Politecnico di Torino

Master's Degree Course in Electronic Engineering

Integrated System Architecture

(Course teached by Prof. M. Martina & Prof. G. Masera)

Special Project

RISCV PULPissimo and freeRTOS

Luca Fiore 256891

Marcello Neri 257090

Elia Ribaldone 265613

Luglio 2020

Contents

Abstract

RISC-V is a very versatile processor that can be used in several ways and for different purposes. It is capable to fully support both basic or complex operating systems (such as Linux), if supported by other essential components and peripherals. This makes the RISC-V core well-liked also from an industrial point of view. In the following paper, a first approach on how to port an Operating System on the RISC-V core is presented. Specifically, the FreeRTOS operating system has been analyze and once the porting is complete, the system is ready to execute more complex applications.

1 Introduction

The porting of an Operating System (FreeRTOS in this case) on a processor (RI5CY) mapped on fpga (Xilinx ZCU102) is a complex task which needs some requirements. First of all, it requires a well-stable and complete system, incorporating the processor, capable to allow basic operations, such as memory communication, peripherals support, etc. Then, it requires the knowledge of the hardware composing the system. At the end, it requires the coding of OS files in order to support all the services specified. In this work, the system role is played by the micro-controller PULPissimo (explained in section 3) which comes with some adhoc tools in order to make its porting on fpga possible and prepare the whole environment to build and execute programs.

1.1 Objective

The goal of this paper is to show the method and the results obtained to map PULPissimo on the ZCU102 board, and run the OS on it.
In the first part, it is presented a general overview on the Hardware platform exploited with a main focus on the RI5CY core and the general organization of PULPissimo's structure.
Then, there is an analysis of the PULPissimo software dependencies in terms of Toolchain and SDK, and the introduction of a set of scripts which make the downloading and configuration of all the required files automatic. This allows to exploit correctly PULPissimo's tools. At this stage, the way to load the micro-controller on the ZCU102 board is presented and the execution of a custom application is shown.
Finally, the real OS porting is introduced. Starting from a general overview on FreeRTOS, the steps to have a secure and full porting of the FreeRTOS on PULPissimo platform are described.

1.2 Environment

As explained before, for the realization of this project, some devices and software are required. In particular, the following things have been used:

- Xilinx ZCU102 board;
- microUSB-USB cables;
- JTAG programmer and debugger olimex-arm-usb-tiny-h;
- Ubuntu 18.04 (or Ubuntu 16.04);
- Vivado 2019.2 ;
- GitHub;
- PC with at least 97GB available (85GB for Vivado and 12GB for Pulpissimo project).

2 RI5CY

[1] RI5CY is a small and efficient, 32-bit, in-order RISC-V processor core with a 4-stage pipeline written in SystemVerilog. The ISA of RI5CY was extended to support multiple additional instructions including hardware loops, post-increment load and store instructions and additional ALU instructions that are not part of the standard RISC-V ISA.
In Figure 2, the block diagram of the core is depicted. It is possible to see the main components in each stage of the pipeline, as well as the interface with memory.

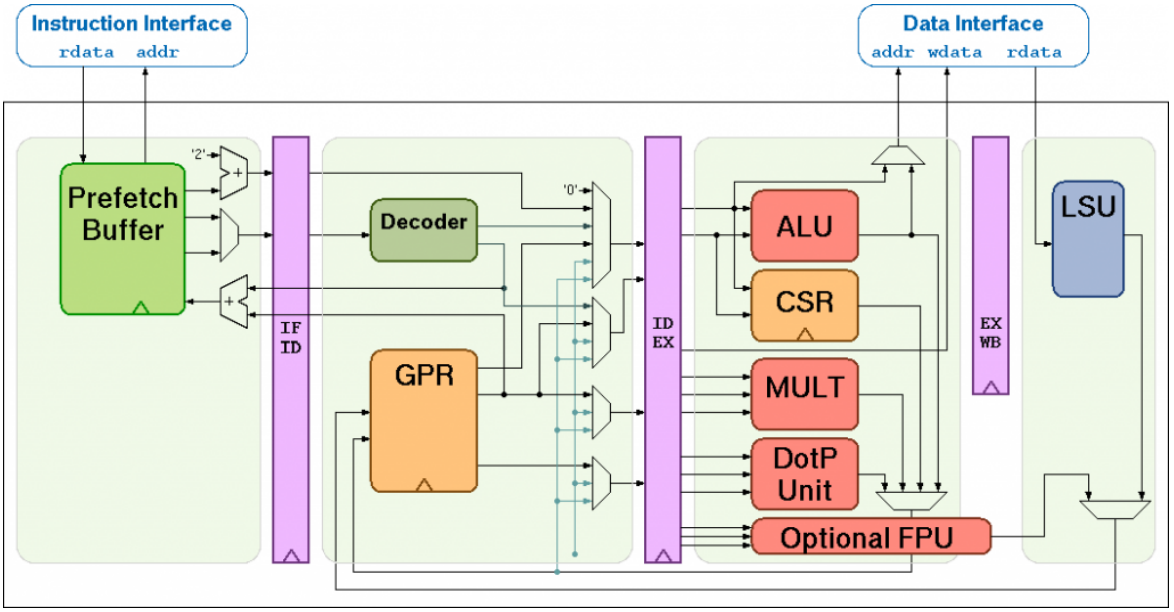

Figure 1: RI5CY block diagram

As it is possible to notice in the previous scheme, a main difference with other RISC-V cores is the reduction of the number of pipeline stages. A higher number of pipeline stages allows for higher operating frequencies, increasing overall throughput, but also increases data and control hazards, which, in turn, reduces the IPC (Instructions Per Cycle). So, this is a key design decision to avoid stalls and have an higher IPC.
One of the peculiarity of this RISC-V core is the support of compressed instructions. They are 16 bits wide, as opposed to the usual 32 bit RISC-V instructions, allowing a great reduction of code size, if possible. Another peculiarity is the optional availability of a single-precision Floating Point Unit (FPU).

2.1 Pipeline

The 4-stage pipeline is composed by Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), and Write Back (WB) operations. This is different from the traditional pipeline in RISC-V cores which includes another stage, the Memory Access (MEM) operation, between the EX and WB.

2.1.1 Instruction Fetch

In this stage, the fetch of the instructions from instruction cache or instruction memory is executed as usual, but with the peculiarity that the instruction address must be half-word-aligned (a word is 32 bits wide) due to the support of compressed instructions. These compressed instructions replace the normal instructions with a one to one

mapping. The use of compressed instructions is possible only when some conditions occur, such as the immediate or address offset is small, one of the registers is the zero register (x0), the destination register and the first source register are identical, or other cases, and an ad-hoc decoder is required.
To achieve optimal performance, a prefetcher is used which fetches instruction (32 bits instruction or 128 bits cache line) from the instruction memory, and send it to the compressed decoder. If these bits hold a compressed instruction, then it outputs the decompressed variant of the instruction, and signals to the decode stage that there is a compressed instruction.

2.1.2 Instruction Decode

In this stage, the instruction arrived from the IF is decoded with the purpose of setting up signals to properly control the execution stage. So, based on the current instruction, operands can be read from the register file and signals can be generated to select which logic unit is needed in the EX stage.
The register file can be latch- or flipflop-based, it is a user choice. While the latch-based register file is recommended for ASICs, the flip-flop based register file is recommended for FPGA synthesis, although both are compatible with either synthesis target. Obviously, the flip-flop based register file is significantly larger than the latch-based one for an ASIC implementation. The register file has 32 locations (x0 to x31) 32 bits wide, from x1 to x31 there are normal locations (read and write are allowed), while x0 is a special location that can be only read. If the FPU is instantiated, another register file with 32 locations (f0 to f31) is used. In the case of synthesis on FPGA, the register file is FF-based.

2.1.3 Execution

As regard the execution stage, many operations can be performed by the ALU according to the opcode. The ISA supported by the RI5CY are **I** (Base Integer Instruction Set), **C** (Standard Extension for Compressed Instructions), **M** (Integer Multiplication and Division Instruction Set Extension), **F** (Single Precision Floating Point Extensions), and other PULP specific extensions. Indeed, the RI5CY core operations supported are:

- bit-manipulation;
- addition/subtraction;
- multiplication (32bit x 32bit with 32bit output) (it requires 4 cycles to complete);
- division/remainder (it may requires from 2 to 32 cycles to complete);
- multiply-accumulate;
- fixed-point operations, such as saturating, normalizing and clipping;
- (optionally) floating point operations.

At this stage, the data memory can be accessed for a write operation (by means of a "store" instruction) or can be simply pointed by the address (calculated by the ALU or arrived from the instruction) in order to read its content at the WB stage.

2.1.4 Write Back

This is the last stage of the pipeline. Here data are written back into the register file. Data to be written can arrive either from the EX stage or from the data memory (by means of a "load" instruction).
The core follows a protocol called LSU (Load Store Unit) to communicate with memory (a more detailed explanation is provided in the Manual User of RI5CY). Here, this unit takes care of accessing the data memory, which supports "load" and "store" on words (32 bit), half words (16 bit) and bytes (8 bit).
Moreover, as regards the memory, an additional protection unit can be optionally enabled. It is the Physical Memory Protection (PMP) Unit which allows the core to possibly run in USER MODE . Every fetch, load and store access executed in USER MODE are first filtered by the PMP unit which can possibly generated exceptions.

2.2 Interrupts

RI5CY core supports also interrupts and exceptions on illegal instructions and on PMP filtered requests (note that PULPissimo does not support USER MODE interrupts). Interrupts can only be enabled or disabled globally and not individually. That task is eventually committed to an interrupt controller outside the core. That controller is assumed also to handle multiple interrupts requests. On the contrary, exceptions are always active.
When an interrupt/exception arrives, the core saves the current Program Counter (PC) and the Status into specific registers (depending on the working mode) and then it jumps to the corresponding handler using the content of a another specific register as base address. After the handler has finished, the core restores the PC and the Status previously saved and resumes the execution of applications.

3 PULP and PULPissimo

PULP (Parallel Ultra Low Power) platform is a project started by Integrated Systems Laboratory (IIS) of ETH Zürich and Energy-efficient Embedded Systems (EEES) group of the University of Bologna in 2013, in order to achieve an efficient architecture for Parallel Ultra Low Power Processing and so improve SoC performance keeping low power consumption. Starting from these ideas, PULP project has been developed as a series of Hardware and Software IPs, which create the PULP platform and are used to create PULPino and PULPissimo microcontroller architecture; we decided to use PULPissimo microcontroller because it is the latest architecture released and the evolution of PULPino.

3.1 PULPissimo Architecture

PULPissimo is a single-core platform characterized by a consistent completeness and complexity. In fact, it provides either the RI5CY core or the Ibex one as main core (in this project, the RI5CY core is selected), autonomous Input/Output subsystem (uDMA), new memory subsystem, support for Hardware Processing Engines (HWPEs), new simple interrupt controller, new peripherals and new SDK. PULPissimo also supports I/O on interfaces such as: SPI (as master), I2S, Camera Interface (CPI), I2C, UART and JTAG.
The very interesting feature of this platform is the possibility of integration of hardware accelerators (Hardware Processing Engines) that share memory with the RI5CY core and are programmed on the memory map. This allows to personalize the system with special functions, eventually extending the Instruction Set Architecture.

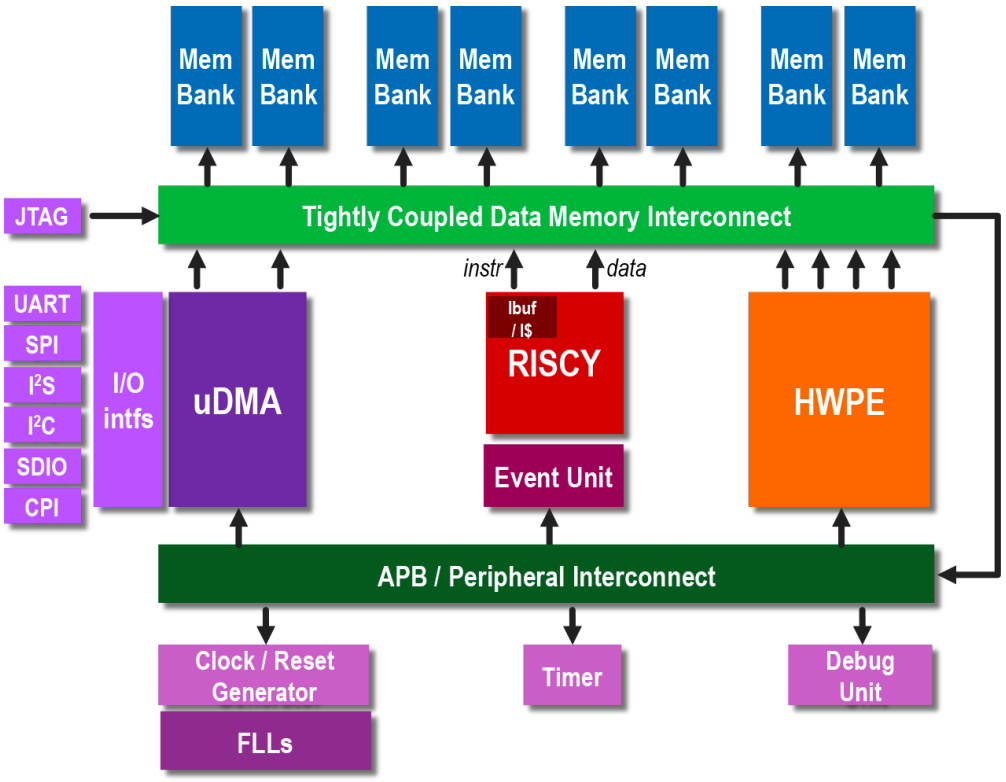

Figure 2: PULPissimo architecture

3.2 PULPissimo environment

PULPissimo platform can be exploited in different ways. It is possible to build the RTL simulation platform, to build the virtual platform or to map the system on FPGA. The latter option is supported only for few models of FPGA, specifically the mapping on the ZCU102 board is what has been done in this project. In order to build the platform, independently on the purpose, a specific environment needs to be prepared. Fortunately, an ad-hoc Software Development Kit (SDK) is provided together with the RISCV gcc-toolchain. The installation of these tools is a very important step to work on PULPissimo platform but before this, the installation of all the libraries and dependencies on the machine has to be performed.
In the repository https://github.com/RISKVFT/AutomatePULPissimo there are some script that can be **installed and used to configure and run application on pulpissimo platform**, all scripts that will be mentioned are refferred to this repository. After the preparation of the environment, the user can build the platform intended for its purpose and test it using some test files provided with the platform itself.

3.3 Configuration of PULPissimo

As explained in the previous subparagraph, the first step is to prepare the PULPissimo environment installing all the necessary tools (SDK, toolchain and others).
We start the creation of PULPissimo project in a **virgin Virtual Machine with Ubuntu 16.04.4 LTS 64-bit and 120GB** of free disk space. Since the installation require many different PULP tools installation with the proper dependencies, we decided to create a bash script (called **pulp_script/pulp_install.sh**) that automatizes the installation process and gives the user the possibility to personalize the configuration (depending on the purpose and on the tools used).
The script is structured in a modular and error-tracking way, meaning that the user can decide which tool to install and can fix eventual errors occurred thanks to log files generated during the process. In the script, some custom functions are used in order keep the code more readable and more user-friendly and many useful function are sourced by **lib/ccommon.sh** bash file.
Here there are the steps executed by the script:

- **pulp-riscv-gnu-toolchain installation -** This toolchain is available at https://github.com/pulp-platform/pulp-riscv-gnu-toolchain. It provides the RISCV C and C++ cross-compiler. After the installation of all the necessary Ubuntu (or CentOS) dependencies, the script clones the git repository of the toolchain and configures the correct cross-compiler according to the option -c|–cross-compiler set as command line argument. With the -t|–test-suite command line argument, instead, the user can decide if install or not the DejaGnu test suite for RISC-V, that is a front-end to write tests for any program with a portable interface [11]. In this step, the environment variables **PULP_RISCV_GCC_TOOLCHAIN** and **VSIM_PATH** are set. The installation directory of the riscv toolchain is "/opt/riscv/", so this is the value assigned to PULP_RISCV_GCC_TOOLCHAIN variable. Instead, VSIM_PATH is set to "(pulpissimo_root)/pulpissimo". Obviously, also other minor variables are set in this step. All these env variables are also added to /.bash_profile file so as to make them permanent every time a new bash shell is created.

- **PULP SDK installation:** The PULP SDK is available at https://github.

com/pulp-platform/pulp-sdk.git and after the installation of all the dependencies (such as specific version of Python) the repository is cloned. In this stage, target and platform (rtl simulator, fpga, virtual platform) are chosen, so that the SDK can be built properly. This is a very important step, because the SDK allows to compile and run applications. To do this, it needs to be initialized by sourcing the "sourceme.sh" file which is inside the installation folder. Also in this stage, other env variables are set and saved permanently.

- **PULP Builder installation:** The PULP Builder repository is available at https://github.com/pulp-platform/pulp-builder.git and is used to build application for pulp project. After the clone of the repository the builder is configured.

- **PULPissimo installation:** PULPissimo project repository is available at https://github.com/pulp-platform/pulpissimo.git. This is the most important step, since the PULPissimo project contains all the files required to build the rtl simulator or to map the system on fpga. The repository also contains all the files of IPs which compose the PULP system, including the RI5CY core files, and documentation, fpga support, testbench, simulation and examples.
  Files related to the ZCU102 will be used later in another script aimed to facilitate the fpga configuration process and the generation of the bitstream to flash into the board.

- **PULPissimo example:** Example of application project can be found at https://github.com/pulp-platform/pulp-rt-examples.git. The script downloads this repository and build hello application by default (it is possible to change the application to build). These example applications will be used later in another script for the fpga.

- **Virtual platform installation:** This is the last step that has been added for completeness. This step prepares the PULP environment for the virtual platform.

Another important feature of the script is the capability of run the script without starting necessarily from the beginning, this idea was developed due to many installation error that was encountered during script creation, indeed some errors required, once solved, the continuation of the script from where it was blocked due to that error. With **-p|–part_install [0|1|2|3|4|5]** option is possible to:

- **0** start from scratch
- **1** start after the toolchain
- **2** start after the sdk
- **3** start after pulp-builder
- **4** start from test (hello)
- **5** only virtual platform

So we could use this command to install all pulpissimo project with dependencies:

```
pulp_install -v -c pulp -t y
```

The installation can require some hours and at the end of installation there will be six new directories: **log** for log and error file, **pulp-sdk**, **pulp-rt-examples**, **pulp-riscv-gnu-toolchain**, **pulpissimo** and **pulp-builder**. Now we are ready to mapping PULPissimo on board and run application using another script.

3.4 PULPissimo mapping and loading app on ZCU102 board

After having installed all the tools required by the PULPissimo environment, it is possible to generate all the files required to map PULPissimo into the fpga (the ZCU102 board) and load the compiled application (also for these operations bash script has been created, called **pulp_app.sh**). This operation needs some prerequisites. In fact, this step requires the installation of Vivado (the version 19.2 works well, while the version 20.1 not, and note that this software requires about 70GB available on the pc) and the availability of USB cables and JTAG adapter.
The script pulp_app.sh is composed by many parts, each one aimed to carry out a particular step. To choose the operation to execute, command line arguments need to passed when launching the script. The possible operations, with the related argument, are:

- **Generation of pulpissimo bitstream [-s]:** This operation aims to generate the PULPissimo bitstream to be loaded into the board. Note that two files are generated in this step: pulpissimo_BOARD.bit and pulpissimo_BOARD.bin (BOARD can be setted with -o). The first one is the bitstream file for JTAG configuration of the FPGA and it is cancelled when the board is turned off. The last one is the binary configuration file to flash to a non-volatile configuration memory. The generation of these files is carried out by means of Vivado. However, a copy of these files is stored into the Git repository in case the user encounters errors in generating them, because for example he can use only a recent version of Vivado (20.1 or later). This script now supports only download of .bit file so .bin file is unused.

- **Download of pulpissimo bitstream into the board [-d]:** This operation requires a USB-microUSB cable to be connected between pc and board (ports J2 for zcu102, their function is explained in the next subparagraph). Obviously, also this step requires Vivado. Errors may occur during the process because of pc configuration. Some of them have been faced, solved, and explained in pulp_app man page. Anyway, at the end of this process, PULPissimo is loaded into the fpga and ready to be used.

- **Build SDK and openOCD patch [-b]**:This action create the BOARD.sh file in pulp-sdk/configs/fpgas/pulpissimo directory and source it in order to configure the SDK for the FPGA platform, then the configs/pulpissimo.sh file is source to select correct microcontroller (pulpissimo), finally the makefile in pulp-sdk directory is used to build the SDK. Also the json file pulp-sdk/pulp-config/configs/fpgas/BOARD.json is generated and used in building. Finally all environment variable setted are saved in ./.environ.env file.

- **Compilation of custom application [-c C_APPDIR]:** This operation allows to compile the example applications at ./pulp-rt-examples/*C_APPDIR*, for the RISCV core used in PULPissimo. This step requires the SDK and the riscv-gcc-toolchain to be correctly installed. After this step, the ./pulp-rt-examples/*C_APPDIR*/build/pulpissimo/test/test file should be generated.

- **Testing the application by means of terminals [-t T_APPDIR]:** This is the final operation to carry out when everything is ready, both fpga configuration and application compilation. For this step, another connection between pc and fpga (port J55) needs to be done using JTAG programmer/debugger, we use olimex-arm-usb-tiny-h connector. This is necessary in order to allow the loading of the application bitstream file into the core memory and to debug it. During this stage, the user is able to test and verify the correctness of the application by means of the **gdb** terminal (which makes the debugging possible) and the **screen** terminal (which shows the output of the application).

- **Export variables [-e]:** ./.environ.env file is used to save this environment variable in ~/.bash_profile file, in this way all variable will be available in a whatever shell. This could be useful if user want to execute pulpissimo make and script without using pulp_app script.

- **selection of USB ports [-u USB|all|i]:** This is an auxiliary operation for the user in order to manage USB, these are the possible argument:

  - **USB:** USB should be a valid usb name in /dev directory, for example:

    ```
    pulp_app -u ttyUSB0 -t
    ```

    is a valid command if exist /dev/ttyUSB0. The usb name should be the name of the usb connected to fpga UART used by PULPissimo application as default io peripheral. This usb will be screen in a terminal by -t action using:

    ```
    screen $USB 112500
    ```

  - **all:** Supposing that user has the USB 3.0 4-port HUB, with this option will be screen all usb from ttyUSB0 to ttyUSB5 in 5 different terminal.

  - **i:** This option is useful to see all usb connected and their name in order to choose correct usb to use as USB argument.

- **Board selection [-o BOARD]:** This option is used to define the board to use, BOARD variable is used in -b, -c and -t action. If you also use -r|–connector option, to set connector, be careful because BOARD option should be set before!! otherwise default board will be used. For example:

  ```
  pulp\_app -o zcu104 -r olimex-arm-usb-tiny-h -b -c -t
  ```

  is a valid command, while:

  ```
  pulp\_app -r olimex-arm-usb-tiny-h -o zcu104 -b -c -t
  ```

  does not work as you want since olimex-arm-usb-tiny-h.cfg file is searched in ./pulpissimo/fpga/pulpissimo-zcu102/ path since "zcu102" is the default board.

- **JTAG programmer [-r CONNECTOR_NAME]:** This option set the name of the .cfg file, to use as openOCD configuration, which is usually the same name of JTAG programmer used. This config file should be in ./pulpissimo/fpga/pulpissimo-$BOARD/ directory.

Using previous explained command we could perform all operation using this simple command:

```
pulp_app -s -d -b -c "hello" -o zcu102 -r olimex-arm-usb-tiny-h
        -u ttyUSB2 -t "hello"
```

Summing up, previous commands perform this operation:

- **-s** : generate bitstream files of PULPissimo;
- **-d** : download bitstream into zcu102 board selected with -o option;
- **-c "hello"** : compile ./pulp-rt-example/hello/test.c application creating ./pulp-rt-example/hello/build/pulpissimo/test/test elf file;
- **-t "hello"** : open a series of terminals for debugging:
  - **openOCD:** In the left-up terminal openOCD is runned using olimex-arm-usb-tiny-h.cfg configuration file.
  - **gdb:** In the right-up terminal is runned gdb that immediately run ./pulpissimo/fpga/pulpissimo-zcu102/elf_run.gdb file and wait for user command.
  - **UART:** In the bottom terminal it's shown the data received from ttyUSB2 usb.

3.4.1 Board connections

The ZCU102 board should be connected with three cables to computer, for this purpose we use **USB 3.0 4-Port Hub** that occupy only one usb port of the computer and can drive up to four usb port. The connection with the board are:

- **USB/JTAG/UART** micro-USB connector (J2): This is used to download the bitstream into fpga.
- **USB/UART** micro-USB connector (J83): This port is used to UART connection of PULPissimo with the computer.
- **Olimex_OpenOCD_JTAG_ARM-USB-TINY-H** programmer and debugger (J55): This device is used to load the application into core's memory and debug PULPissimo's core that has its own JTAG interface. Indeed, J83 port is connected to FPGA JTAG, while PULPissimo JTAG is mapped on J55 connector. It is important to observe that this connection can be made by using other different JTAG adapters, such as Digilent JTAG-HS2 or OLIMEX ARM-USB-OCD-H that are those ones suggested by the original guide. Anyway, each adapter needs its own configuration file to setup the communication. Following PULPissimo guide [12], we configured the correct connection between JTAG_ARM-USB-TINY-H and PULPissimo JTAG:

JTAG Signal	FPGA Port	J55 Pin	ARM-USB-TINY-H pin
tms	PMOD0_0	Pin 1	Pin 7
tdi	PMOD0_1	Pin 3	Pin 5
tdo	PMOD0_2	Pin 5	Pin 13
tck	PMOD0_3	Pin 7	Pin 9
gnd	GND	Pin 9	Pin 6-20
vdd	3V3	Pin 11	Pin 1

We should take attention in ARM-USB-TINY-H connection since the ribbon

cable flips the pin order. The pin swapping is reported in Figure 3:

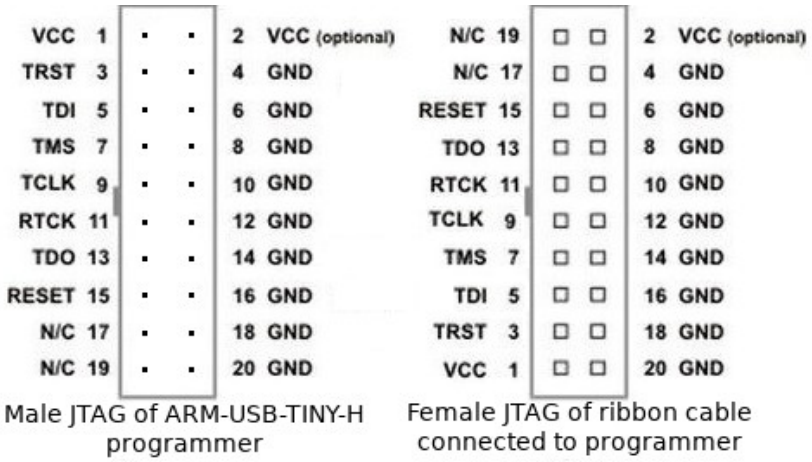

Figure 3: JTAG pins

In order to avoid to manually connect pins to the J55 port every time we test the board (that is source of errors), we created an handmade soldered connector between ribbon cable and J55 connector Figure 4.

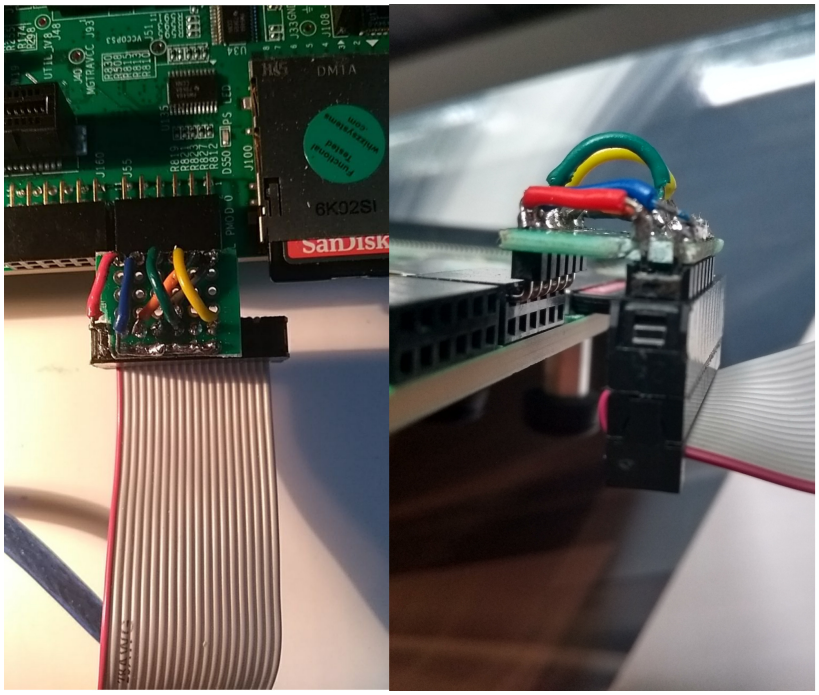

Figure 4: Handmade connector between J55 and ARM-USB-TINY-H ribbon cable

4 FreeRTOS

Note: in this FreeRTOS analysis the word task is used with the same meaning of thread.

4.1 The kernel

[4] FreeRTOS is a real time kernel (real time scheduler) developed by Real Time Engineers Ltd. used in various microcontroller/microprocessor environments to ensure a secure and strong base for any real time application.
Common applications usually includes hard and soft deadlines that require a specific managing by the OS.
Hard deadline missing can cause disastrous or very serious consequences so validation is essential: can all the deadlines be met, even under worst case scenario?
Soft deadline should be met for maximum performance. The performance degrades in case of deadline misses.
FreeRTOS allows applications to be treated as a set of independent threads. If the HW platform has more than one processor more than one thread can be executed in parallel. The scheduler organize threads in queues by examining the priority assigned to each thread by the application designer. In the simplest case, the application designer could assign higher priorities to threads that implement hard real-time requirements, and lower priorities to threads that implement soft real-time requirements. This would ensure that hard real-time threads are always executed ahead of soft real-time threads, but priority assignment decisions are not always that simplistic.

4.1.1 Why use a RTOS?

It is known that a good application can be run on a processor without any sort of OS. Anyway having a strong software layer of operating system has many advantages:

1. simpler application code:
   infact the timing information are now in the kernel domain that provides a time-related API to the application.

2. Maintainability/Extensibility:
   Abstracting away timing details results in fewer interdependencies between modules, and allows the software to evolve in a controlled and predictable way. Also, the kernel is responsible for timing, so application performance is less susceptible to changes in the underlying hardware.

3. Improved efficiency:
   Using a kernel allows software to be completely event-driven, so no processing time is wasted by polling for events that have not occurred. Code executes only when there is something that must be done.
   Counter to the efficiency saving is the need to process the RTOS tick interrupt, and to switch execution from one task to another. However, applications that don't make use of an RTOS normally include some form of tick interrupt anyway.

4.2 The porting

FreeRTOS supports many different platform including some RISCV based microcontrollers. So, to develop a good porting for PULPissimo we started from one of the pre-configured example projects available on the FreeRTOS web page [5] and from two projects available on GitHub [6] [7].

4.2.1 Files organization

FreeRTOS is made by a set of C source files. Some of them are required by all ports while others are specific to a port.
In Figure 5 there is a summary of the FreeRTOS file organization.

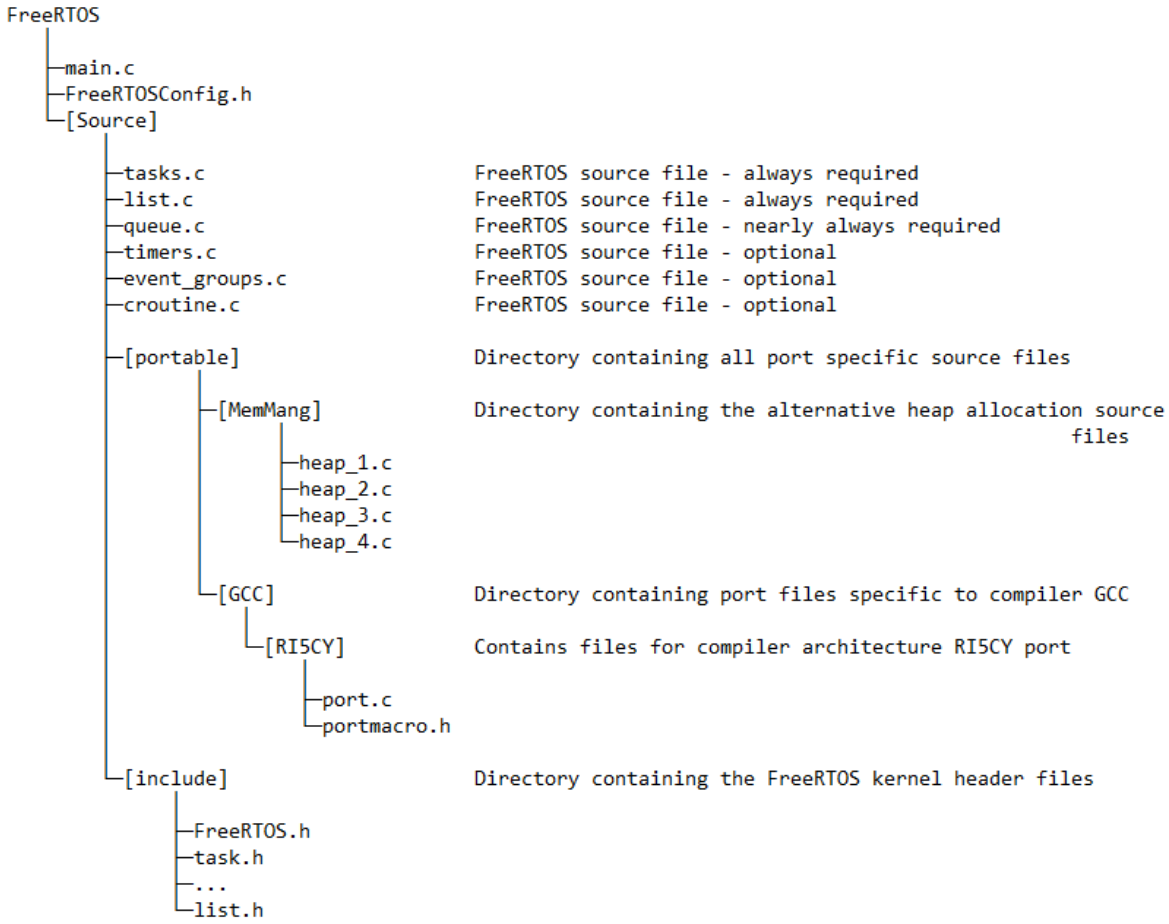

Figure 5: File organization

- main.c

  This is the custom application that can be organized as a set of task function and a main() function. The main() call the tasks with xTaskCreate() [10] and start the scheduler with the directive vTaskStartScheduler() [9].

- FreeRTOSConfig.h

  [8] is the configuration file needed by FreeRTOS to know how to manage a specific application. Actually it is a set of define that configure the OS based on the specific HW and application requirements.
  For example the macro configUSE_PREEMPTION that defines whether the

co-operative or preemptive scheduling algorithm will be used or the macro configCPU_CLOCK_HZ that defines the frequency in Hz at which the internal clock that drives the peripheral used to generate the tick interrupt will be executed.

- Source

  this directory contains the source files required by all the portings and some optional source files:

  - tasks.c, list.c and queue.c

    compose the kernel and they are always required in all portings.

  - timers.c

    provides software timer functionality. It needs only to be included in the build if software timers are actually going to be used.

  - event_group.c

    provides event group functionality. It need only be included in the build if event groups are actually going to be used.

  - coroutine.c

    is a deprecated feature.

  - Portable

    this directory contains source files specific to a FreeRTOS port.

  - include

    is the directory containing all the header files required by the kernel.

In the Portable directory there are two main folders: MemMang regarding the dynamic memory management and [Compiler]/[architecture] to configure FreeRTOS correctly for the given hardware and compiler, in our case is GCC/RISCV.

4.2.2 Build FreeRTOS for RISCV

[5] To build FreeRTOS for a RISC-V core you need to:

1. Include the core FreeRTOS source files and the FreeRTOS RISC-V port layer source files in your project as described in the previous section.

2. Ensure the assembler's include path includes the path of the header file that describes any chip specific implementation details.
   In /FreeRTOS/Source/Portable/[compiler]/RISC-V/chip_specific_extensions directory an additional header file called freertos_riscv_chip_specific_extensions.h has to be included in the assembler path. It describes chip specific details, and is required because RISC-V chips often include chip specific architecture extensions.

3. Define either a constant in FreeRTOSConfig.h or a linker variable to specify the memory to use as the interrupt stack. The memory to use as the interrupt stack can either be defined in the linker script or declared within the FreeRTOS port layer as a statically allocated array. The linker script method is preferred on memory constrained MCUs as it allows the stack that was used by main() prior to the scheduler being started (which is no longer used for that purpose after the scheduler has been started) to be re-purposed as the interrupt stack.

4. Define configMTIME_BASE_ADDRESS and configMTIMECMP_BASE_ADDRESS in FreeRTOSConfig.h. If the target RISC-V chip includes a machine timer (MTIME) then set configMTIME_BASE_ADDRESS to the MTIME base address and configMTIMECMP_BASE_ADDRESS to the address of the MTIME's compare register (MTIMECMP). Otherwise set both definitions to 0.

5. For the assembler, define portasmHANDLE_INTERRUPT to the name of the function provided by your chip or tools vendor for handling external interrupts.

6. Install the FreeRTOS trap handler. The FreeRTOS trap handler is called freertos_risc_v_trap_handler() and is the central entry point for all interrupts and exceptions. The FreeRTOS trap handler calls the external interrupt handler when the source of a trap is an external interrupt.

References

[1] RI5CY User Manual Rev 4.0

[2] Sam Leonard, "A dive into RI5CY core internals"
www.embecosm.com/2019/08/13/a-dive-into-ri5cy-core-internals/

[3] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini: "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices", IEEE

[4] Richard Barry: 161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide

[5] FreeRTOS - RISC_V:
www.freertos.org/Using-FreeRTOS-on-RISC-V.html#RISC_V_QUICK_START

[6] PULPINO:
https://github.com/pulp-platform/pulpino/tree/master/sw/apps/freertos

[7] Greenwaves GAP8:
https://github.com/GreenWaves-Technologies/freeRTOS

[8] FreeRTOSConfig.h:
https://www.freertos.org/a00110.html

[9] vTaskStartScheduler():
https://www.freertos.org/a00132.html

[10] xTaskCreate():
https://www.freertos.org/a00125.html

[11] DejaGnu:
https://www.math.utah.edu/docs/info/dejagnu_1.html#SEC2

[12] pulpZcu102Guide:
https://github.com/pulp-platform/pulpissimo/tree/master/fpga/pulpissimo-zcu102