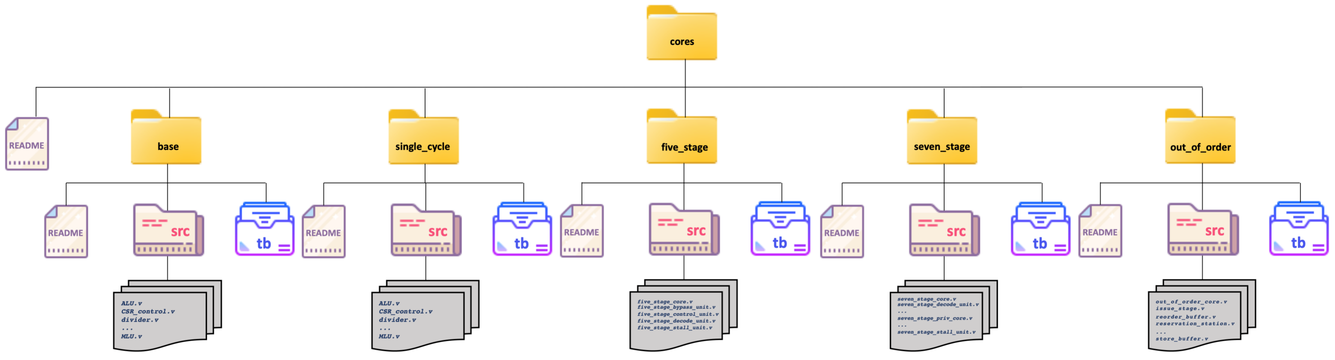

Figure 7: The Trireme cores folder structure.

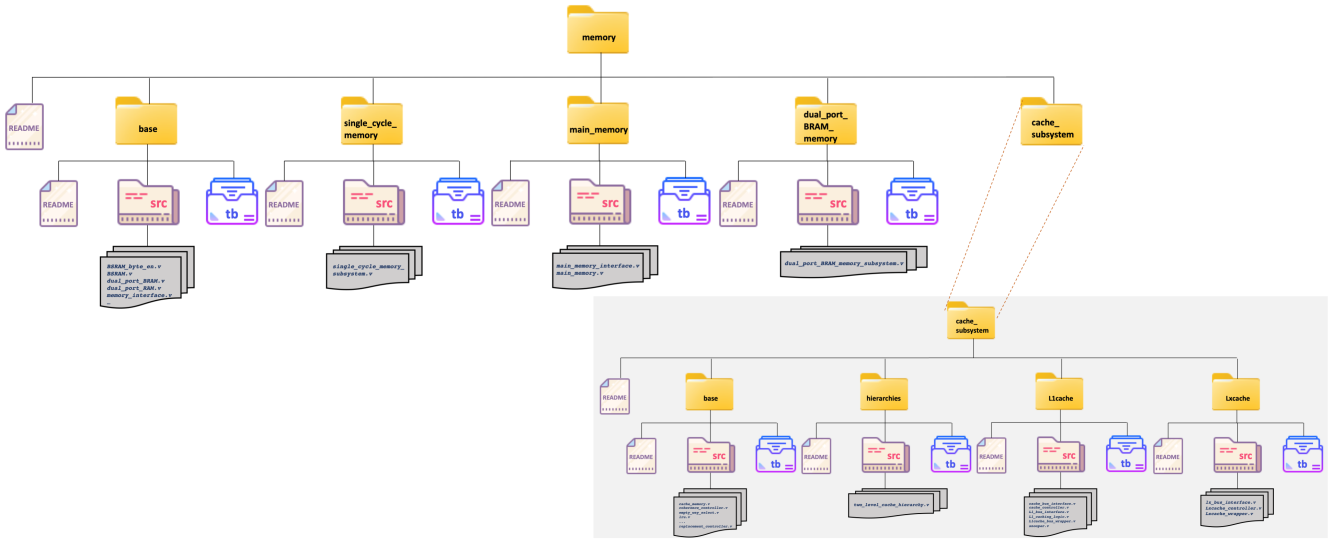

Figure 8: The Trireme memory folder structure.