POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Multithreaded support Embedded Application on RISC-V

Supervisors

Prof. Alessandro SAVINO

Prof. Michele PORTOLAN

Prof. Ernesto Edgar SANCHEZ SANCHEZ

Candidate

Mert Dogan ARISOY

2020-2021

Abstract

In recent years, RISC processors regain their prominence by dint of RISC-V. The main benefits of RISC-V which are flexibility, accessibility, and royalty-free structure, have made RISC-V an undeniable open-source rival against semiconductor companies. Those advantages bring RISC-V is a decent option for bare-metal embedded systems that the systems without any operating system support or centralized kernel. Also, for the bare-metal systems, concurrency and parallelism have been challenging and contradictory issues, to solve them multi-threading can be an option, and POSIX threads are the first term that comes to mind when multi-threading is mentioned. This thesis proposes to provide POSIX threads to the RISC-V bare-metal embedded systems. To exploit POSIX threads, musl has been chosen which is a C standard library implementation. Regarding the results of musl's bare-metal execution, new approaches are proposed and discussed. The experimental results have reported and discussed for the PULP platform and the other RISC-V cores. The overall conclusion and probable future works are reported at the end of the thesis.

Acknowledgements

First of all, I would like to thank my mother Emine, my father Sedat, and my brother Berk who have meritorious contributions for coming these days.

I would like to express my sincere gratitude to my thesis advisors of Prof. Alessandro Savino, Prof. Ernesto Edgar Sanchez Sanchez, and Prof. Michele Portolan for allowing me to conduct this thesis topic under their supervision.

Table of Contents

List of Tables

List of Figures

Listings

Acronyms

APB

Advanced Peripheral Bus

API

Application Programming Interface

CPU

Central Processing Unit

CSR

Control and Status Register

FIFO

First-In First-Out

I/O

Input Output

IoT

Internet of Things

ISA

Instruction Set Architecture

PIE

Position Independent Executable

POSIX

Portable Operating System Interface for Unix

PULP

Parallel Ultra Low Power

RAM

Random Access Memory

RISC

Reduced Instruction Set Computer

RTOS

Real-Time Operating System

SDK

Software Development Kit

SoC

System-on-a-chip

uDMA

Autonomous Input/Output subsystem

Chapter 1

Introduction

1.1 Premise

Bare-metal systems are known as a system without the support of any operating system or centralized kernel, the instructions execute on the CPU directly. Due to a lack of operating system functionality, compared to the fully-fledged systems, any application requires more operation i.e. I/O management, memory management, start-up, scheduling, etc. in the development section to run successfully. Nowadays, it usually convenient for embedded systems. Therefore, close-to-hardware languages i.e. C, C++, and assembly language are preferred in bare-metal programming. A bare-metal embedded application can be dependent on developers' software structure approach although the simple base model can be shaped around of infinite main loop which can perform actions. Those actions can vary and this thesis' work examines multi-threading support for those. Multi-threading is the ability to execute multiple threads, concurrently. This parallelism can be provided by instruction-level parallelism or thread-level parallelism. Modern systems' architecture has already combined these two parallelism techniques by dint of operating system support and standard C libraries, but multi-threading in bare-metal systems are challenging and generally will be provided by architecture-specific or platform-specific solutions. Besides, there would be significant constraints for multi-threading on bare-metal systems which would the general constraints for bare-metal development. Depending on preference, there are several processor architectures that can be chosen for bare-metal embedded systems that can eliminate several constraints for the application and the system. This study examines bare-metal multi-threading on RISC-V processors. RISC-V is an open-standard ISA based on RISC principles, and apart from the others i.e. ARM, Intel; the ISA is provided under the open-source licenses. RISC-V ISA has been designed considering a wide range of usability with the advantages of compactness, efficiency and, low power consumption.

1.2 Work Introduction

The idea of the thesis starting on a lightweight musl standard C library and port it to bare-metal RISC-V PULP platform in order to exploit POSIX threads. PULP platform is developed in collaboration between ETH Zurich and the University of Bologna, aimed to build new efficient architectures and systems for ultra-low-power processing to meet the computational demands of IoT applications. PULP platform contains various micro-controllers like PULP, PULPino, PULPissimo, and etc. for wide application areas and they have different but similar RISC-V cores that will be introduced in the following sections.
As mentioned before there are several limitations in bare-metal environments, lack of operating system, and thus resource management can be counted as one of the major inadequacies. This thesis's work is aimed to port musl to the PULP platform and RISC-V processors, and proposes alternative solutions in order to provide multi-threading.
In the context of bare-metal multi-threading, there are several solutions that have been put forward to provide multi-threading or multi-tasking, the most common way is to use an RTOS for a platform. It should be noted that PULP platforms are small and lightweight embedded systems. Besides, can be inferred from its name it consumes ultra-low-power thus the executed program should be lightweight and satisfied in terms of performance. However, an RTOS allows users to create a limited task and uses heavy system resources. Therefore, using an RTOS would not be a feasible solution for the PULP platform or light RISC-V processors.

Chapter 2

RISC-V Architecture

This chapter is aimed to introduce some fundamental concepts and topics related to this thesis work's point of view. This thesis focus on RISC-V architecture and its ISA. Then, this chapter will introduce RISC-V PULP and SiFive processors family. RISC-V PULP platform contains three different cores which are RI5CY, Ariane and, Zero-riscy however this thesis topic is working on RI5CY and Zero-riscy. As the name PULP suggests, they offer a low-energy consumption system and thus this chapter also examines correspondent cores in the SiFive processor family to the PULP platform's cores.

2.1 RISC-V ISA

As mentioned before, this work examines multi-threading support for embedded applications on RISC-V processors. So, RISC-V ISA has an important part in this work. RISC-V ISA is an open-standard ISA based on RISC principles with bringing advantages to RISC processors and it has a modular and flexible form which means any supported instruction by the core set can be added or detachable. Furthermore, anyone can create their own custom instruction set for a custom RISC-V processor in order to offer top-notch solutions for dedicated applications. Nowadays, RISC processors and therefore RISC-V have gained great importance with the enhancement of compilers. This section describes RISC-V base integer ISA which must be present in any implementation, and RISC-V extensions ISA. Thanks to the open-standard license, anyone can extend, add, remove any instruction from the instruction set depending on their application area.

2.1.1 Base Integer ISA

The RISC-V base integer ISA (RV32I-RV64I) is essential for each implementation and consists of forty instructions that can emulate modern operating system environments' functionality although it has minimal implementation. It has two variants which are RV32I and RV64I depending on the architecture. This section is focused on RV32I which is used in 32-bit systems. Base integer ISA has 32 integer registers (x0-x31) whose bits wide is 32-bit. Also, there is a reduced version of RV32I is called RV32E, the only difference is that RV32E reduced the number of integer registers to 16(x0-x15), where x0 is dedicated to a zero register both in RV32I and RV32E. In a nutshell, RV32I contains load and store instructions, arithmetic operations instructions, control transfer instructions, CSRs operation instructions, memory ordering instructions and, etc.
There are four standard types of RV32I base instruction formats which are R, I, S, and U. Besides, to handle immediate operations there are further two instruction format variants which are B and J. The whole RISC-V instruction formats are shown in Figure 2.1.

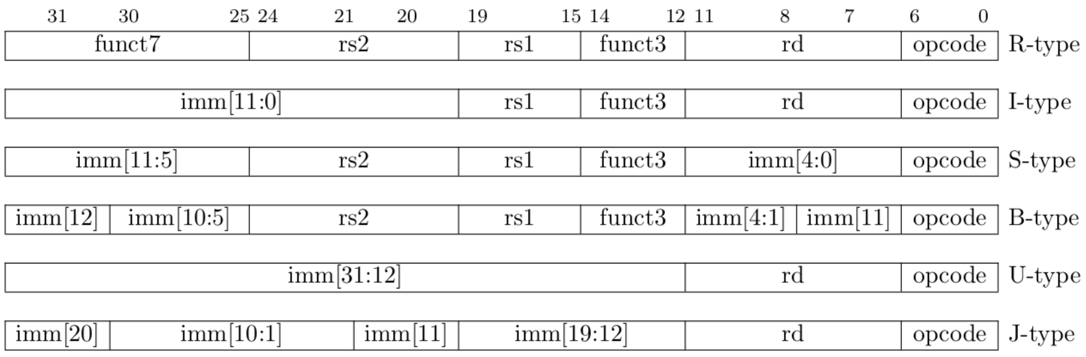

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type

Figure 2.1: RV32I Instructions Format

For better understanding the concept that will introduce in the following chapters, it will be convenient to describe several base integer instructions, according to [1], these instructions can be explained as follows,

Load Instructions

`Syntax: <LOAD> rd, offset(rs2)`

Load instructions have a standard format as shown in Figure 2.2. The opcode can

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

Figure 2.2: RV32I Load Instructions Format

be `lb, lh, lw, lbu,` and `lhu`. Depending on the opcode, it loads an X-bit value from memory to its first operand(rd) without any atomicity.

Store Instructions

```
Syntax: <STORE> rs2, offset(rs1)
```

31 — 25	24 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

Figure 2.3: RV32I Store Instructions Format

Same as load instructions store instructions also have a standard format as shown in Figure 2.3, but different than load instructions format. The opcode can be `sb, sh,` and `sw`. Depending on the opcode, it stores X-bit values from the low bits of register rs2 to memory without any atomicity.

jal Instruction

```
Syntax: jal rd, offset
```

The meaning of the `jal` abbreviation is "Jump and Link". As understandable from its name, it is an unconditional jump and it jumps to an address and places a return address in rd. The instruction format of the `jal` instruction is shown in Figure 2.4

31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
	offset[20:1]			dest	JAL

Figure 2.4: RV32I jal Instruction Format

fence Instruction

`Syntax: fence pred, succ`

31 28	27	26	25	24	23	22	21	20	19 15	14 12	11 7	6 0
fm	PI	PO	PR	PW	SI	SO	SR	SW	rs1	funct3	rd	opcode
4	1	1	1	1	1	1	1	1	5	3	5	7
FM		predecessor				successor			0	FENCE	0	MISC-MEM

Figure 2.5: RV32I fence Instruction Format

When a code compiled, some optimizations can be applied to instructions ordering, for example, load instructions can be carried at the top of the program even the programmer loads the variable at any place of the program but especially in the concurrent programming, it can be led some concurrency pitfalls. Thus these memory barriers such as FENCE instruction and its variants can be used to put a barrier in order to guarantee instruction orderings. Briefly, it used to order memory accesses and I/O orderings.
According to the RISC-V manual [1], the unused fields in the fence instruction which are `rs1` and `rd` are reserved for finer-grain fences for future improvements.

2.1.2 Extensions ISA

As mentioned before, RISC-V ISA has a modular structure. Thus, any supported instruction set can append or detachable. RISC-V extensions ISA is designed to expand RISC-V functionality and provides additional operations. There are many RISC-V extensions instruction set, however, this sub-section describes the extensions set which are examined in this thesis's work. Those are, "M" standard extension, stands for integer multiplication and division; "A" standard extension, stands for atomic instructions; "F" standard extension, stands for single-precision floating-point instructions.

RV32M - Integer Multiplication and Division Instruction Set

RV32M contains instructions that can operate standard integer multiplication and division operations between two integer registers. The RISC-V designers have separated these instructions from the base integer instruction set because multiplication and division operations either infrequent or better handled for specific purposes. More detailed information and instructions format can be found in RISC-V manual[1]

RV32C - Standard Extension for Compressed Instructions

RV32C instructions are used to reduce static and dynamic code size by adding short 16-bit instruction encodings for common operations.[1] RV32C instructions increase the compactness of the code and allows memory efficiency.

RV32A - Atomic Instructions Set

From [2], atomicity may have many meanings in computer science, but one can define atomicity as an "indivisible operation" in this thesis. So, in this context, one can say that atomic memory operations provide indivisible memory access operation.
RV32A contains instructions that can operate read, modify, write operations, atomically. In order to avoid pitfalls such as deadlocks and provide synchronization between multiple RISC-V harts, RV32A guarantees mutual exclusion.
There are mainly two-instructions for atomic load and store operations.

31 27	26	25	24 20	19 15	14 12	11 7	6 0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
LR.W/D	ordering		0	addr	width	dest	AMO
SC.W/D	ordering		src	addr	width	dest	AMO

Figure 2.6: RV32I Atomic Instructions Format

`Syntax: lr.w rd, rs1`

`lr.w` instruction loads a word from the address in `rs1` to `rd`, different from load instructions which are described in section 2.1.1, it registers a reservation on the addressed word's memory address.

```
Syntax: sc.w rd, rs1, rs2
```

`sc.w` instruction conditionally stores the word in `rs2` to the address in `rs1`, this condition depends on whether the reservation is still valid and the reservation set contains the bytes that have been written.

RV32F - Single-precision Floating-point Instruction Set

This instruction set contains the instructions which are capable to operate floating-point computational operations with single-precision.

2.2 RISC-V PULP

PULP Platform is developed by a collaboration between ETH Zurich and the University of Bologna, aimed to develop energy-efficient processors and microcontrollers. PULP platform contains RISC-V cores such as RI5CY, Ariane, and Zero-riscy; single-core microcontrollers of PULPino and PULPissimo; and multi-core processors such as PULP (also called OpenPULP). For the context of this thesis, this section describes RI5CY and Zero-riscy cores, then introduces the PULPissimo microcontroller.

RI5CY Core

RI5CY[3] is a four-stage in-order 32-bit RISC-V processor core whose core architecture shown in Figure 2.7. It can be as main core in the PULPissimo. The supported instruction set can be ordered as follows,

- RV32I Base Integer Instruction Set
- RV32M Integer Multiplication and Division Instruction Set Extension
- RV32C Standard Extension for Compressed Instructions
- RV32F Single-precision Floating Point Extensions (Optional full support)

Zero-riscy Core

RI5CY[4] is a two-stage in-order 32-bit RISC-V processor core, it can be called as a child of the RI5CY core. It can be as main core in the PULPissimo and its core architecture shown in Figure 2.8. The supported instruction set can be ordered as follows,

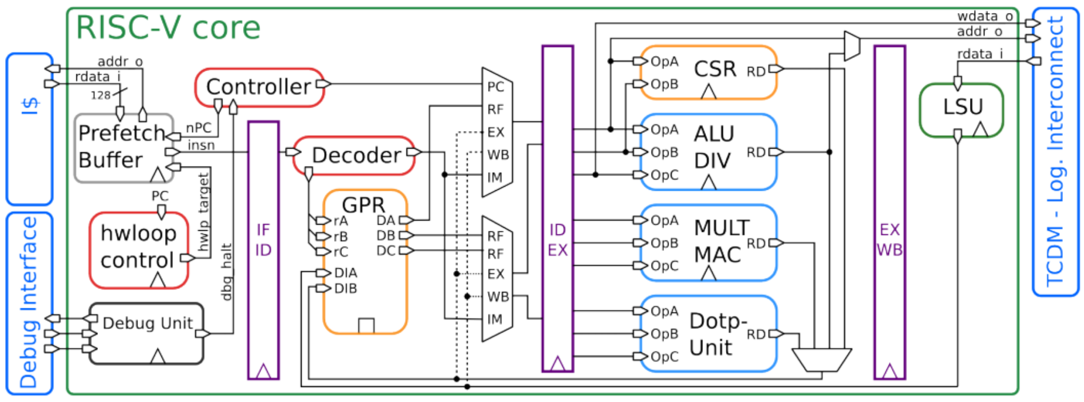


Figure 2.7: RI5CY Core Architecture

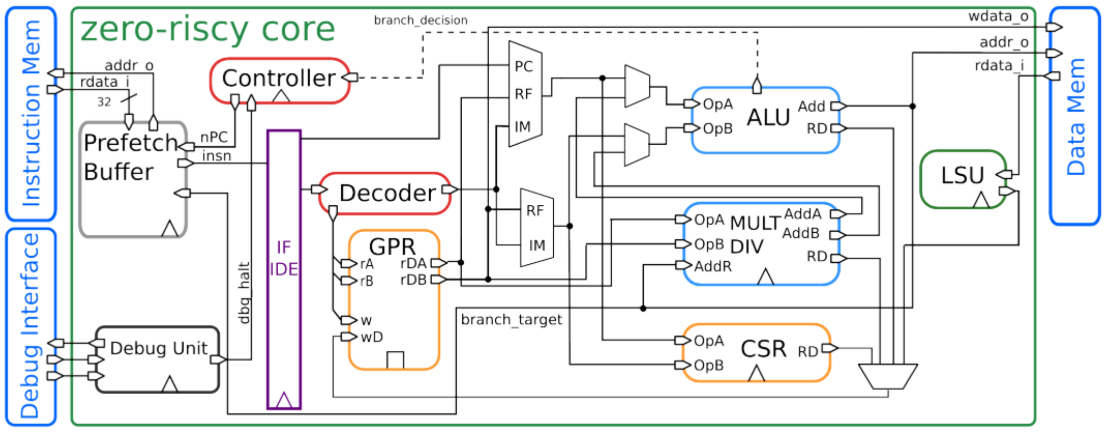


Figure 2.8: Zero-riscy Core Architecture

- RV32I Base Integer Instruction Set
- RV32E Base Integer Instruction Set (Light version of RV32I)
- RV32M Integer Multiplication and Division Instruction Set Extension
- RV32C Standard Extension for Compressed Instructions

2.2.1 PULPissimo

PULPissimo[5] is a 32-bit single-core microcontroller architecture, the core can be configurable either RI5CY or Zero-riscy, and uses a complex memory subsystem. The supported RISC-V instruction set depends on the selected core. RI5CY and Zero-riscy's supported instruction sets have declared in previous sections.
The simplified block diagram of the PULPissimo shown in Figure 2.9. The RI5CY

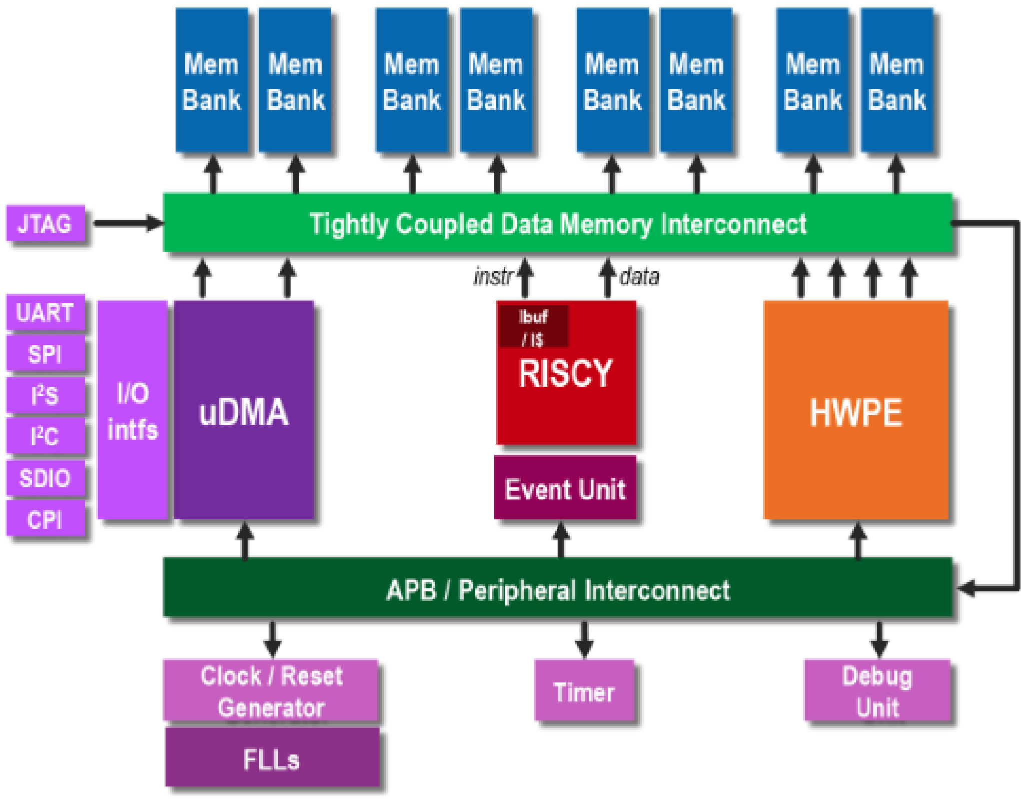


Figure 2.9: PULPissimo [6]

and Zero-riscy cores have the same external interfaces, so they are plug-compatible and can interchangeable with each other. All core registers have been memory-mapped and thus they are accessible through logarithmic-interconnect sub-system. PULPissimo has various peripherals, most of them are connected to the uDMA sub-system and thus all the data transfers can be held autonomously for the uDMA interconnected peripherals. However, FLLs, GPIO, timers, event unit, and event generator peripherals have connected to the APB bus instead of uDMA

subsystem. PULPissimo has a lightweight event and interrupt controller which supports FIFO events from the peripherals, or software events. Furthermore, the event and interrupt controller can manage events relying on their priority level.
The memory map of the PULPissimo is crucial for this work, this importance will be detailed in the further chapters. In a nutshell, as stated in the figure the RAM size is 512 kB, this would be another constraint for multi-threaded applications, because every thread has a different stack areas. Furthermore, application has to be designed considering this memory space.
The memory map of the PULPissimo is shown in Figure 2.10.

2.3 SiFive Cores

SiFive Cores has been designed by SiFive company[7]. SiFive company is one of the leading semiconductor companies that aimed to develop products such as cores, SoC, and boards with RISC-V architecture and its ISA. Despite SiFive has a wide range of RISC-V cores and products, this section digs into the E2 series of cores and their supported instruction sets. E2 series has been chosen to examine because they have able to work in a bare-metal environment and more or less they are equivalent to the RISC-V PULP platform. Although there would be vast sub-topics to introduce about the SiFive E2 Cores, their supported instruction sets will be examined.
E2 Series consists of E20, E21 and, E24 cores. They are mainly designed for energy-efficient applications, microcontrollers, and embedded systems. They can be customizable through specific requirements.

E20 Core

SiFive E20[8] core is a two-stage in-order 32-bit RISC-V core. The supported instruction sets can be ordered as follows,

- RV32I Base Integer Instruction Set
- RV32M Integer Multiplication and Division Instruction Set Extension
- RV32C Standard Extension for Compressed Instructions
- RV32F Single-precision Floating Point Extensions (Optional full support)

E21 Core

SiFive E21[9] core is a three-stage in-order 32-bit RISC-V core. The supported instruction sets can be ordered as follows,

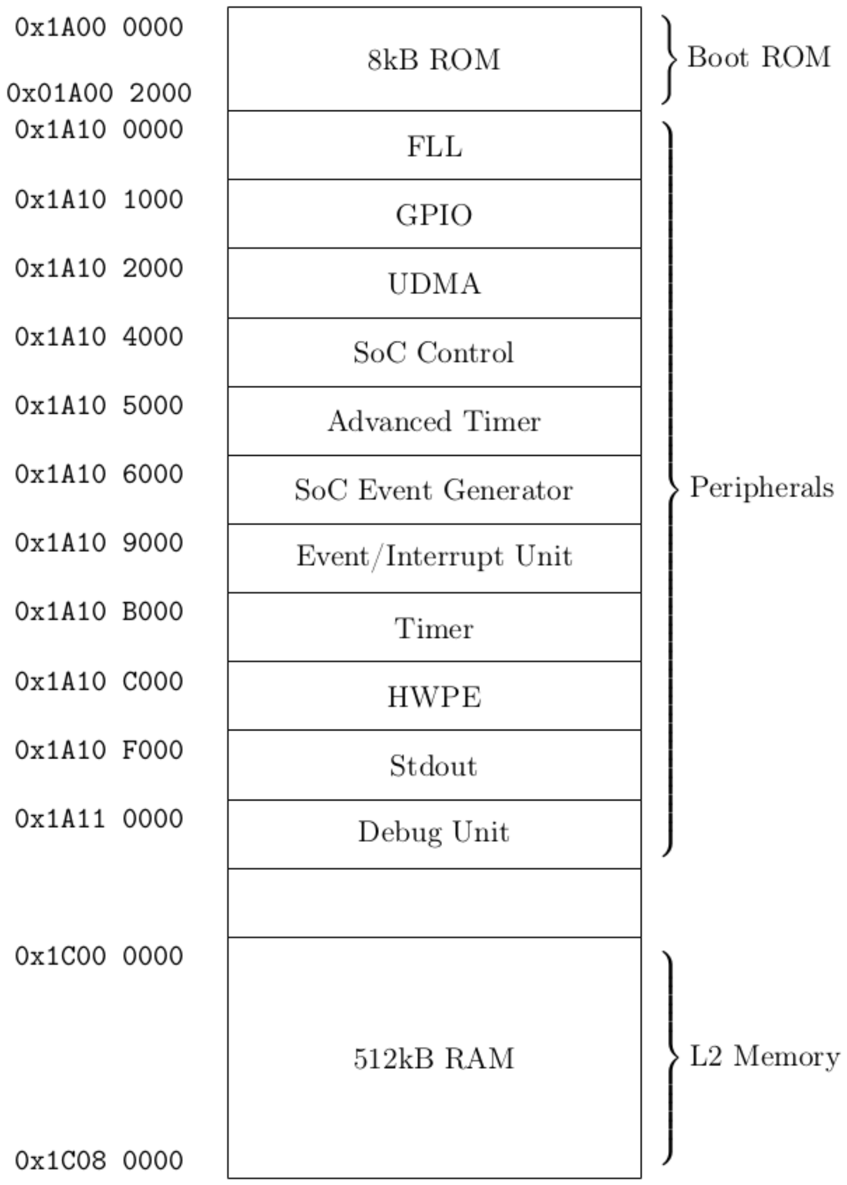


Figure 2.10: PULPissimo's Memory Map[5]

- RV32I Base Integer Instruction Set
- RV32M Integer Multiplication and Division Instruction Set Extension
- RV32A Atomic Instructions Set Extension

- RV32B Standard Extension for Bit Manipulation
- RV32C Standard Extension for Compressed Instructions
- RV32F Single-precision Floating Point Extensions (Optional full support)

E24 Core

SiFive E24[10] core is a three-stage in-order 32-bit RISC-V core. The supported instruction sets can be ordered as follows,

- RV32I Base Integer Instruction Set
- RV32M Integer Multiplication and Division Instruction Set Extension
- RV32A Atomic Instructions Set Extension
- RV32F Single-precision Floating Point Extensions
- RV32B Standard Extension for Bit Manipulation
- RV32C Standard Extension for Compressed Instructions

Chapter 3

Multi-threading and Standard C Libraries

This chapter is aimed to describe the multi-threading concept and its sub-topics related to this thesis work. The first section introduces the multi-threading concept, its types either software or hardware threads, then touches POSIX threads which are the first thing in the mind when multi-threading word is announced. Lastly, described Protothreads which is an architecture-independent event-driven multi-threading library.
In the second section, several standard C libraries will be introduced in the context of multi-threading. Those are the newlib which is the standard C library for embedded systems and the musl which is a barely new standard C library compared to others.

3.1 Multi-threading

Before introducing multi-threading, it is more convenient to define the word *thread* in computer science. A thread can be defined as a concurrent standalone function or a concurrent standalone sequence of instructions. Thus, multi-threading is the execution of more than one thread by a scheduler, simultaneously and in perfect harmony. Another important characteristic of the execution of threads is that they are architecture-dependent because threads are deeply managed by assembly instructions. The visualization of multi-threading is summarized in Figure 3.1.

The threads are shown in Figure 3.1 which are running the same process, simultaneously. If they are independent threads, means they do not make any operation (read, modify, or write) on the same variable or in the same memory address, they

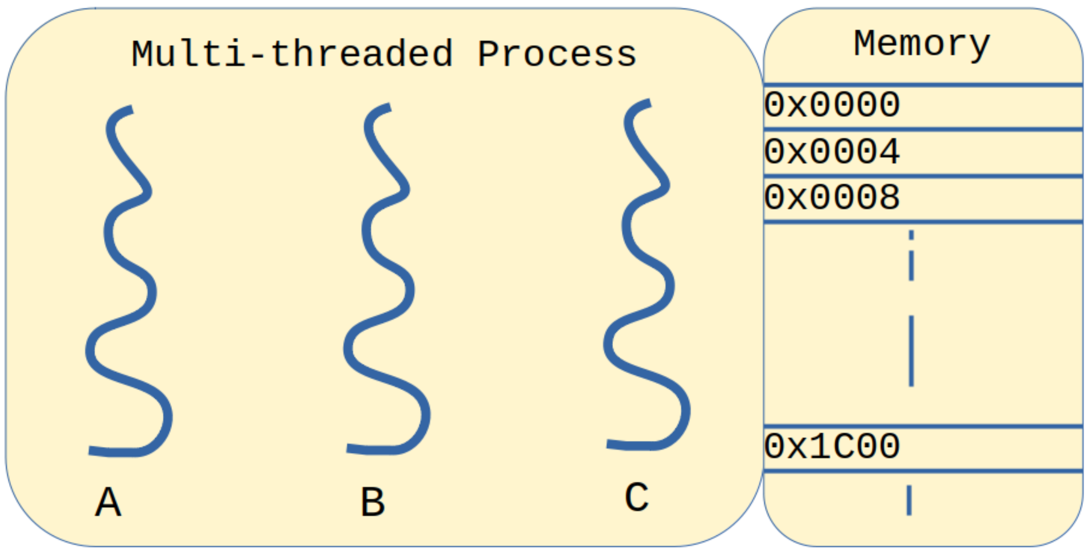


Figure 3.1: A Multi-threaded Process

can run at the same time. However, when more than one thread tries to access the same variable or the same memory address at the time, then there occurs unpredictable behaviour. Thus, synchronization is needed, in order to run those threads in perfect harmony or in other words run concurrently. The thread synchronization can be provided in several ways, using synchronization primitives such as mutexes and semaphores are the common way of providing a thread synchronization. A mutex can be defined as a flag in order to provide mutual exclusion and mutexes protect the critical section. Semaphores can be defined as a sophisticated version of mutexes. Different than mutexes, semaphores have a counter for allows or blocks the access of critical sections depending on the counter.

3.1.1 Schedulers

Schedulers have a big importance in an operating system's kernel so it is a vast subject in computer science but this section just describes its fundamental concepts and a scheduling algorithm. It manages the CPU for executing instructions, concurrently. A scheduler is a simple software that provides that operation, even in single-core systems, an illusion of parallelism can be achieved by dint of schedulers. In Linux systems, this scheduling process does not require additional code or software, because it has already implemented in the Linux kernel. However, in bare-metal environments, a scheduler has to be implemented manually, due to the lack of an operating system or a centralized kernel. As mentioned before, there are several approaches for scheduling algorithms, for the scope of this thesis, Round-Robin[11] is introduced below.

Round-Robin Scheduling Algorithm

The Round-Robin scheduling algorithm is based on a simple play and pause operation depending upon its scheduling quantum (as known as time-slices). The number of the quantum can be determined previously from different approaches. To explain Round-Robin's working mechanism, it's better to explain with an example. Assume that, two threads are scheduling to execute on a single-core CPU, concurrently. The first thread starts to execute for a given time slice, when the given time-slice has been ended, the first thread's context(all of the registers including stack pointer and program counter) saves. It should be noted that the first thread neither finished nor completed, it just paused in order to execute the second thread. Then the second thread starts to execute for a given time slice. Likewise, as in the first thread, when the given time-slice has been ended, the second thread's context saves. Then the CPU will carry on from the first thread, thus it restores the first thread's context and continues its execution. This operation lasts until the threads will finish their execution. For better understanding, the round-robin scheduling algorithm running mechanism is shown in Figure 3.2.

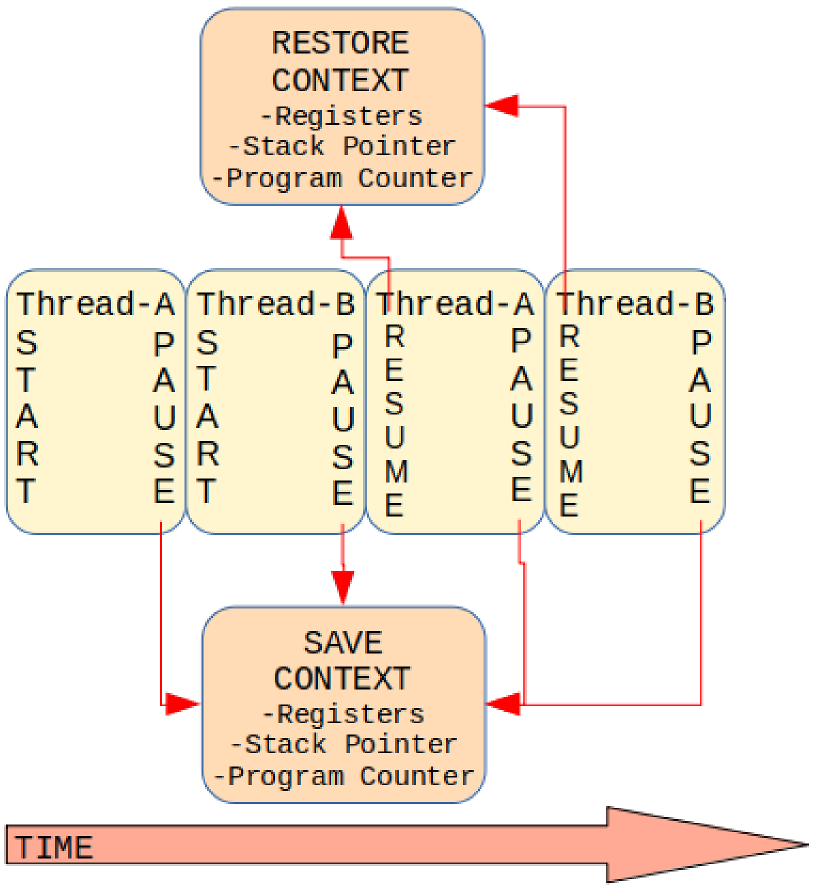


Figure 3.2: Round-Robin Scheduling Algorithm

3.1.2 POSIX Threads

As mentioned before thread execution depends on the CPU architecture. Historically, hardware vendors have implemented their thread libraries. Therefore, this situation was led to a complication for programmers in terms of the portability and compatibility of their programs. Thus, that situation has provided to the birth of thread library standards, POSIX threads[12] are implemented in the C programming language, and one of these standards, probably the most widely used one. POSIX threads generally referred to as "pthreads" is a thread library standard. Today, it is widely using in Linux systems, but the version supported by Linux has not supported by the Windows systems. However, apart from the Win32 threads, pthreads-w32 named pthreads library is available in the same syntax as the version of Linux pthreads for Windows systems.

Listing 3.1: Simple Thread Creation with POSIX Threads

```c
#include <pthread.h>
#include "led.h"

void *thread(void *ptr)
{
    blinkLED((int)ptr);
}

int main(int argc, char **argv)
{
    pthread_t thread1, thread2;
    int thread_1 = 1;
    int thread_2 = 2;

    pthread_create(&thread1, NULL, *thread, (void *)
  thread_1);
    pthread_create(&thread2, NULL, *thread, (void *)
  thread_2);

    pthread_join(thread1,NULL);
    pthread_join(thread2,NULL);

    return 0;
}
```

POSIX threads define and implement a set of C programming language types, functions, and constants. To illustrate POSIX threads, simple thread creation and execution example is shown in Listing 3.1. Firstly, two `pthread_t` structs are defined, a `pthread_t` which holds the elements for a thread. This struct is defined in the `pthread.h` library. Then, `pthread_create` function is called with given arguments, the first parameter takes the address of `pthread_t` struct's, the second parameter is for passing a thread attribute `pthread_attr_t`, for sake of simplicity it can be passed as NULL, the third parameter takes the thread function which will be executing as concurrent, and the last parameter takes the arguments of the thread function which passed to the third parameter of `pthread_create` function. As mentioned before, threads are executing as parallel or in an illusion of parallelism. Thus, the execution unit has to wait for each thread until they finish their execution. This wait operation can be done with `pthread_join`. Otherwise, the program may be terminated before threads are still running.

3.1.3 Protothreads

Dunkels *et al.(2006)*[13] proposes stackless threads named *Protothreads* regarding to an illusion of concurrency. It was designed for memory-constrained embedded systems based on an event-driven programming model. A *Protothread's* size only two bytes and does not require any stack per thread. Actually, the design of *Protothreads'* located between an event-driven and multithreaded programming model. Comparing POSIX threads' default stack size and the total memory size of tiny embedded systems will show us the inconsistencies in terms of memory usage. *Protothreads'* implemented on C programming language and its principle is based on preprocessor directives and switch-case statements. Therefore, it does not dependent on the processor's architecture or does not require any special compiler. Also, it does not require any specific scheduler to provide a concurrency or an illusion of concurrency. The scheduling of *Protothreads* is based on invoking its function and a *Protothreads* invokes by an event handler and it supports nested *Protothreads* invocation.
Protothreads also have semaphores in order to provide synchronization of each *protothread*, it mainly uses a *unsigned integer* in order to hold the semaphore's counter. As mentioned before, it mainly designed for considering tiny embedded systems. Generally, those embedded systems can not able to do real parallelism. Thus, any lock mechanism for the semaphore's counter is not necessary.
To clarify, it would be better to illustrate *Protothreads'* running mechanism with a source code both before the preprocessing and after.

Listing 3.2: Protothreads: Before Preprocessing

```c
#include "pt.h"
#include "led.h"

static struct pt pt1, pt2;
static int pt1_flag, pt2_flag;
static int protothread1(struct pt *pt)
{
  PT_BEGIN(pt);
  while(1) {
    PT_WAIT_UNTIL(pt, pt2_flag != 0);
    blinkLED(0);
    pt2_flag = 0;
    pt1_flag = 1;
  }
  PT_END(pt);
}
static int protothread2(struct pt *pt)
{
  PT_BEGIN(pt);
  while(1) {
    pt2_flag = 1;
    PT_WAIT_UNTIL(pt, pt1_flag != 0);
    blinkLED(1);
    pt1_flag = 0;
  }
  PT_END(pt);
}

int main(int argc, char **argv)
{
  PT_INIT(&pt1);
  PT_INIT(&pt2);
  while(1) {
    protothread1(&pt1);
    protothread2(&pt2);
  }
  return 0;
}
```

Listing 3.3: Protothreads: After Preprocessing

```c
static struct pt pt1, pt2;
static int pt1_flag, pt2_flag;
static int protothread1(struct pt *pt)
{
  { char PT_YIELD_FLAG = 1; switch((pt)->lc) { case 0:;
  while(1) {
    do { (pt)->lc = 13; case 13:; if(!(pt2_flag != 0)) {
   return 0; } } while(0);
    blinkLED(0);
    pt2_flag = 0;
    pt1_flag = 1;
  }
  }; PT_YIELD_FLAG = 0; (pt)->lc = 0;; return 3; };
}

static int protothread2(struct pt *pt)
{
  { char PT_YIELD_FLAG = 1; switch((pt)->lc) { case 0:;
  while(1) {
    pt2_flag = 1;
    do { (pt)->lc = 31; case 31:; if(!(pt1_flag != 0)) {
   return 0; } } while(0);
    blinkLED(1);
    pt1_flag = 0;
  }
  }; PT_YIELD_FLAG = 0; (pt)->lc = 0;; return 3; };
}

int main(int argc, char **argv)
{
  (&pt1)->lc = 0;;
  (&pt2)->lc = 0;;
  while(1)
  {
    protothread1(&pt1);
    protothread2(&pt2);
  }
  return 0;
}
```

In a nutshell, the main principle of this example is that either carries on existing protothread or return from the protothread to the while loop in main then enter to another protothread.

3.2 C Standard Library

The standard C library is the collection of built-in C functions, constants, and header files such as `stdio.h, stdlib.h`, and etc. The C standard library works as a reference manual for C programmers. There are many implementations of the C standard library. This section examines those implementations from the point of view of embedded systems and bare-metal environments.

Newlib

Newlib[14] is one of the C standard library implementations designed to use on embedded systems that neither have an operating system nor centralized kernel. In the context of multi-threading, newlib does not provide multi-threading by default, because newlib is not designed considering an operating system functionality and thus does not provide multi-threading by default.

Musl

Musl[15] is one of the C standard library implementations that built on top of the Linux system calls API and targets a wide range of systems from lightweight to fully-fledged. The musl is lightweight, simple, fast, and allows efficient static linking thanks to its design. In the context of multi-threading, musl supports POSIX threads and C11 threads. The further chapters will describe how to port musl to riscv32 architecture in order to exploit POSIX threads.

Chapter 4

Simulation Environment

This chapter introduces the concepts of our simulation environment. First, the chapter describes the code compiling and linking, then introduces the simulation platforms of ModelSim[16] which is mainly supported for PULP Platform's simulation tool. Then, introduces the RV-8 simulator which is used to simulate programs for SiFive RISC-V cores.

4.1 Code Compilation in C

Code compilation is the process of transforming the source code or codes into the object code, which means ready for the loader. The loader loads the executable object file into memory in order to execute by the CPU. The code compilation process can be held in two main stages by the compiler. The first stage called front-end compiling which is responsible for preprocessing, lexical analysis, syntax analysis and, semantic analysis; and the second stage called back-end compiling and which consists of a compiler, assembler, and linker. This section briefly introduces preprocessing stage in front-end compiling and examines back-end compiling. The overview of the code compilation process in C is shown in Figure 4.1.

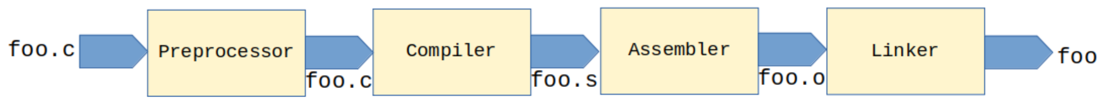


Figure 4.1: Code Compilation Process

Preprocessing

As known, the C programming language allows programmers to use preprocessor directives. These preprocessor directives take the # (hash) symbol at the beginning of the line. These processor directives simply make a substitution to their equivalents. So, preprocessing stage can be explained as changing the code parts with their equivalents. Also, this substitution can be conditional, which means only puts the chosen part for the back-end compilation, or as an opposite, it deletes the irrelevant parts. For better understanding, preprocessing examples are shown below,

Listing 4.1: Preprocessor Directives

```c
#define RISCV 0
#define ARM   1

#define ARCH RISCV

int main(int argc, char **argv)
{
    #if ARCH == RISCV
    // do RISC-V specific operations
    #endif

    #if ARCH == ARM
    // do ARM specific operations
    #endif

    return 0;
}
```

Preprocessor directives provide flexibility and easy-configurable source files, as shown in the example, a programmer can able change architecture just by setting the `ARCH` preprocessor variable. Then, the preprocessor only selects the code parts depending upon the condition and it cleans the irrelevant parts from the source code.

Compiling

Compiling is the process of translating C (or any high-level language) source code into a machine-specific assembly code.

Assembler

Assembler takes the machine-specific assembly code and translates it to relocatable machine code. It also checks the correctness of each instruction.

Linking

Linking is the process of creating a single executable file from an object file or a bundle of object files. There are two types of linking procedures that exist, static linking and dynamic linking, static linking has taken into account in the context of this thesis. Static linking is a process of merging all of the given object files and static libraries into a single execution file.
Linking process is managed by the linker file which has an extension of `.ld`. The linker file defines a set of rules for the linking including stack area, entry symbol, and memory sections.

Cross-compilation

Cross-compilation is a compilation technique that compiling sources for a system called target on a different system called a host. This is an often technique for generating executable files for embedded systems. The reason is that most of the embedded systems have not any operating system and have not sufficient capability to compile sources for themselves. Thus cross-compilation is quite advantageous than compiling sources in target systems. Cross-compilers is the tool built for doing cross-compilation, to configuring cross compilers there are many compilers build parameters are existing[17], but for the cross-compiling, the `build`, `host`, and `target` options are important, `build` is the system that currently using, `host` is the system where the programmer desires to run the compiler, `target` is the system that the executable files which are generated by the compiler will run to. There are several types of cross-compilers are listed in Table 4.1.

Cross-compiler	`build`, `host` are same; `target` is different
Crossback Compiler	`build`, `target` are same; `host` is different
Crossed Native Compiler	`host`, `target` are same; `build` is different
Canadian Cross Compiler	`build`, `host`, and `target` are different

Table 4.1: Cross-compiler Types

4.2 PULP Platform

This section describes how to compile and execute the desired program in the PULP platform, it is quite complex than simply build executables just by typing as follows,

```
gcc foo.c -o foo
```

This command preprocesses, compiles, assembles, and links, respectively for default compiler configuration. After that process, a single executable file has been created. A set of source files including start-up files has to be compiled and linked in order to build a single executable file for the PULP platform. Start-up files ignite the execution and forward the execution flow to the main function, the start-up files generally have a symbol of `_start` which is the entry point of the C programs. By the way, this simulation environment is designed considering mainly C programs and C++ programs.
In fact, the PULP platform has its own runtime environment, but it is not extensible and feasible for us in terms of the targets of this thesis work. Because of that, an extensible and easily configurable simulation environment has been built.
First of all, a toolchain has to be built for the PULP platform, PULP platform has its own toolchain which named PULP RISC-V toolchain[18], this toolchain will be used in order to compile and link our sources.

Toolchain Build

To get a successful toolchain built, the PULP toolchain's installation instructions can be followed. However, the toolchain's arch parameter can be configured as written below,

```
--with-arch=rv32imafc
```

Despite the RI5CY and Zero-riscy do not support atomic instructions set extension, the toolchain has been configured with this `arch` option in order to flexibly compile sources and trace the flow whether the sources had an inline assembly call.
This PULP toolchain contains Binutils, GCC compiler, and newlib C standard library which all ported to riscv32 architecture. However, the binaries will be taken into account after the build. They have a prefix of `riscv32-unknown-elf-`, for example, our C and C++ compilers are `riscv32-unknown-elf-gcc` and `riscv32-unknown-elf-g++`, respectively.

Simulation Environment Skeleton

Referenced from PULP-runtime[19], a custom runtime skeleton has been created as shown in the directory tree,

```
pulp-runtime
├── pulp-sdk
├── pulp
├── pulpissimo
├── pulp-toolchain
└── baremetal-riscv-threads
    ├── drivers
    ├── include
    ├── kernel
    ├── lib
    └── src
```

`pulp-runtime` includes the shell scripts in order to set necessary environment variables through selected configuration. `pulp-sdk` contains SDK sources and shell scripts for simulating applications using PULP-SDK. `pulp` and `pulpissimo` directories contain configuration files and shell scripts, respectively for each chip. `pulp-toolchain` contains the built files of PULP toolchain, `drivers` contains the driver source files like UART, SPI, GPIO interfaces, `include` contains the header files, `kernel` contains the configurable source files of PULP platform, `lib` contains the source files of minimal C library, and `src` contains our application source files which include the `main` function. Referenced the PULP-runtime, the necessary source files has been placed as follows,

```
baremetal-riscv-threads
├── drivers
│   └── uart.c
├── include
│   ├── archi
│   ├── bench
│   ├── chips
│   ├── crt
│   ├── data
│   ├── hal
│   ├── implem
│   ├── ctype.h
│   ├── io.h
│   ├── pulp.h
│   ├── stdio.h
│   ├── stdlib.h
│   └── string.h
├── kernel
│   ├── chips
│   ├── crt0.S
│   ├── irq-asm.S
│   ├── alloc-pool.c
│   ├── alloc.c
│   ├── bench.c
│   ├── cluster.c
│   ├── fll-v1.c
│   ├── freq-domains.c
│   ├── init.c
│   ├── irq.c
│   ├── kernel.c
│   └── soc-event.c
├── lib
│   ├── fprintf.c
│   ├── io.c
│   ├── prf.c
│   └── sprintf.c
└── src
    └── sample-project
        ├── main.c
        └── Makefile
```

Compiling and Linking

The compile and linking phases are driven by a Makefile. Makefile defines a set of rules for compile and linking phases. The base Makefile's rules are provided in Appendix A.1 for building a single executable file. This base Makefile's rules can be explained with the following bullet points in a nutshell,

1. Chip selection, the available options either `pulp` or `pulpissimo`
2. Setting toolchain, GCC, and build directories
3. Definition of executable file name
4. Definition of compiler and linker flags
5. Definitions for preprocessing stage
6. Setting kernel and library source files
7. Compiling of assembly sources, object files will be created at the end of the operation
8. Compiling of C sources including the application source file or source files, object files will be created at the end of the operation
9. Linking all of the source files and static libraries (if chosen) into a single executable file

In order to build single a executable file for C++ applications, the base Makefile has to be modified with minor changes, the base Makefile rules for C++ applications are provided in Appendix A.2. The only modification is that after compiling C sources, C++ sources must be compiled, and object files must be created for them. By dint of these base Makefiles, any modification can be made easily, besides any supported API can be added.
Afterwards, the single executable file can be created with `make` command as follows,

```
>>: ~/<PATH>/baremetal-riscv-threads/src/sample-project$ make
```

4.2.1 PULPissimo

For compiling sources in order to simulate applications for PULPissimo[6], the base Makefile can be used without any major modification. The only modification is that setting application source files and configuring `CHIP` variable that shown below,

```
CHIP = pulpissimo
```

```
PULP_APP_SRCS = main.c sample-api.c
```

For the simulation, the PULP platform suggests simulating applications on ModelSim/QuestaSim's VSIM RTL Simulator for `PULPissimo`. VSIM is a fully-featured VHDL and/or Verilog simulator. Before executing VSIM, some environment variables have to be defined and some necessary files have symbolically linked to the build directory as written in the base Makefile.
After building of executable file, these commands as follows has to be entered in the command line in order to set environment variables and other necessary configuration.

```
>>: ~/<PATH>/pulp-runtime$ source configs/pulpissimo.sh

>>: ~/<PATH>/pulpissimo$ source setup/vsim.sh
```

Then, the RTL simulation platform is ready to execute the application, the execution can be started with the command as follows,

```
>>: ~/src/sample-project$ make modelsim
```

After `make modelsim` command, the Makefile symbolically links `modelsim.ini`, `boot`, `tcl_files`, and `waves` files to the given build directory then sets `--binary` argument to the single executable file. Then executes `VSIM` with stated configurations for the simulation.

4.3 RV8

Before introducing the RV8 simulator, it would be better to explain how to achieve simulations in the RV8 simulator. As mentioned before, a set of rules is needed for the simulation, and these rules are provided by Makefiles. As predicted, processors do not have any systems or memory like PULPissimo, they just execute the instructions, respectively. Thus the Makefile rules for SiFive processors will be much simpler than the PULP platform's provided Makefile rules. The Makefile for compiling sources with musl for SiFive processors has provided in Appendix A.4. As mentioned before, this work also examines SiFive processors for the context of bare-metal POSIX thread execution. In order to simulate executables for SiFive processors, RV8 has been preferred. RV8[20] is a simulation suite for RISC-V architecture-based applications. It contains a set of simulators for various purposes, those are listed in the table below,

For sake of simplicity, `rv-jit` and `rv-sim` has been preferred to simulate applications. Furthermore, these two simulators are sufficient for trace the execution

`rv-jit`	user mode x86-64 binary translator
`rv-sim`	user mode system call proxy simulator
`rv-sys`	full system emulator with soft MMU
`rv-bin`	ELF disassembler and histogram tool
`rv-meta`	code and documentation generator

Table 4.2: RV8 Simulation Suite Simulators

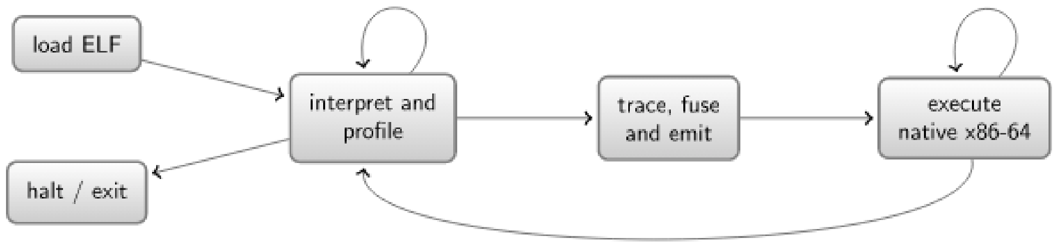


Figure 4.2: RV8 Working Principle

flow, thanks to their user-friendly command sets.
The working principle of RV8 simulator is summarized in Figure 4.2.

As shown in Figure 4.2, RV8 translates the instructions to x86-64 architecture which is the most used personal computer architecture and executes the instructions in the native x86-64 environment. Then, it reports the output of the code to the simulator.
To execute an executable file in the RV8 simulator is quite easy, especially comparing to the ModelSim/QuestaSim's VSIM RTL simulator. The execution commands are written below,

```
>>:~/src/sample-project$ rv-jit <exec_file>

>>:~/src/sample-project$ rv-sim <exec_file>
```

Chapter 5

Case Study

This chapter aims to introduce a case study that porting musl to riscv32 architecture in order to exploit POSIX threads for a bare-metal RISC-V environment and execution of these threads in PULP platform and in SiFive environment. The latest stable version of musl is not supporting riscv32 architecture and thus musl's riscv32 port is required first. Then, the cross-compilation of the "musl" sources has introduced in order to build libraries and headers. After that, the simple musl POSIX threads program have implemented which is shown in Listing 5.1, both on the PULP platform and SiFive processors instruction set simulator of RV8 in order to trace thread creation and execution. The creation and execution result is provided in the given sections, but the overall conclusion will be introduced in the last chapter.

Listing 5.1: Simple musl POSIX Threads Program

```c
#include <math.h>
#include <pthread.h>
#include "led.h"

void *foo()
{
    blinkLED(0);
    pthread_exit(NULL);
}

int main()
{
    pthread_t thread_id;
    int ret;
    double result = pow(10.0,2.0);
    double result2 = sqrt(100.0);

    if(result == 100) blinkLED(1);
    if(result2 == 10) blinkLED(2);

    ret = pthread_create(&thread_id, NULL, foo, NULL);
    if (ret) {
      blinkLED(3);
      exit(-1);
    }
    pthread_join(thread_id, NULL);

    return 0;
}
```

The program also contains `math.h` functions, the reason is that checks whether musl's riscv32 port has been succeeded. The program also controls the `pthread_create` return value, this would be a good indicator for tracing the execution flow using musl POSIX threads' source codes.

5.1 Multi-threading with musl

As stated in previous chapters, musl supports POSIX threads, before describing musl POSIX threads support, it would be better to describe musl port to riscv32 which is the desired architecture configuration. After that, this section describes musl's POSIX threads execution on the PULP platform and SiFive processors. Then, introduces the encountered problems and stated results.

Musl riscv32 Port

Michael J. Clark et al. (2018)[21] have been ported musl's 1.1.18 version to riscv32 architecture. Despite of musl's latest stable version is 1.2.2, there are no major changes between version 1.1.18 and 1.2.2 for POSIX threads. There are only changes for performance improvements and advanced POSIX features implementations. Therefore, the work of Michael J. Clark can be used for this thesis's work.

Musl Cross Build

As mentioned in the previous chapters, the application will be executed in the bare-metal RISC-V platforms either in the PULP platform or SiFive Processors, and thus cross-compilation is needed. So, the musl library will be compiled in the x86-64 machine and executed in the riscv32 machine, and thus the build configuration should be as given in the table.

`build`	x86-64
`host`	x86-64
`target`	riscv32-unknown-elf

Table 5.1: Musl Cross Compilation Architecture Options to 32-bit RISC-V

In order to start cross-compilation, the `riscv32-unknown-elf-*` binaries' path should be appended to the `PATH` environment variable. Otherwise, the musl's Makefile cannot find the riscv32's compiler to compile the sources. This operation can be done with the following command as written in below

```
>>: ~/export PATH=$PATH:/<TOOLCHAIN_PATH>/bin
```

After that, with `which riscv32-unknown-elf-gcc` command can be check whether the operation is succeeded.

```
>>: ~/which riscv32-unknown-elf-gcc
<TOOLCHAIN_PATH>/bin
```

Before the build, `config.mak` or musl has to be configured for the desired configuration. This can be done with the following command for our purpose.

Listing 5.2: Musl Build Command for 32-bit RISC-V

```
CFLAGS="-DSYSCALL_NO_INLINE" ./configure --prefix=<
    INSTALLATION_PATH> --target=riscv32-unknown-elf --
    enable-multilib --disable-shared
```

Then, the build process can be started with the following command, this command will compile the sources, then creates static libraries, dynamic libraries are not chosen to be built, because the target is a bare-metal embedded environment thus dynamic libraries are not necessary.

```
>>: ~/<MUSL_PATH>/ make
```

It should be noted that, after the `make` command, the Makefile rules do not initiate the installation of the musl headers and libraries to the installation path that the programmer declared in the `--prefix` parameter. The headers and libraries are created in the local musl directory. To install musl headers and libraries in the given path, the following command has to be executed.

```
>>: ~/<MUSL_PATH>/ make install
```

Finally, musl has been cross-compiled and the libraries and the headers are created, the final musl built is shown below,

```
 include
 │ └── // header files
 └── lib
      ├── crt1.o
      ├── crti.o
      ├── crtn.o
      ├── libc.a
      ├── libcrypt.a
      ├── libdl.a
      ├── libm.a
      ├── libpthread.a
      ├── libresolve.a
      ├── librt.a
      ├── libutil.a
      ├── libxnet.a
      ├── rcrt1.o
      └── Scrt1.o
```

Because of the `--disable-shared` option, all the objects have been placed in the `libc.a` static library. The other static libraries are entirely empty. Therefore, the only static library that will be used is `libc.a`.

`crt1.o` can be called as ignition file, this object file contains `_start` symbol which is the default entry point for C applications, then it calls the function of `_libc_init_main` which calls other functions in order to initialize musl `libc`'s parameters.

`crti.o` and `crtn.o` object files are define function prologues and epilogues for the `.init` and `.fini` sections, respectively. `Scrt1.o` is used when generating PIEs. `rcrt1.o` is used when generating static-PIEs. Eventually, the musl is ready for linking and simulation.

5.1.1 PULP Platform

This section is aimed to integrate musl into the PULP platform, for the sake of simplicity, the PULPissimo has been selected as a microcontroller. As mentioned in previous chapters, PULPissimo is a single-core microcontroller and the expected output is that of running a multi-threaded application with an illusion of parallelism. To integrate musl into the PULP platform, several modifications are required. The most important one is that start-up files. Musl and PULPissimo's runtime has their own start-up files, but both start-up files' logic is almost the same, they both enter from the `_start` symbol, doing necessary pieces of stuff then forward the flow to the main function. However, these two start-up files cannot be used at the same time, because the same symbols have already existed in these files. Thus, musl's `crt1.o` cannot able to be used, a new file called `crt1.c` has to be created in the PULPissimo's API. The source code of the `crt1.c` is provided in Appendix B.1. Then, the PULPissimo's start-up file (`crt0.S`) has to be modified in the `.text` section. Original `crt0.S` forwards the flow to the main function, directly, but before the main, `__libc_start_main` function should be executed for the musl's libc initialization, and thus the modification has been made in the PULPissimo's `crt0.S` file.

Listing 5.3: PULPissimo's crt0.S modification

```
.section .text

    # On all other chips we simply pass 0.
    addi  a0, x0, 0
    addi  a1, x0, 0

    # Jump to main program entry point (argc = a0, argv
  = a1).
    # la    t2, main
    la    t2, _start_musl_c
    jalr  x1, t2
    mv    s0, a0
```

Instead of jumping to main, this modified `crt0.S` jumps to the `_start_musl_c` function in order to do initialization for musl libc. Then, it forwards the flow to the `main` function.

To integrate musl with PULPissimo API, the base Makefile has to be modified with minor changes. These modifications can be summarized as,

- musl's path declaration
- musl's include path declaration

- musl `libc.a`'s integration in the linking phase.

The modified Makefile is provided in Appendix A.5.

Simulation Result

As mentioned before, the simulation can be started with a simple command of,

```
>>: ~/<PROJECT_PATH>/ make modelsim
```

The simulation output is given below,

```
# [STDOUT-CL31_PE0] LED-1 is blinking..
# [STDOUT-CL31_PE0] LED-2 is blinking..
# 13304880ns: Illegal instruction (core 0) at PC 0x1c009fb8:
```

The `illegal instruction` error generally occurs for two possible reasons, the most common one for C applications is that because of an unsupported instruction. Or program counter jumps to the wrong address. To trace the error, the single executable file has dissembled and the address `0x1c009fb8` and its content is shown below,

```
1c009fb2:       e08a8a93      addi  s5,s5,-504
1c009fb6:       c85c          sw    a5,20(s0)
1c009fb8:       100aa7af      lr.w  a5,(s5)
1c009fbc:       0785          addi  a5,a5,1
1c009fbe:       18faa9af      sc.w  s3,a5,(s5)
1c009fc2:       fe099be3      bnez  s3,1c009fb8
1c009fc6:       57fd          li    a5,-1
```

As written in the error, there is an illegal instruction in the address of `0x1c009fb8`, the address contains `lr.w` instruction which is the member of RV32A - Atomic Instruction Set which the PULP platform does not support these instruction set extension. Furthermore, the RISC-V instruction set manual[1] states that on page 13, *"RV32I can emulate almost any other ISA extension (except the A extension, which requires additional hardware support for atomicity).* The reason behind this statement is that RV32A - Atomic Instruction Set provides address-based reservation and instruction ordering, even the current version of `FENCE` instruction cannot able do address-based reservation, it just provides instruction-based ordering. Thus, it is impossible to execute atomic instructions on the PULP platform without any hardware support for atomicity.

5.1.2 SiFive Processors

As mentioned in previous sections, SiFive processors have not any API like the PULP platform, they just contain processors that purpose executing the instructions one by one. Also, the Makefiles rules with musl integration have been provided which can be found in Appendix A.4.
Compilation can be started with the `make` command in the related project directory. After a successful built, simulation can be started with logging system calls and memory map information,

```
>>: ~/<PROJECT_DIR>/ make rv
```

Simulation Result

The simulation output is given below,

```
sys_set_tid_address(0xc7b28)        = 1
ioctl(1,21523)                      = 0
LED-1 is blinking..
writev(1,0x7effffffab0,2)           = 31
LED-2 is blinking..
writev(1,0x7effffffad0,2)           = 28
mmap(0x0,143360,0,34,-1,0)          = 0x40000000
mprotect(0x40002000,135168,3)       = 0
clone(8195840,1073884912)           = 22
munmap(0x40000000,143360)           = 0
MAIN: pthread_create return 11
writev(1,0x7effffffad0,2)           = 31
ERROR; return code from pthread_create() is 11
writev(1,0x7effffffad0,2)           = 47
exit(-1)
make: *** [Makefile:71: rv] Error 255
```

As inferred from the output, `pthread_create` is calling `clone` system call however, the code is running on a bare-metal environment. As mentioned in previous chapters, bare-metal environments do not have any operating system support or centralized kernel. Thus, there is no system call. Also, for tracing further errors, musl has re-compiled with the assumption of the `clone` system call has returned success. Then, `pthread_join` has called `futex` system call for synchronization and `pthread_create` has called `set_scheduler` system call when a POSIX threads attribute is preferred for thread creation. For a counter approach, writing support for these system calls such as `clone`, `futex`, and etc. is not guaranteed to execute musl's POSIX threads. Because as mentioned in previous chapters, musl is built on top of the Linux system call API and, `clone` system call clones a child process with a new stack, but the program is running on a bare-metal environment so there is no Linux process. Furthermore, assume that all these system calls have been supported. However, In the end, a kernel will be formed, unwillingly. Therefore, the whole system would stay out of the bare-metal environment term.

5.2 Multi-threading with Protothreads

As mentioned in previous sections, Protothreads' working principle is based on preprocessor directives and does not use any source code for invoking or running threads. So, in order to exploit Protothreads, the related header files have to be included to the include search path of the compiler, this can be done using the base Makefile as follows,

```
PT_DIR = $(CURDIR)/../../../pt-1.4
INC_DIRS += $(addprefix -I, $(PT_DIR)/include/)
```

It should be noted that Protothreads' syntax is different than POSIX threads. So the simulation source code has to be re-implemented considering the POSIX thread program which is given in Listing 5.1. The re-implemented source code is given in Listing 5.4. Also, in this case study, a simple producer-consumer problem source code with Protothreads and its simulation has been provided. The producer-consumer problem source code is referenced from in the Protothreads' official project[13] and provided in the Appendix B.3 with the copyright notice.

Listing 5.4: A Simple Protothread Program

```c
#include <pt.h>
#include "led.h"

static struct pt pt1;

static int foo(struct pt *pt)
{
    PT_BEGIN(pt);
    blinkLED(0);
    PT_END(pt);
}

int main()
{
    PT_INIT(&pt1);
    foo(&pt1);
    return 0;
}
```

Simulation Result

The simulation outputs for Listing 5.4 and Appendix B.3 on the PULPissimo is given below, respectively.

```
Simulation Output for Listing 5.4:
-------------------------------------------------
# [STDOUT-CL31_PE0] LED-0 is blinking..
-------------------------------------------------

Simulation Output for Appendix B.3:
-------------------------------------------------
# [STDOUT-CL31_PE0] LED-0 is on by producer
# [STDOUT-CL31_PE0] LED-0 added to buffer at place 0
# [STDOUT-CL31_PE0] LED-1 is on by producer
# [STDOUT-CL31_PE0] LED-1 added to buffer at place 1
# [STDOUT-CL31_PE0] LED-0 retrieved from buffer at place 0
# [STDOUT-CL31_PE0] LED-0 is off by consumer
# [STDOUT-CL31_PE0] LED-1 retrieved from buffer at place 1
# [STDOUT-CL31_PE0] LED-1 is off by consumer
# [STDOUT-CL31_PE0] LED-2 is on by producer
# [STDOUT-CL31_PE0] LED-2 added to buffer at place 0
# [STDOUT-CL31_PE0] LED-3 is on by producer
# [STDOUT-CL31_PE0] LED-3 added to buffer at place 1
# [STDOUT-CL31_PE0] LED-2 retrieved from buffer at place 0
# [STDOUT-CL31_PE0] LED-2 is off by consumer
# [STDOUT-CL31_PE0] LED-3 retrieved from buffer at place 1
# [STDOUT-CL31_PE0] LED-3 is off by consumer
# [STDOUT-CL31_PE0] LED-4 is on by producer
# [STDOUT-CL31_PE0] LED-4 added to buffer at place 0
# [STDOUT-CL31_PE0] LED-5 is on by producer
# [STDOUT-CL31_PE0] LED-5 added to buffer at place 1
# [STDOUT-CL31_PE0] LED-4 retrieved from buffer at place 0
# [STDOUT-CL31_PE0] LED-4 is off by consumer
# [STDOUT-CL31_PE0] LED-5 retrieved from buffer at place 1
# [STDOUT-CL31_PE0] LED-5 is off by consumer
-------------------------------------------------
```

Thus, one can say that Protothreads can provide concurrency by dint of event-driven programming techniques, but should be noted that, Protothreads cannot provide real parallelism, although providing an illusion of parallelism.

Chapter 6

Proposed Approach

This chapter is aimed to introduce a multi-threading approach for bare-metal platforms. As stated in previous chapters, musl's POSIX threads are not convenient for bare-metal systems that have been focused on, due to musl's inline assembly calls, start-up files and the implementation based on the Linux system calls API. Thus, this approach is based on implementing a standalone multi-threading API inspired from POSIX threads and Protothreads, and porting it to the PULPissimo. The API is called "bare-threads" or "bthreads". Hereinafter, this API is called "bare-threads".

6.1 Bare-Threads API

Bare-threads API is an experimental standalone multi-threading library developed for the context of this thesis. The main motivation of implementing bare-threads API is based on portability and compatibility. The implementation has designed considering musl POSIX threads' issues for the bare-metal compatibility, the issues can be summarized as bullet points as follows.

- Bare-threads API does not need any pre-built objects or files, although musl's POSIX threads need musl's pre-built start-up files in order to initialize POSIX threads.

- Musl's POSIX threads implementation contains inline assembly calls and uses RV32A - atomic instruction set extension in spite of that bare-metal API does not implement any atomic instruction in its base functions.

- The musl is built on top of the Linux system calls API nevertheless Bare-threads API does not call any Linux system call by default.

The bare-threads API has implemented in the C programming language and consists of a set of C source code files and header files. The implementation is based on simplicity and compactness. The implementation shows a lot of similarities with POSIX threads implementation. Listing 6.1 shown below declares a thread creation both in POSIX threads and bare-threads. Also, the other functions have a similarity with POSIX threads.

Listing 6.1: POSIX thread creation versus bare-thread creation

```c
// POSIX threads
int pthread_create(pthread_t *__restrict, const
   pthread_attr_t *__restrict, void *(*)(void *), void *
   __restrict);

// bare-threads
int bthread_create(struct bthread_t *bId, struct
   bthread_attr_t *bAttr, void *(*entry)(void *), void *
   args);
```

The directory structure of the bare-threads is given below, it consists of three main directories, `include` path contains header files, `src` path contains source codes, and `arch` path contains architecture-specific sources. The `arch` path has to contain mandatory files and macros for each architecture.

```
bare-threads
 ├─ arch
 ├─ include
 └─ src
```

Thread Creation

As mentioned before thread creation is dependent on the platform and the architecture of the platform. Indeed, it depends on many parameters such as the number of the cores, supported instruction set, and etc. Thus, thread creation in bare-threads API is offered by architecture-dependent functions, `bthread_create` function initializes necessary elements for thread creation and then sets the elements of the `bthread_t` struct, and calls for the execution of the thread. The `run` function is the thread execution call provided from the architecture-dependent sources, and the syntax of this function has to be the same for all architecture ports. The source code of the `bthread_create` is provided in Listing 6.3. Also, bare-threads offer a thread pool for further improvements. It should be noted that the `run` function has to be implemented as architecture-dependent or platform-dependent and implemented in the `arch` directory. The `run` function takes a `bthread_t` struct as an argument and the syntax is provided in Listing 6.4. The implementation of the `run` function will be described in further sections.

Listing 6.2: bthread_t Structure

```
#define THREAD_STACK_SIZE 8192

typedef struct {
    unsigned int pc;
    unsigned int regs[31];
} context_t;

typedef struct bthread_t{
    unsigned short tId;
    void *(*f)(void *);
    void *args;
    context_t context;
    unsigned int stack[THREAD_STACK_SIZE]
    volatile void *status;
    struct bthread_attr_t *attr;
}bthread_t;
```

`bthread_t` struct contains necessary elements for a thread. `bthread_t` structure shown in Listing 6.2. `tId` is the unique id number for a thread, `f` is the function pointer which holds the entry function of a thread, `args` is the argument of the thread function, `context` is the integer registers' values and program counter value for the thread, `stack` is the stack space for the thread, `status` is the state of the thread to check whether a thread has been finished or still running, and the `attr` holds the thread attributes of the thread.

Listing 6.3: bthread_create.c Source Code

```
int bthread_create(struct bthread_t *bId, struct
   bthread_attr_t *bAttr, void *(*entry)(void *), void *
   args)
{
    if(!isInit)
        __bthread_init();
    bId->tId = unique_id++;
    bId->f = entry;
    bId->args = args;
    if(bAttr != NULL)
        bId->attr = bAttr;
```

```c
    __enqueue(thread_queue, *bId);

    run(bId);

    return 0;
}
```

Listing 6.4: run Function Syntax

```c
short run(struct bthread_t *thread);
```

Synchronization

Thread synchronization in bare-threads has provided by mutexes and semaphores. Mutexes and semaphores are defined in previous chapters. As stated in the beginning, bare-threads API is an experimental library. Therefore, the mutex and semaphore implementations are simplified. The mutex implementation is simply based on a flag variable, depending on the flag variable's value, the mutex either allows a thread to enter the critical section or blocks threads until the critical section becomes available. The semaphore implementation is based on a mutex and a counter. Simply, when a semaphore has been signaling, the counter value will have been increased and when it has been waiting, the counter value will have been decreased.
The mutexes and semaphores are called `bthread_mutex_t` and `bthread_sem_t`, respectively. The source codes and implementations are provided in Appendix B.4.

Porting

The bare-threads porting can be provided with the files in the `arch` directory. The architecture-specific functions, constants, or any elements have to be reachable through `core.h`. Therefore, `core.h` is a mandatory header file that has to exist in all architecture ports and it has to contain a `run` function which is able to execute a thread. The porting sources should be in the form which is written below,

```
arch
└─riscv32
   ├─pulpissimo
   │  ├─core.h
   │  └─core.c
   └─pulp
      ├─core.h
      └─core.c
```

Scheduling

The source code of the bare-threads API does not provide any scheduling mechanism, because scheduling depends on many parameters such as application type, core architecture, supported instruction set, and etc. Therefore, any scheduler has not provided by default. However, a scheduler has to be provided in architecture-dependent sources. Furthermore, bare-threads are designed for bare-metal systems which do not have any operating system support nor centralized kernel. Therefore, default scheduler support would be not a realistic approach. A scheduling instance will be examined in further sections.

Attributes

As mentioned in the previous sections, bare-threads is an experimental API, Therefore, the thread attributes implemented for future improvements. Thus, bare-threads attributes have not tested. They implemented in order to show, bare-threads can take attributes and has an extensible interface.

6.1.1 Single-Core Multi-threading

As mentioned in previous chapters, single-core multi-threading can be provided only and only if with an illusion of parallelism. Hyper-threaded cores are excluded and they are not in the scope of this thesis. This concurrency can be provided with a scheduler, it schedules threads and executes on the core. There are various types of schedulers approach has been available, but in this scope, Round-Robin scheduling has been preferred. It has a simple algorithm based on play and pause operation. To simulate and port bare-threads on a single-core microcontroller, PULPissimo has been chosen which is one of the main actors of this thesis work.
Before start porting bare-threads API to the PULPissimo, it would be better to figure out the maximum asynchronous thread capability of the PULPissimo. Herdt et al. (2018)[22] propose a new approach for the RISC-V simulator which named RISC-V VP or RISC-V Virtual Prototype, and RISC-V VP's source directory has an example code for simple scheduling single-core threads. Referenced from it, a test code has written. The test code is based on a simple concurrent program that can be useful to figure out PULPissimo's capabilities. The test code's scheduling mechanism has preferred as the simplest. The scheduling is provided with a context switch that switches the execution context of each thread, including the main thread. This context switch, interchanges the main registers with each thread stack's context for the execution of threads, concurrently. This context can be extensible, but the skeleton context has integer registers, stack pointer, and program counter. Furthermore, with that code also a minimal concurrency will have been proven on the PULPissimo platform. The test code is provided on RISC-V VP's source

directory[23], to test this code on PULPissimo the base Makefile for PULPissimo can be used with adding source files to the `PULP_APP_SRCS` variable.

Simulation Result

The default stack area has defined as ~64 kB, with this stack area configuration, eight threads are capable to run concurrently. Eight threads with ~64 kB stack area are sufficient for many applications will execute on the PULPissimo. So, one can infer that PULPissimo is suitable for multi-threaded applications. It should be noted that this simulation is made without bare-threads' PULPissimo port. This simulation has made with an experimental test code with a scheduler in order to figure out PULPissimo's capabilities.

Porting to PULPissimo

After proving PULPissimo's suitability for the multi-threaded applications, bare-threads API's PULPissimo port can be explained. It should be noted that bare-threads PULPissimo's port is not implemented, this section is given as a guideline for proposing how multi-threaded applications can be run on the PULPissimo. The proposed approach for single-core systems in order to provide multi-threading is summarized in Figure 6.1. In the figure, the CPU shown with a purple arrow, the arrow on the left defines the threads that have not started yet, the arrow on the right defines the threads that have already finished, the curved arrow represents a timer interrupt in order to trigger the context switch, in other words, it turns the wheel when the timer interrupt occurred, the wheel which contains threads stands for the thread pool, and each wheel slices represents a thread. Hereinafter this thread pool is called as thread wheel.

According to Figure 6.1, after the program has been executing, when a thread created with `bthread_create`, bare-threads appends this thread on the wheel as a wheel slice or this can be called thread slice. The number of the slices represents the total running threads and this can be defined or limited for each platform or architecture. As mentioned before, the curved arrow represents a timer interrupt, when the timer interrupt occurred, first, the context switch saves the active context of the thread to the current running thread's stack area, then the context switch loads the next thread's context to the active context and executes the related thread. For sake of simplicity, a thread's context can consist of a program counter, integer registers, and stack pointer, but it can be extensible depending on the application area. The context switch's source code is provided in Appendix C.1. This operation is carrying on until every thread finishes its execution. The wait operation can be done with the `bthread_join` function. Besides, any thread can be appended while the other threads are running, or in the opposite, any thread can finish its execution and detach from the thread wheel while the other threads

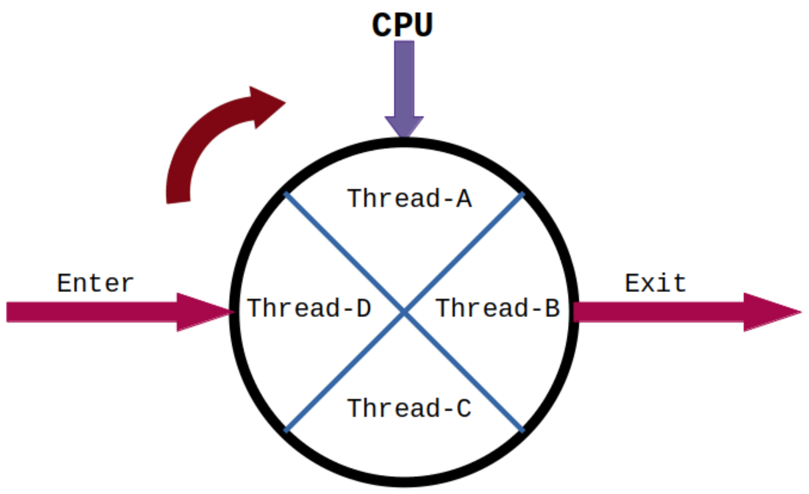


Figure 6.1: Thread Scheduling Approach for Single-core Systems

are running. Listing 6.5 is provided below in order to illustrate the whole execution flow. As mentioned before, bare-threads' syntax is similar to the POSIX threads, the example shows how to create a thread and to wait until the threads have been finished.

Listing 6.5: An example code for bare-threads

```c
#include <bthread.h>
#include "led.h"
#include "util.h"
volatile char done;
bthread_mutex_t mutex;

void *foo(void *a){
    while(!done){
        bthread_mutex_lock(&mutex);
        pause_interrupts();
        toggleLED(0);
        resume_interrupts();
        bthread_mutex_unlock(&mutex);
        sleep_milliseconds(2000);
    }
}
void *bar(void *a){
    int *real_roots = findRealRoots(1, 6,-10, 34, 132,
  31, -10, -1450);
}
void *tar(void *a){
    Complex *cmp_roots = findComplexRoots(1, 6,-10, 34,
  132, 31, -10, -1450);
}

int main(int argc, char **argv){
    struct bthread_t foo_t, bar_t, tar_t;
    bthread_mutex_init(&mutex);
    bthread_create(&foo_t, NULL, foo, (void *)4);
    bthread_create(&bar_t, NULL, bar, (void *)5);
    bthread_create(&tar_t, NULL, tar, (void *)6);

    bthread_join(&bar_t);
    bthread_join(&tar_t);
    done = 1;
    bthread_join(&foo_t);
    return 0;
}
```

In Listing 6.5, a simple operation has been illustrated. In the example, the `bar` thread will run for finding the real roots of the given equation, the `tar` thread will run for finding the complex roots of the given equation. The `foo` thread flashes `LED-0` every 2 seconds to let the programmer know that the program is running. Assume that there is no thread except the `main` function in the thread wheel at the beginning, the initial form of the scheduler is shown in Figure 6.2. The threads are ready to bind to the thread wheel. Each `bthread_create` function adds the thread to the thread wheel as a thread slice. It should be noted that the "main" function manages thread adding operation to the thread wheel. After each `bthread_create` is called, threads are appending to the thread pool one by one, the illustration is shown in Figure 6.3, Figure 6.4, and Figure 6.5, respectively.

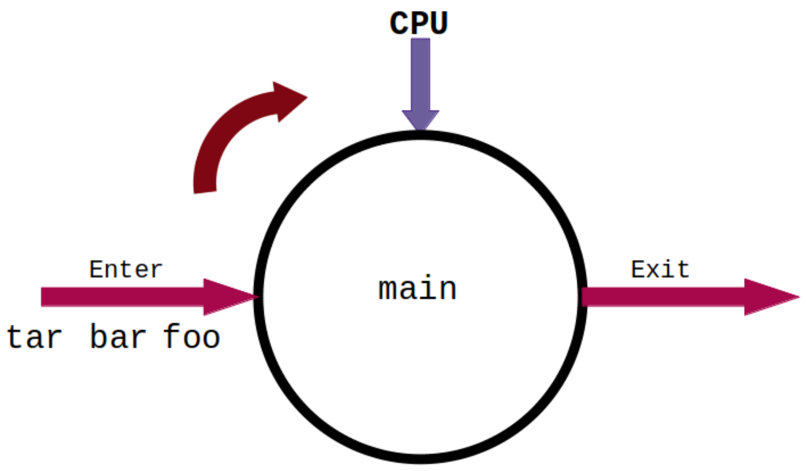


Figure 6.2: Initial form of the scheduler

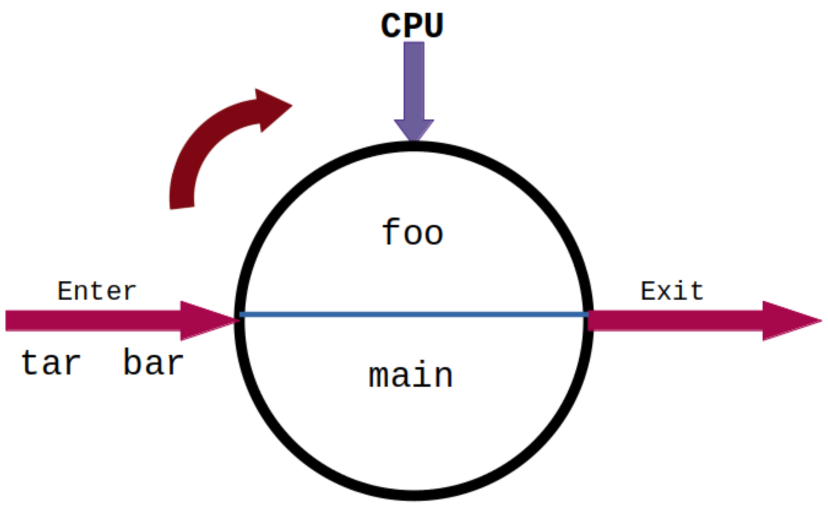


Figure 6.3: The form of the scheduler after foo thread is appended

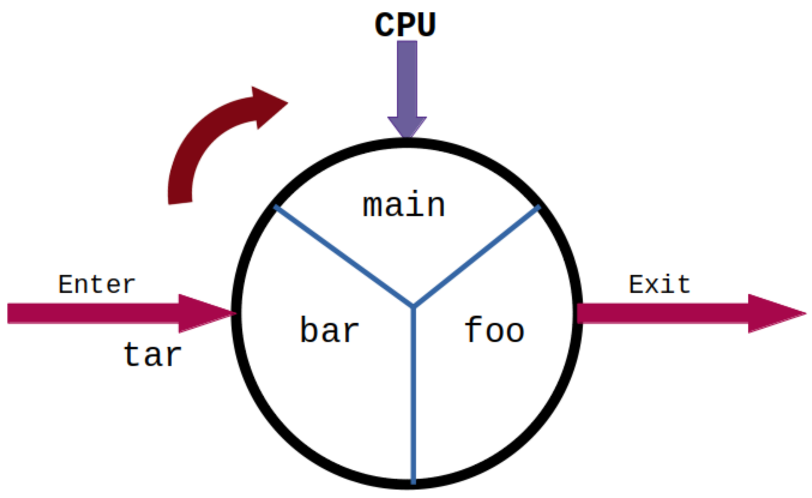


Figure 6.4: The form of the scheduler after bar thread is appended

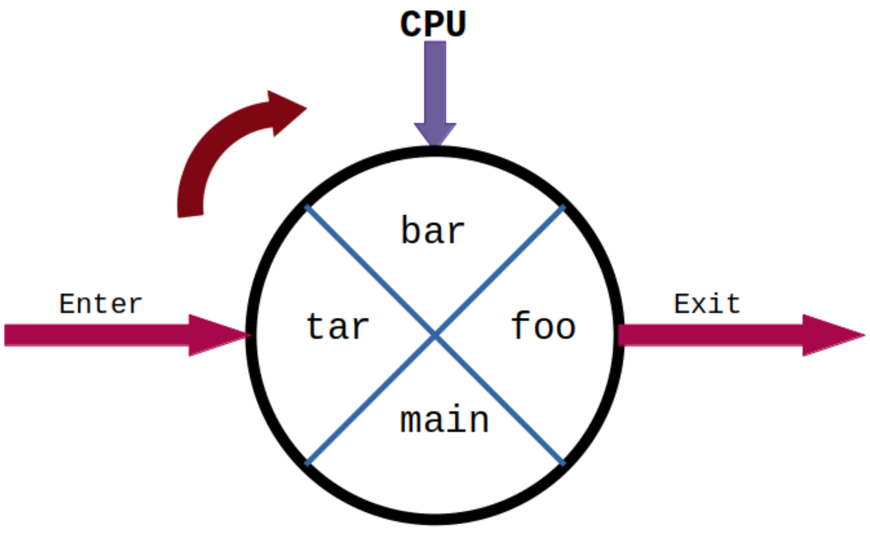


Figure 6.5: The form of the scheduler after all threads are appended

As shown in the figures, initially only the `main` function is running. After a thread is added to the thread wheel, the scheduler starts to schedule and execute threads towards the proposed approach. The state which is shown in Figure 6.5, indicates the `bar` thread is running for a given time interval, when a timer interrupt occurred, the interrupt function makes a context switch between the current thread and the next thread. The working mechanism of the context switch is explained before. It should be noted that this operation does not mean the `bar` thread has been ended, the `bar` thread has paused for the other threads' execution including the main thread. This operation is carrying on until the program finishes its execution. Regarding Listing 6.5, the `bar` and `tar` find the roots of the given equation both real and complex, then their execution will be finished, but the foo thread has to be run until the bar and tar threads finish. Also, the `main` thread will not be ended until all threads have been finished, because the `main` function has to wait for all threads until their executions had been finished. Furthermore, it should be noted that the `foo` thread makes an I/O operation, and thus a critical section is needed because output writes should not be interrupted due to the hardware health considerations. The scheduler is paused in the area between `pause_interrupts` and `resume_interrupts`. So, one can state that `pause_interrupts` pauses the scheduler and `resumes_interrupts` resumes the scheduler from the point of scheduler had been paused. Figure 6.6 and Figure 6.7 show the scheduler states after `bar` and `tar` threads have been finished, respectively. The point that should be noticed, the threads' state which is running or finished does not affect any threads. Thus, concurrency can be provided with this approach.

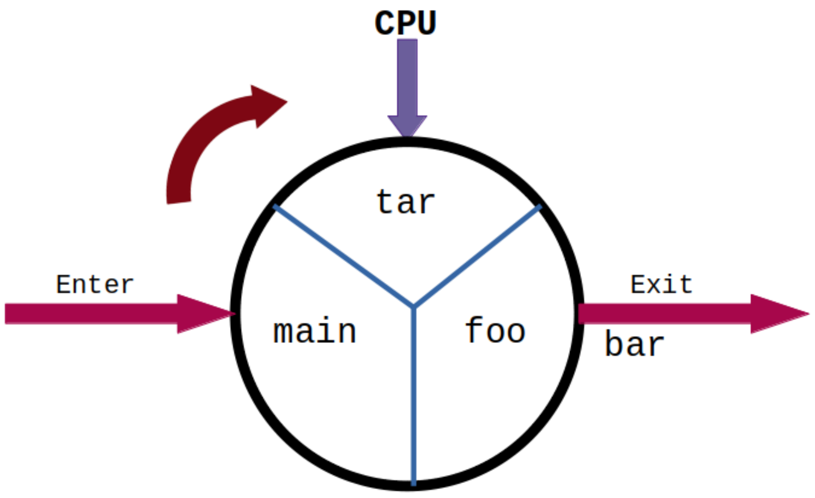


Figure 6.6: The form of the scheduler after bar thread is finished

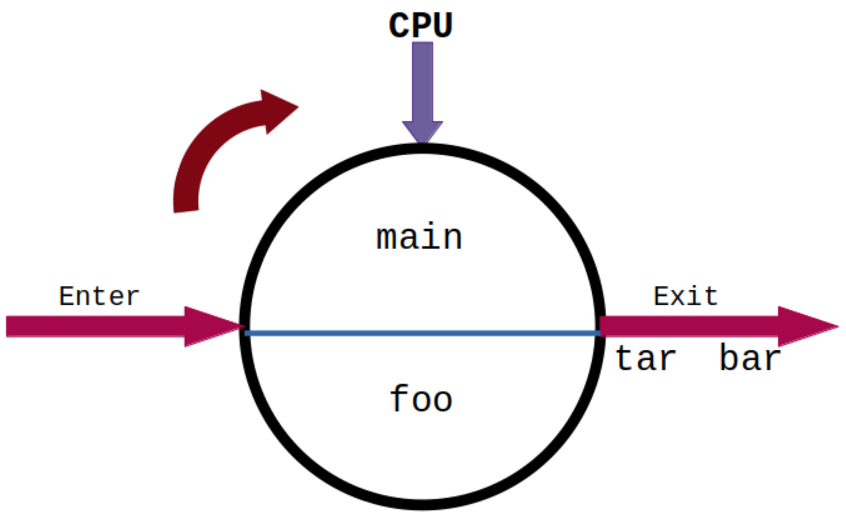


Figure 6.7: The form of the scheduler after bar and tar threads are finished

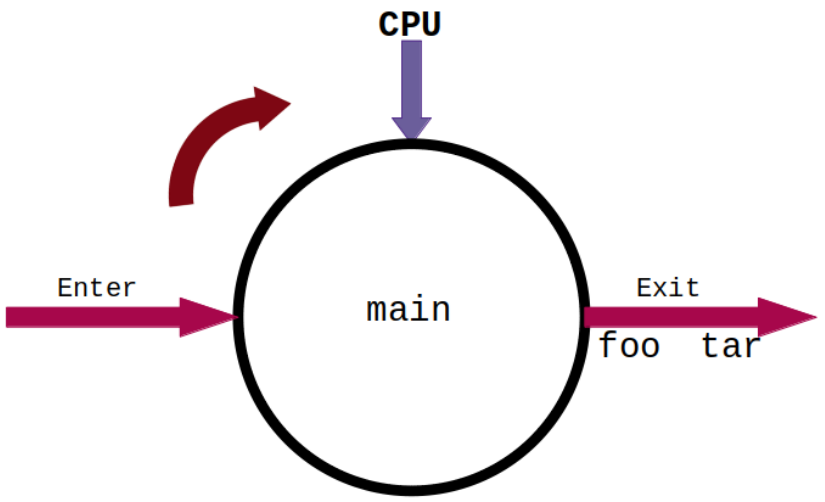


Figure 6.8: The form of the scheduler after foo, bar, and tar threads are finished

When the `foo` thread had been finished successfully, the program can terminate and finish its execution.

I/O Operations

I/O operations denote that the program contains some operations that receive or transmit some data from the hardware. Especially, the input operations generally handle an immediate interrupt that interrupts the execution flow and receives the data from the regarding sources. The bare-threads are supporting I/O operations with an approach. As mentioned before, bare-threads provide multi-threading with a timer interrupt, and thus the interrupts have to be managed in a smart way to avoid data loss. To handle external input interrupts, bare-threads proposes a two-stage interrupt handling mechanism. when an interrupt occurred, the execution flow will jump to the `stage_0_interrupt_handler`, then `stage_0_interrupt_handler` primarily pauses all interrupts including the timer interrupt using by the bare-threads' scheduling mechanism. Then, it saves the current context to the stack and jumps to the `stage_1_interrupt_handler` function. `stage_1_interrupt_handler` function can be considered as a simple switch-case mechanism that handles the interrupt associated with the interrupt cause. The timer interrupt is also handled in there. The important point is that, if the timer interrupt has occurred which is the scheduler's interrupt, the interrupts have to be enabled just after the context switch operation, then the execution flow carry out through the next thread. However, if another interrupt has been occurred, after the interrupt handler function, it will return to the `stage_0_interrupt_handler` function and it restores the saved context, and enables the interrupts, then resumes the execution flow. The proposed interrupt management mechanism provided in Listing 6.6.

Listing 6.6: Interrupt Handling Mechanism for I/O Operations

```c
void stage_0_interrupt_handler()
{
    pause_interrupts();
    save_current_context();
    stage_1_interrupt_handler();
    restore_saved_context();
    resume_interrupts();
    jump();
}

void stage_1_interrupt_handler()
{
    switch(cause)
    {
        case SCHEDULER_INTERRUPT:
            context_switch();
            resume_interrupts();
            jump();
        case EXTERNAL_INTERRUPT:
            interrupt_handler_function();
            return;
        case CUSTOM_INTERRUPT:
            // do something
            return;
    }
}
```

However, output operations generally do not require an interrupt operation that transmits data to peripherals. Thus, bare-threads can operate output operations in a safe zone. This safe zone defines a code area that disables all interrupts and the whole code placed in that safe zone can be executed without any interruption. As mentioned in the sample code, the `foo` thread contains an output operation that toggles the status of `LED-0`, this operation will be executed without any interruption. Furthermore, the output operations have to be protected by synchronization primitives such as a mutex or a semaphore in order to avoid multi-write on the same output register.

Thread Timing

As mentioned before, bare-threads are proposing to use the Round-Robin scheduling algorithm to schedule threads. According to the Round-Robin scheduling algorithm, it works with time slices. The timeline has to divide into equal blocks as called a time slice. Assume that the timeline divided into 10 ms blocks and the time slice value determined as 3. So, this means that each thread is running 30 ms at an operation. Thread timing operation is illustrated in Figure 6.8.
Each time slot represents a 10 ms time interval, for example, the `main` thread executes in the time slots 1 and 3. In other words, the `main` thread starts executing on time slot 1 and its execution paused on time slot 3, then the "main" thread resumes at time slot 13, and pauses again at time slot 15.
In fact, there are many approaches available for adjusting parameters for the Round-Robin scheduling algorithm such as obtaining the optimum values for time slots, time quantum, or even if a dynamic parameter calculation can be made. However, those can be a subject for another research, this approach only proposes the simplest Round-Robin scheduling algorithm for bare-threads.

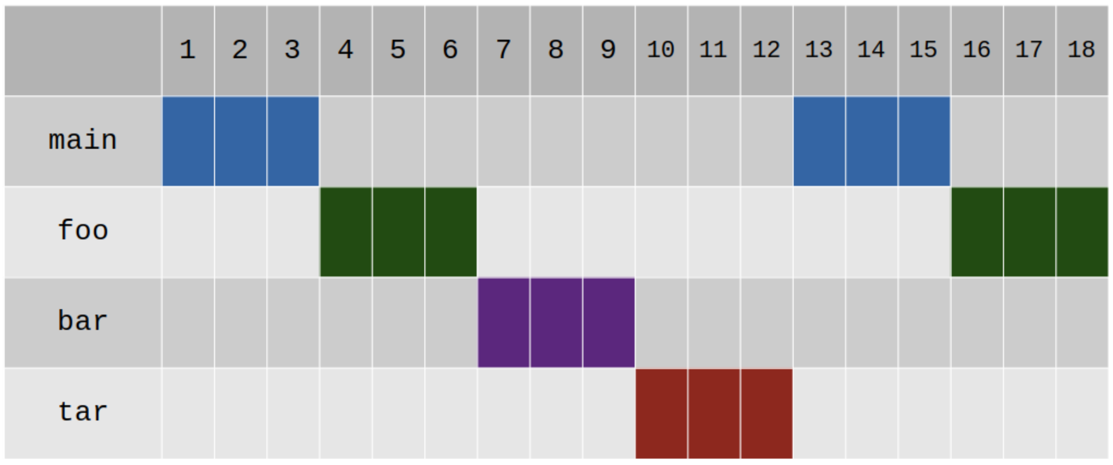
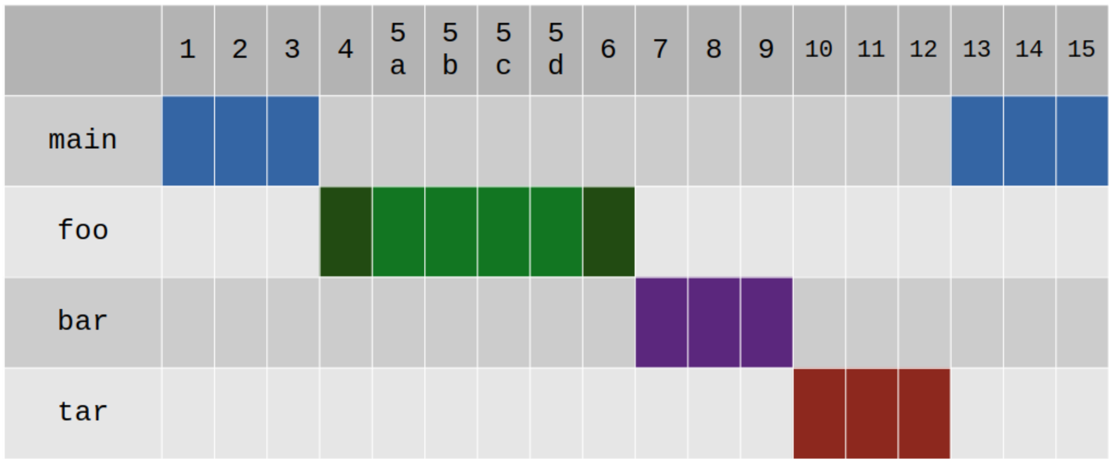


Figure 6.9: Execution timing schedule of threads

However, the `foo` thread contains an output operation that toggles the status of the `LED-0` every 2 seconds, and this operation is protected by `pause_interrupts` and `resume_interrupts` functions. Thus, the scheduler will be paused between the area which covered these two functions. So, the scheduler will be paused while the `foo` thread has been executing. The actual execution timing schedule is illustrated on Figure 6.10.



	1	2	3	4	5 a	5 b	5 c	5 d	6	7	8	9	10	11	12	13	14	15
main	🟦	🟦	🟦													🟦	🟦	🟦
foo				🟩	🟩	🟩	🟩	🟩	🟩									
bar										🟪	🟪	🟪						
tar													🟥	🟥	🟥			

Figure 6.10: Actual execution timing schedule of threads

Memory Occupancy

The `bthread_t` has been already defined and declared in previous sections. The size of the `bthread_t` can be calculated using `sizeof` operator. The total size of `bthread_t` is 65.6 kB, including stack area. As mentioned before PULPissimo has 512 kB of RAM and the simulation has executed eight simple threads concurrently. However, considering each thread that fully occupies 65.6 kB of memory space, eight threads will lead to memory overflow. Therefore, the possible maximum number of concurrent threads is seven for PULPissimo.
However, the multi-threaded program will execute in the bare-metal environment, neither a centralized kernel nor memory management unit has existed. Although the calculation states the maximum number of threads is seven, even seven threads can lead to a memory overflow. Thus, the programmer has to manage the memory considering each thread occupies 65.6 kB of memory space.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This work has presented a case study and a proposed approach in order to provide multi-threading support for bare-metal RISC-V systems. The case study proposes that to port musl C standard library to the RISC-V 32-bit architecture in order to exploit POSIX threads for 32-bit RISC-V bare-metal environment. First, the musl C standard library has been ported to the 32-bit RISC-V architecture, then the ported musl C standard library has tested on the PULPissimo's simulator and the SiFive E2 Series Cores' simulator. Then, a set of incompatibilities has been inferred for both the PULPissimo and the SiFive E2 cores. Musl standard C library has inline assembly calls and those calls contain the instructions which are belong to the RV32A - RISC-V atomic instructions set extensions, where PULPissimo's core which can be RI5CY or Zero-riscy do not support RV32A - atomic instructions set extension. Furthermore, the RISC-V ISA manual[1] states that the atomic instructions cannot able to be emulated with the instruction sets that RI5CY and Zero-riscy supports. Thus, the current version of PULPissimo cannot support RISC-V atomic instructions set extension unless any additional hardware support. The other incompatibility is start-up files which are responsible for igniting the program execution. PULPissimo has its own start-up file called `crt0.S`, also musl has its own start-up files. Musl's start-up files have vast importance for multi-threading because the functions placed in the musl's start-up files initialize and set variables that are using in musl's POSIX threads. Both start-up files cannot able to use at the same time, because both of them have the same symbols. However, this issue has been solved by merging these start-up files without overlap and explained in the thesis. Although SiFive E2 Series cores are supporting the RISC-V 32-bit atomic instruction set extension, musl was not worked on SiFive E2 Series cores[9] because of the lack of a Linux system call API. Musl is built on the Linux system calls API and thus musl C standard library contains the functions that call Linux system calls. In the context of musl's POSIX threads implementation, a set of system calls have to required such as `SYS_clone`, `SYS_futex`, `SYS_set_scheduler`, and etc. However, this thesis examines multi-threading in bare-metal environments and thus neither operating system support nor centralized kernel has existed. Therefore, musl C standard library has not succeeded on SiFive E2 Series cores in the bare-metal mode. Nevertheless, if these system calls will implement to support the dedicated bare-metal platform and the musl C standard library sources will change in that aspect, one can be stated that musl will work on the bare-metal platform which has SiFive E2 Series cores. Furthermore, this situation is also valid for PULPissimo. Even if the atomic instruction set instruction has been supported by additional hardware support, these system calls support required to implement for PULPissimo too. Referenced from these conclusions and inferences, a standalone multi-threading API has been proposed called bare-threads. Bare-threads' design takes into account these constraints encountered on musl's RISC-V 32-bit port. The main motivation of this proposal can be concluded as bullet points as follow,

- To build a standalone multi-threading API that can be easily portable for any architecture or platform, just considering multi-threading demands.
- To provide more flexibility and compatibility to the programmers.
- To create an extensible API that programmers can extend or modify the API with minor modifications through their needings in bare-metal mode.

The bare-threads API have not ported to the PULPissimo, it has proposed as an approach for further approaches or implementations.

7.2 Future Work

The future work can be divided into two main branches. One approach can move forward on musl C standard library branch and the other future work approach can move forward on bare-threads API.
The future work on musl C standard library should be based on the Linux system calls that musl's POSIX threads implementation called. A set of system call supports have to be implemented such as `SYS_clone`, `SYS_futex`, `SYS_set_scheduler`, and etc. with taking into account the platform has been working on.
The future work on bare-threads API can be various, but the most important part is that porting to the PULPissimo. After a successful port to the PULPissimo, the bare-threads API will be proved itself in the context of multi-threading. After

that, there might be several improvements exist, one of them is that modifying the bare-threads syntax to make the same as the POSIX threads' standard. After that, any multi-threaded program that contains POSIX threads will also compatible with bare-metal platforms. Another improvement can be associated with the synchronization primitives. As mentioned in the thesis, the synchronization primitives have implemented in simplicity. Therefore, the synchronization primitives can be advanced through the work of, Peterson et al. (1981)[24]are proposed a way for mutual exclusion problems without using additional hardware support. The Peterson algorithm is simply based on a willingness to enter the critical section and notice for has entered the critical section. Thus the bare-threads API's synchronization primitives can be enhanced with this algorithm.

Appendix A

Makefile Rules

Listing A.1: PULPissimo's Base Makefile Rules for C Applications

```
TOOLCHAIN_DIR = $(CURDIR)/../../../pulp-rv32
BUILD_DIR = $(CURDIR)/build
GCC_DIR = $(TOOLCHAIN_DIR)/lib/gcc/riscv32-unknown-elf/7.1.1/

CHIP = pulpissimo

CC = riscv32-unknown-elf-gcc

PULP_TARGET = mainApp
PULP_APP_SRCS = main.c

PULP_KERNEL_SRCS_STAGE_1 = $(CURDIR)/../../kernel/fll-v1.c $(CURDIR)/../../kernel/freq-domains.c $(CURDIR)/../../kernel/chips/$(CHIP)/soc.c
PULP_LIB_SRCS = $(wildcard $(CURDIR)/../../lib/*.c)
PULP_KERNEL_SRCS_STAGE_2 += $(CURDIR)/../../kernel/init.c $(CURDIR)/../../kernel/kernel.c $(CURDIR)/../../kernel/alloc.c
PULP_KERNEL_SRCS_STAGE_2 += $(CURDIR)/../../kernel/alloc_pool.c $(CURDIR)/../../kernel/irq.c  $(CURDIR)/../../kernel/soc_event.c $(CURDIR)/../../kernel/bench.c
PULP_KERNEL_SRCS_STAGE_2 += $(CURDIR)/../../drivers/uart.c
```

```
PULP_ASM_SRCS = $(CURDIR)/../../kernel/crt0.S $(CURDIR)/../../kernel/irq_asm.S

PULP_SRCS_BIN = $(patsubst %.c,%,$(PULP_KERNEL_SRCS_STAGE_1)) \
$(patsubst %.c,%,$(PULP_LIB_SRCS)) \
$(patsubst %.c,%,$(PULP_KERNEL_SRCS_STAGE_2)) \
$(patsubst %.c,%,$(PULP_APP_SRCS)) \

PULP_ASM_BIN = $(patsubst %.S,%,$(PULP_ASM_SRCS))

PULP_OBJS = $(patsubst $(notdir %.c),%.o,$(PULP_APP_SRCS)) \
$(patsubst $(notdir %.c),%.o,$(PULP_KERNEL_SRCS_STAGE_1)) \
$(patsubst $(notdir %.c),%.o,$(PULP_LIB_SRCS)) \
$(patsubst $(notdir %.c),%.o,$(PULP_KERNEL_SRCS_STAGE_2)) \
$(patsubst $(notdir %.S),%.o,$(PULP_ASM_SRCS))

PULP_LINKER = -T/$(CURDIR)/../../kernel/chips/$(CHIP)/link.ld

PULP_DEFS = -DPULP_CHIP_STR=$(CHIP) -D__PLATFORM__=ARCHI_PLATFORM_RTL -DCONFIG_IO_UART_BAUDRATE=115200 -DCONFIG_IO_UART_ITF=0 -D__RISCV_GENERIC__
PULP_ASM_DEFS = -DLANGUAGE_ASSEMBLY
PULP_CFLAGS = -Os -v -static -nostdinc -g3 -fdata-sections -ffunction-sections -fno-jump-tables -fno-tree-loop-distribute-patterns -fno-exceptions
PULP_CFLAGS += -include $(CURDIR)/../../include/chips/$(CHIP)/config.h -MMD -MP
PULP_LDFLAGS = -v -nostdlib -nodefaultlibs -nostartfiles -Wl,--gc-sections $(PULP_LINKER)
# PULP_LDFLAGS += $(addprefix -L, $(TOOLCHAIN_DIR)/lib/gcc/riscv32-unknown-elf/7.1.1/rv32imcxgap9/ilp32) \
$(addprefix -L, $(TOOLCHAIN_DIR)/lib/gcc/riscv32-unknown-elf/7.1.1)

PULP_INC_DIRS := $(addprefix -I, $(CURDIR)/../../include/chips/$(CHIP)) \
$(addprefix -I, $(CURDIR)/../../include/) \
$(addprefix -I, $(CURDIR)/../../kernel/)

# LIBGCC = $(addprefix -L, $(GCC_DIR)) -lgcc
LIBGCC = $(GCC_DIR)/libgcc.a

all: clean build link

build:
	mkdir build

$(PULP_ASM_BIN): %: %.S
	$(info ***************************************)
	$(info CC $<)
	$(info ***************************************)
	$(TOOLCHAIN_DIR)/bin/$(CC) $(PULP_ASM_DEFS) $(
    PULP_DEFS) $(PULP_CFLAGS) \
	$(PULP_INC_DIRS) \
	$< -c -o $(BUILD_DIR)/$(notdir $@).o

$(PULP_SRCS_BIN): %: %.c
	$(info ***************************************)
	$(info CC $<)
	$(info ***************************************)
	$(TOOLCHAIN_DIR)/bin/$(CC) $(PULP_DEFS) $(
    PULP_CFLAGS) \
	$(PULP_INC_DIRS) \
	$< -c -o $(BUILD_DIR)/$(notdir $@).o

link: $(PULP_ASM_BIN) $(PULP_SRCS_BIN)
	$(info ***************************************)
	$(info Linking..)
	$(info ***************************************)
	$(TOOLCHAIN_DIR)/bin/$(CC) $(PULP_DEFS) $(
    PULP_CFLAGS) $(PULP_LDFLAGS) \
	$(PULP_INC_DIRS) \
	$(addprefix $(BUILD_DIR)/, $(notdir $(PULP_OBJS))) $
    (LIBGCC) -o $(BUILD_DIR)/$(PULP_TARGET)


$(BUILD_DIR)/modelsim.ini:
	ln -s $(VSIM_PATH)/modelsim.ini $@

$(BUILD_DIR)/boot:
	ln -s $(VSIM_PATH)/boot $@

$(BUILD_DIR)/tcl_files:
	ln -s $(VSIM_PATH)/tcl_files $@

$(BUILD_DIR)/waves:
	ln -s $(VSIM_PATH)/waves $@


modelsim-prepare: $(BUILD_DIR)/modelsim.ini  $(BUILD_DIR)/boot $(BUILD_DIR)/tcl_files $(BUILD_DIR)/waves
	$(PULPRT_HOME)/bin/stim_utils.py --binary=$(BUILD_DIR)/$(PULP_TARGET) --vectors=$(BUILD_DIR)/vectors/stim.txt

modelsim: modelsim-prepare
	cd $(BUILD_DIR) && \
	export VSIM_RUNNER_FLAGS="+ENTRY_POINT=0x1c008080 -gLOAD_L2=JTAG -dpicpppath /usr/bin/g++ -permit_unmatched_virtual_intf -gBAUDRATE=115200" && \
	vsim -c -do 'source $(VSIM_PATH)/tcl_files/config/run_and_exit.tcl' -do 'source $(VSIM_PATH)/tcl_files/run.tcl; run_and_exit;'

dump-elf:
	$(TOOLCHAIN_DIR)/bin/riscv32-unknown-elf-readelf -a $(BUILD_DIR)/$(PULP_TARGET)

dump-symbols:
	$(TOOLCHAIN_DIR)/bin/riscv32-unknown-elf-objdump -D $(BUILD_DIR)/$(PULP_TARGET)

dump-code:
	$(TOOLCHAIN_DIR)/bin/riscv32-unknown-elf-objdump -D $(BUILD_DIR)/$(PULP_TARGET)

clean:
	rm -rf $(BUILD_DIR)
```

Listing A.2: PULPissimo's Base Makefile Rules for C++ Applications

```
TOOLCHAIN_DIR = $(CURDIR)/../../../pulp-rv32
BUILD_DIR = $(CURDIR)/build
GCC_DIR = $(TOOLCHAIN_DIR)/lib/gcc/riscv32-unknown-elf/7.1.1/

CHIP = pulpissimo

CC = riscv32-unknown-elf-gcc
CXX = riscv32-unknown-elf-g++

PULP_TARGET = mainApp
PULP_APP_SRCS = main.cpp

PULP_KERNEL_SRCS_STAGE_1 = $(CURDIR)/../../kernel/fll-v1.c $(CURDIR)/../../kernel/freq-domains.c $(CURDIR)/../../kernel/chips/$(CHIP)/soc.c
PULP_LIB_SRCS = $(wildcard $(CURDIR)/../../lib/*.c)
PULP_KERNEL_SRCS_STAGE_2 += $(CURDIR)/../../kernel/init.c $(CURDIR)/../../kernel/kernel.c $(CURDIR)/../../kernel/alloc.c
PULP_KERNEL_SRCS_STAGE_2 += $(CURDIR)/../../kernel/alloc_pool.c $(CURDIR)/../../kernel/irq.c  $(CURDIR)/../../kernel/soc_event.c $(CURDIR)/../../kernel/bench.c
PULP_KERNEL_SRCS_STAGE_2 += $(CURDIR)/../../drivers/uart.c
PULP_ASM_SRCS = $(CURDIR)/../../kernel/crt0.S $(CURDIR)/../../kernel/irq_asm.S

PULP_ASM_BIN = $(patsubst %.S,%,$(PULP_ASM_SRCS))

PULP_SRCS_BIN = $(patsubst %.c,%,$(PULP_KERNEL_SRCS_STAGE_1)) \
$(patsubst %.c,%,$(PULP_LIB_SRCS)) \
$(patsubst %.c,%,$(PULP_KERNEL_SRCS_STAGE_2)) \

PULP_APP_BIN = $(patsubst %.cpp,%,$(PULP_APP_SRCS))

PULP_OBJS = $(patsubst $(notdir %.cpp),%.o,$(PULP_APP_SRCS))
PULP_OBJS += $(patsubst $(notdir %.c),%.o,$(PULP_KERNEL_SRCS_STAGE_1)) \
$(patsubst $(notdir %.c),%.o,$(PULP_LIB_SRCS)) \
$(patsubst $(notdir %.c),%.o,$(PULP_KERNEL_SRCS_STAGE_2)) \
$(patsubst $(notdir %.S),%.o,$(PULP_ASM_SRCS))

PULP_LINKER = -T/$(CURDIR)/../../kernel/chips/$(CHIP)/link.ld

PULP_DEFS = -DPULP_CHIP_STR=$(CHIP) -D__PLATFORM__=ARCHI_PLATFORM_RTL -DCONFIG_IO_UART_BAUDRATE=115200 -DCONFIG_IO_UART_ITF=0 -D__RISCV_GENERIC__
PULP_ASM_DEFS = -DLANGUAGE_ASSEMBLY

PULP_CFLAGS = -Os -static -nostdinc -g3 -fdata-sections -ffunction-sections -fno-jump-tables -fno-tree-loop-distribute-patterns -fno-exceptions
PULP_CFLAGS += -include $(CURDIR)/../../include/chips/$(CHIP)/config.h -MMD -MP

PULP_CXX_FLAGS = -nostdinc -std=c++11 -ffreestanding -fno-threadsafe-statics -fno-unwind-tables

PULP_INC_C_DIRS = $(addprefix -I, $(CURDIR)/../../include/chips/$(CHIP)) \
$(addprefix -I, $(CURDIR)/../../include/) \
$(addprefix -I, $(CURDIR)/../../kernel/) \

PULP_INC_C++_DIRS = $(addprefix -isystem, $(CURDIR)/../../include/) \
$(addprefix -isystem, $(CURDIR)/../../kernel/) \
$(addprefix -isystem, $(CURDIR)/../../include/chips/$(CHIP)) \
$(addprefix -I, $(TOOLCHAIN_DIR)/riscv32-unknown-elf/include/c++/7.1.1) \
$(addprefix -I, $(TOOLCHAIN_DIR)/riscv32-unknown-elf/include/c++/7.1.1/riscv32-unknown-elf) \
$(addprefix -I, $(TOOLCHAIN_DIR)/riscv32-unknown-elf/include/c++/7.1.1/backward) \
$(addprefix -I, $(TOOLCHAIN_DIR)/lib/gcc/riscv32-unknown-elf/7.1.1/include) \
$(addprefix -I, $(TOOLCHAIN_DIR)/lib/gcc/riscv32-unknown-elf/7.1.1/include-fixed) \
$(addprefix -I, $(TOOLCHAIN_DIR)/riscv32-unknown-elf/sys-include) \
$(addprefix -I, $(TOOLCHAIN_DIR)/riscv32-unknown-elf/include)

PULP_LDFLAGS = -Os -static -ffreestanding -std=c++11 -nostdlib -nostartfiles -fno-threadsafe-statics -fno-unwind-tables -Wl,--gc-sections $(PULP_LINKER)
# PULP_LDFLAGS += $(addprefix -L, $(TOOLCHAIN_DIR)/lib/gcc/riscv32-unknown-elf/7.1.1/rv32imcxgap9/ilp32) \
# $(addprefix -L, $(TOOLCHAIN_DIR)/lib/gcc/riscv32-unknown-elf/7.1.1) \

LIBGCC = $(GCC_DIR)/libgcc.a
LIBC = $(TOOLCHAIN_DIR)/riscv32-unknown-elf/lib/libc.a
LIBC++ = $(TOOLCHAIN_DIR)/riscv32-unknown-elf/lib/libstdc++.a

CRUNTIME_BEGIN = $(TOOLCHAIN_DIR)/lib/gcc/riscv32-unknown-elf/7.1.1/crtbegin.o
CRUNTIME_END = $(TOOLCHAIN_DIR)/lib/gcc/riscv32-unknown-elf/7.1.1/crtend.o

all: clean build link

build:
	mkdir build

$(PULP_ASM_BIN): %: %.S
	$(info ***************************************)
	$(info CC $<)
	$(info ***************************************)
	$(TOOLCHAIN_DIR)/bin/$(CC) $(PULP_ASM_DEFS) $(PULP_DEFS) $(PULP_CFLAGS) \
	$(PULP_INC_C_DIRS) \
	$< -c -o $(BUILD_DIR)/$(notdir $@).o

```

```
$(PULP_SRCS_BIN): %: %.c
    $(info **************************************)
    $(info CC $<)
    $(info **************************************)
    $(TOOLCHAIN_DIR)/bin/$(CC) $(PULP_DEFS) $(PULP_CFLAGS) \
    $(PULP_INC_C_DIRS) \
    $< -c -o $(BUILD_DIR)/$(notdir $@).o

$(PULP_APP_BIN): %: %.cpp
    $(info **************************************)
    $(info CXX $<)
    $(info **************************************)
    $(TOOLCHAIN_DIR)/bin/$(CXX) $(PULP_DEFS) $(PULP_CFLAGS) $(PULP_CXX_FLAGS) \
    $(PULP_INC_C++_DIRS) \
    $< -c -o $(BUILD_DIR)/$(notdir $@).o

link: $(PULP_ASM_BIN) $(PULP_SRCS_BIN) $(PULP_APP_BIN)
    $(info **************************************)
    $(info Linking..)
    $(info **************************************)
    $(TOOLCHAIN_DIR)/bin/$(CXX) $(PULP_LDFLAGS) \
    $(CRUNTIME_BEGIN) \
    $(addprefix $(BUILD_DIR)/, $(notdir $(PULP_OBJS))) $(LIBC++) $(LIBC) $(LIBGCC) $(CRUNTIME_END) -o $(BUILD_DIR)/$(PULP_TARGET)

$(BUILD_DIR)/modelsim.ini:
    ln -s $(VSIM_PATH)/modelsim.ini $@

$(BUILD_DIR)/boot:
    ln -s $(VSIM_PATH)/boot $@

$(BUILD_DIR)/tcl_files:
    ln -s $(VSIM_PATH)/tcl_files $@

$(BUILD_DIR)/waves:
    ln -s $(VSIM_PATH)/waves $@


```

```
modelsim-prepare: $(BUILD_DIR)/modelsim.ini  $(BUILD_DIR)/boot $(BUILD_DIR)/tcl_files $(BUILD_DIR)/waves
    $(PULPRT_HOME)/bin/stim_utils.py --binary=$(BUILD_DIR)/$(PULP_TARGET) --vectors=$(BUILD_DIR)/vectors/stim.txt

modelsim: modelsim-prepare
    cd $(BUILD_DIR) && \
    export VSIM_RUNNER_FLAGS="+ENTRY_POINT=0x1c008080 -gLOAD_L2=JTAG -dpicpppath /usr/bin/g++ -permit_unmatched_virtual_intf -gBAUDRATE=115200" && \
    vsim -c -do 'source $(VSIM_PATH)/tcl_files/config/run_and_exit.tcl' -do 'source $(VSIM_PATH)/tcl_files/run.tcl; run_and_exit;'

dump-elf:
    $(TOOLCHAIN_DIR)/bin/riscv32-unknown-elf-readelf -a $(BUILD_DIR)/$(PULP_TARGET)

dump-symbols:
    $(TOOLCHAIN_DIR)/bin/riscv32-unknown-elf-objdump -D $(BUILD_DIR)/$(PULP_TARGET)

dump-code:
    $(TOOLCHAIN_DIR)/bin/riscv32-unknown-elf-objdump -D $(BUILD_DIR)/$(PULP_TARGET)

clean:
    rm -rf $(BUILD_DIR)
```

Listing A.3: Makefile Rules for SiFive with musl integration

```
TOOLCHAIN_DIR = $(CURDIR)/../../../riscv-default-tc
MUSL_DIR = $(CURDIR)/../../../musl-riscv
BUILD_DIR = $(CURDIR)/build
GCC_DIR = $(TOOLCHAIN_DIR)/lib/gcc/riscv32-unknown-elf/10.2.0

CC = riscv32-unknown-elf-gcc

TARGET = mainApp
APP_SRCS = main.c
```

```
ASM_SRCS =

CFLAGS = -Os -v -static -nostdinc -g3 -MMD -MP -
   nostartfiles -march=rv32imafdc -ffreestanding
LINKER = linker.ld
LDFLAGS = -v -nostdlib -nostartfiles -ffreestanding -Wl
   ,-e_start -T$(LINKER)
# LDFLAGS += $(addprefix -L, $(TOOLCHAIN_DIR)/lib/gcc/
   riscv32-unknown-elf/10.2.0/rv32imafc/ilp32f) \
# $(addprefix -L, $(TOOLCHAIN_DIR)/lib/gcc/riscv32-
   unknown-elf/10.2.0)

INC_DIRS := $(addprefix -I, $(MUSL_DIR)/include)

SRCS_BIN = $(patsubst %.c,%,$(APP_SRCS))
ASM_BIN = $(patsubst %.S,%,$(ASM_SRCS))

OBJS = $(patsubst $(notdir %.c),%.o,$(APP_SRCS)) $(
   patsubst $(notdir %.S),%.o,$(ASM_SRCS))

CRT_OBJS = $(MUSL_DIR)/lib/crt1.o $(MUSL_DIR)/lib/crti.o
    $(MUSL_DIR)/lib/crtn.o

LIBGCC = $(GCC_DIR)/libgcc.a
LIBC = $(MUSL_DIR)/lib/libc.a

RISCV_VP_DIR = /home/mert/Desktop/Thesis/riscv-vp/vp/
   build/bin
QEMU_PATH = /opt/riscv-qemu/bin
GEM5_PATH = /home/mert/Desktop/Thesis/fresh/gem5
all: clean build link

build:
    mkdir build

$(ASM_BIN): %: %.S
    $(info ****************************************)
    $(info CC $<)
    $(info ****************************************)
    $(TOOLCHAIN_DIR)/bin/$(CC) $(CFLAGS) \
    $(INC_DIRS) \
	$< -c -o $(BUILD_DIR)/$(notdir $@).o

$(SRCS_BIN): %: %.c
	$(info **************************************)
	$(info CC $<)
	$(info **************************************)
	$(TOOLCHAIN_DIR)/bin/$(CC) $(CFLAGS) \
	$(INC_DIRS) \
	$< -c -o $(BUILD_DIR)/$(notdir $@).o

link: $(ASM_BIN) $(SRCS_BIN)
	$(info **************************************)
	$(info Linking..)
	$(info **************************************)
	$(TOOLCHAIN_DIR)/bin/$(CC) $(CFLAGS) $(LDFLAGS) \
	$(CRT_OBJS) $(addprefix $(BUILD_DIR)/, $(notdir $(OBJS))) $(LIBGCC) $(LIBC) $(LIBGCC) -o $(BUILD_DIR)/$(TARGET)

rv:
	rv-sim -c -m $(BUILD_DIR)/$(TARGET)

dump-elf:
	$(TOOLCHAIN_DIR)/bin/riscv32-unknown-elf-readelf -a $(BUILD_DIR)/$(TARGET)

dump-symbols:
	$(TOOLCHAIN_DIR)/bin/riscv32-unknown-elf-objdump -D $(BUILD_DIR)/$(TARGET) > dumpsym1.txt

dump-code:
	$(TOOLCHAIN_DIR)/bin/riscv32-unknown-elf-objdump -D $(BUILD_DIR)/$(TARGET)

clean:
	rm -rf $(BUILD_DIR)
```

Listing A.4: Makefile Rules for PULPissimo with musl integration

```
TOOLCHAIN_DIR = $(CURDIR)/../../../pulp-rv32
MUSL_DIR = $(CURDIR)/../../../pulp-rv32-musl
BUILD_DIR = $(CURDIR)/build
```

```
GCC_DIR = $(TOOLCHAIN_DIR)/lib/gcc/riscv32-unknown-elf/7.1.1/

CHIP = pulpissimo

CC = riscv32-unknown-elf-gcc

PULP_TARGET = mainApp
PULP_APP_SRCS = main.c

PULP_KERNEL_SRCS_STAGE_1 = $(CURDIR)/../../kernel/fll-v1.c $(CURDIR)/../../kernel/freq-domains.c $(CURDIR)/../../kernel/chips/$(CHIP)/soc.c
PULP_LIB_SRCS = $(wildcard $(CURDIR)/../../lib/*.c)
PULP_KERNEL_SRCS_STAGE_2 += $(CURDIR)/../../kernel/init.c $(CURDIR)/../../kernel/kernel.c $(CURDIR)/../../kernel/alloc.c
PULP_KERNEL_SRCS_STAGE_2 += $(CURDIR)/../../kernel/alloc_pool.c $(CURDIR)/../../kernel/irq.c  $(CURDIR)/../../kernel/soc_event.c $(CURDIR)/../../kernel/bench.c
PULP_KERNEL_SRCS_STAGE_2 += $(CURDIR)/../../drivers/uart.c $(CURDIR)/../../kernel/crt/crt1.c
PULP_ASM_SRCS = $(CURDIR)/../../kernel/crt/crt0.S $(CURDIR)/../../kernel/crt/crti.S $(CURDIR)/../../kernel/irq_asm.S

PULP_SRCS_BIN = $(patsubst %.c,%,$(PULP_KERNEL_SRCS_STAGE_1)) \
$(patsubst %.c,%,$(PULP_LIB_SRCS)) \
$(patsubst %.c,%,$(PULP_KERNEL_SRCS_STAGE_2)) \
$(patsubst %.c,%,$(PULP_APP_SRCS)) \

PULP_ASM_BIN = $(patsubst %.S,%,$(PULP_ASM_SRCS))

PULP_OBJS = $(patsubst $(notdir %.c),%.o,$(PULP_APP_SRCS)) \
$(patsubst $(notdir %.c),%.o,$(PULP_KERNEL_SRCS_STAGE_1)) \
$(patsubst $(notdir %.c),%.o,$(PULP_LIB_SRCS)) \
```

```makefile
$(patsubst $(notdir %.c),%.o,$(PULP_KERNEL_SRCS_STAGE_2)) \
$(patsubst $(notdir %.S),%.o,$(PULP_ASM_SRCS))

PULP_LINKER = -T/$(CURDIR)/../../kernel/chips/$(CHIP)/link.ld

PULP_DEFS = -DPULP_CHIP_STR=$(CHIP) -D__PLATFORM__=ARCHI_PLATFORM_RTL -DCONFIG_IO_UART_BAUDRATE=115200 -DCONFIG_IO_UART_ITF=0 -D__RISCV_GENERIC__ -DSYSCALL_NO_INLINE
PULP_ASM_DEFS = -DLANGUAGE_ASSEMBLY -DSYSCALL_NO_INLINE
PULP_CFLAGS = -Os -v -static -nostdinc -g3 -fdata-sections -ffunction-sections -fno-exceptions -fno-asynchronous-unwind-tables
PULP_CFLAGS += -include $(CURDIR)/../../include/chips/$(CHIP)/config.h -MMD -MP
PULP_LDFLAGS = -v -nostdlib -nodefaultlibs -nostartfiles $(PULP_LINKER)
# PULP_LDFLAGS += $(addprefix -L, $(TOOLCHAIN_DIR)/lib/gcc/riscv32-unknown-elf/7.1.1/rv32imcxgap9/ilp32) \
$(addprefix -L, $(TOOLCHAIN_DIR)/lib/gcc/riscv32-unknown-elf/7.1.1)

PULP_INC_DIRS := $(addprefix -I, $(CURDIR)/../../include/chips/$(CHIP)) \
$(addprefix -I, $(CURDIR)/../../include/) \
$(addprefix -I, $(CURDIR)/../../kernel/) \
$(addprefix -I, $(CURDIR)/../../include/crt/) \
$(addprefix -I, $(MUSL_DIR)/include)

# LIBGCC = $(addprefix -L, $(GCC_DIR)) -lgcc
# LIBC = $(addprefix -L, $(MUSL_DIR)/lib/) -lc

LIBGCC = $(GCC_DIR)/libgcc.a
LIBC = $(MUSL_DIR)/lib/libc.a

all: clean build link

build:
	mkdir build
```

```

$(PULP_ASM_BIN): %: %.S
    $(info ***************************************)
    $(info CC $<)
    $(info ***************************************)
    $(TOOLCHAIN_DIR)/bin/$(CC) $(PULP_ASM_DEFS) $(PULP_DEFS) $(PULP_CFLAGS) \
    $(PULP_INC_DIRS) \
    $< -c -o $(BUILD_DIR)/$(notdir $@).o

$(PULP_SRCS_BIN): %: %.c
    $(info ***************************************)
    $(info CC $<)
    $(info ***************************************)
    $(TOOLCHAIN_DIR)/bin/$(CC) $(PULP_DEFS) $(PULP_CFLAGS) \
    $(PULP_INC_DIRS) \
    $< -c -o $(BUILD_DIR)/$(notdir $@).o

link: $(PULP_ASM_BIN) $(PULP_SRCS_BIN)
    $(info ***************************************)
    $(info Linking..)
    $(info ***************************************)
    $(TOOLCHAIN_DIR)/bin/$(CC) $(PULP_DEFS) $(PULP_CFLAGS) $(PULP_LDFLAGS) \
    $(PULP_INC_DIRS) \
    $(LIBGCC) $(LIBC) $(LIBGCC) \
    $(addprefix $(BUILD_DIR)/, $(notdir $(PULP_OBJS))) $(LIBC) -o $(BUILD_DIR)/$(PULP_TARGET)


$(BUILD_DIR)/modelsim.ini:
    ln -s $(VSIM_PATH)/modelsim.ini $@

$(BUILD_DIR)/boot:
    ln -s $(VSIM_PATH)/boot $@

$(BUILD_DIR)/tcl_files:
    ln -s $(VSIM_PATH)/tcl_files $@

$(BUILD_DIR)/waves:
	ln -s $(VSIM_PATH)/waves $@


modelsim-prepare: $(BUILD_DIR)/modelsim.ini  $(BUILD_DIR)/boot $(BUILD_DIR)/tcl_files $(BUILD_DIR)/waves
	$(PULPRT_HOME)/bin/stim_utils.py --binary=$(BUILD_DIR)/$(PULP_TARGET) --vectors=$(BUILD_DIR)/vectors/stim.txt

modelsim: modelsim-prepare
	cd $(BUILD_DIR) && \
	export VSIM_RUNNER_FLAGS="+ENTRY_POINT=0x1c008080 -gLOAD_L2=JTAG -dpicpppath /usr/bin/g++ -permit_unmatched_virtual_intf -gBAUDRATE=115200" && \
	vsim -c -do 'source $(VSIM_PATH)/tcl_files/config/run_and_exit.tcl' -do 'source $(VSIM_PATH)/tcl_files/run.tcl; run_and_exit;'

dump-elf:
	$(TOOLCHAIN_DIR)/bin/riscv32-unknown-elf-readelf -a $(BUILD_DIR)/$(PULP_TARGET)

dump-symbols:
	$(TOOLCHAIN_DIR)/bin/riscv32-unknown-elf-objdump -D $(BUILD_DIR)/$(PULP_TARGET) > dumpsym1.txt

dump-code:
	$(TOOLCHAIN_DIR)/bin/riscv32-unknown-elf-objdump -D $(BUILD_DIR)/$(PULP_TARGET)

clean:
	rm -rf $(BUILD_DIR)
```

Appendix B

C Source Codes

Listing B.1: crt1.c code for PULPissimo

```c
#include <features.h>

#define START "_start_musl"

#include "crt_arch.h"
#include <stdio.h>

int main();
void _init() __attribute__((weak));
void _fini() __attribute__((weak));
_Noreturn int __libc_start_main(int (*)(), int, char **,
    void (*)(), void(*)(), void(*)());

extern void (*const __init_array_start)(void), (*const
   __init_array_end)(void);


void _start_musl_c(int argc, char **argv)
{
    __libc_start_main(main, argc, argv, _init, _fini, 0)
   ;
}
```

Listing B.2: crt-arch.h code for PULPissimo

```c
__asm__(
".text\n"
".global " START "\n"
".type " START ",%function\n"
START ":\n"
".weak __global_pointer$\n"
".hidden __global_pointer$\n\t"
".option push\n"
".option norelax\n\t"
"lla gp, __global_pointer$\n"
".option pop\n\t"
"mv a0, sp\n"
".weak _DYNAMIC\n"
".hidden _DYNAMIC\n\t"
"lla a1, _DYNAMIC\n\t"
"andi sp, sp, -16\n\t"
"jal " START "_c"
);
```

Listing B.3: Producer-Consumer Problem implemented with Protothreads

```
/*
 * Copyright (c) 2004-2005, Swedish Institute of
   Computer Science.
 * All rights reserved.
 *
 * Author: Adam Dunkels <adam@sics.se>
 *
 * $Id: example-buffer.c,v 1.5 2005/10/07 05:21:33 adam
   Exp $
 */

#include "pt-sem.h"
#include "led.h"

static struct pt_sem full, empty;

static
PT_THREAD(producer(struct pt *pt))
{
  static int produced;
  PT_BEGIN(pt);

  for(produced = 0; produced < NUM_ITEMS; ++produced) {
    PT_SEM_WAIT(pt, &full);
    add_to_buffer(turnOnLed());
    PT_SEM_SIGNAL(pt, &empty);
  }
  PT_END(pt);
}

static
PT_THREAD(consumer(struct pt *pt))
{
  static int consumed;
  PT_BEGIN(pt);
  for(consumed = 0; consumed < NUM_ITEMS; ++consumed) {
    PT_SEM_WAIT(pt, &empty);
    turnOffLed(get_from_buffer());
    PT_SEM_SIGNAL(pt, &full);
  }
  PT_END(pt);
}

static
PT_THREAD(driver_thread(struct pt *pt))
{
  static struct pt pt_producer, pt_consumer;
  PT_BEGIN(pt);
  PT_SEM_INIT(&empty, 0);
  PT_SEM_INIT(&full, BUFSIZE);
  PT_INIT(&pt_producer);
  PT_INIT(&pt_consumer);
  PT_WAIT_THREAD(pt, producer(&pt_producer) &
           consumer(&pt_consumer));
  PT_END(pt);
}


int
main(void)
{
  struct pt driver_pt;
  PT_INIT(&driver_pt);
  while(PT_SCHEDULE(driver_thread(&driver_pt))) {
    /*
     * When running this example on a multitasking
   system, we must
     * give other processes a chance to run too and
   therefore we call
     * usleep() resp. Sleep() here. On a dedicated
   embedded system,
     * we usually do not need to do this.
     */
  }

  return 0;
}
```

Listing B.4: Source code of bare-thread API's mutexes and semaphores

```
typedef struct bthread_mutex_t{
    volatile unsigned bool lock;
}bthread_mutex_t;

typedef struct bthread_sem_t{
    volatile unsigned short count;
    struct bthread_mutex_t _mtx;
}bthread_sem_t;

void bthread_mutex_init(struct bthread_mutex_t *mtx);
void bthread_mutex_lock(struct bthread_mutex_t *mtx);
void bthread_mutex_unlock(struct bthread_mutex_t *mtx);

void bthread_sem_init(bthread_sem_t *sem, unsigned int
   initialValue);
void bthread_sem_post(struct bthread_sem_t *sem);
void bthread_sem_wait(struct bthread_sem_t *sem);
unsigned short bthread_sem_get_value(struct
   bthread_sem_t *sem);

void bthread_mutex_init(struct bthread_mutex_t *mtx)
{
    mtx->lock = FALSE;
}

void bthread_mutex_lock(struct bthread_mutex_t *mtx)
```

```c
{
    while(mtx->lock);
    __mem_barrier();
    mtx->lock = TRUE;
}

void bthread_mutex_unlock(struct bthread_mutex_t *mtx)
{
    __mem_barrier();
    mtx->lock = FALSE;
}

void bthread_sem_init(struct bthread_sem_t *sem,
   unsigned int initialValue)
{
    sem->count = initialValue;
}

void bthread_sem_post(struct bthread_sem_t *sem)
{
    bthread_mutex_lock(&sem->_mtx);
    sem->count++;
    bthread_mutex_unlock(&sem->_mtx);
}

void bthread_sem_wait(struct bthread_sem_t *sem)
{
    while(sem->count == 0);
    bthread_mutex_lock(&sem->_mtx);
    sem->count--;
    bthread_mutex_unlock(&sem->_mtx);
}

unsigned short bthread_sem_get_value(struct
   bthread_sem_t *sem)
{
    return sem->count;
}
```

Appendix C

RISC-V Assembly Source Codes

Listing C.1: Context Switch Source Code

```
contextswitch:
    // store registers
    sw    x1,4(a0)       // ra
    sw    x2,8(a0)       // sp
    sw    x8,32(a0)      // s0
    sw    x9,36(a0)      // s1
    sw    x18,72(a0)     // s2
    sw    x19,76(a0)     // s3
    sw    x20,80(a0)     // s4
    sw    x21,84(a0)     // s5
    sw    x22,88(a0)     // s6
    sw    x23,92(a0)     // s7
    sw    x24,96(a0)     // s8
    sw    x25,100(a0)    // s9
    sw    x26,104(a0)    // s10
    sw    x27,108(a0)    // s11

    // store pc
    la    t0,_resume
    sw    t0,0(a0)

    // restore other registers (NOTE: callee saved only
   + ra)
    lw    x1,4(a1)       // ra
    lw     x2,8(a1)        // sp
    lw     x8,32(a1)       // s0
    lw     x9,36(a1)       // s1
    lw     x18,72(a1)      // s2
    lw     x19,76(a1)      // s3
    lw     x20,80(a1)      // s4
    lw     x21,84(a1)      // s5
    lw     x22,88(a1)      // s6
    lw     x23,92(a1)      // s7
    lw     x24,96(a1)      // s8
    lw     x25,100(a1)     // s9
    lw     x26,104(a1)     // s10
    lw     x27,108(a1)     // s11

    // load new program counter and perform context
   switch
    lw     t0,0(a1)
    jr     t0

    _resume:

    ret
```

Bibliography

[1] Krste Asanovic Andrew Waterman. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. SiFive Inc, CS Division, EECS Department, University of California, Berkeley, CA, 2019 (cit. on pp. 4, 6, 7, 37, 59).

[2] *Atomic (computer science)*. URL: `https://encyclopedia2.thefreedictionary.com/Atomic+(computer+science)` (cit. on p. 7).

[3] Pasquale Davide Schiavone Andreas Traber Michael Gautschi. *RI5CY: User Manual*. Micrel Lab and Multiherman Lab, University of Bologna, Integrated Systems Lab, ETH Zurich. Apr. 2019 (cit. on p. 8).

[4] Pasquale Davide Schiavone. *zero-riscy: User Manual*. Micrel Lab and Multiherman Lab, University of Bologna, Integrated Systems Lab, ETH Zurich. Jan. 2018 (cit. on p. 8).

[5] The PULP Team. *PULPissimo: Datasheet*. May 2020 (cit. on pp. 10, 12).

[6] *PULPissimo*. Micrel Lab and Multiherman Lab, University of Bologna, Integrated Systems Lab, ETH Zurich. URL: `https://github.com/pulp-platform/pulpissimo` (cit. on pp. 10, 29).

[7] *SiFive*. URL: `https://www.sifive.com/` (cit. on p. 11).

[8] SiFive Inc. *SiFive E20 Manual, Version: 20G1.03.00*. June 2020 (cit. on p. 11).

[9] SiFive Inc. *SiFive E21 Manual, Version: 20G1.03.00*. June 2020 (cit. on pp. 11, 60).

[10] SiFive Inc. *SiFive E24 Manual, Version: 20G1.03.00*. June 2020 (cit. on p. 13).

[11] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 1.00. Arpaci-Dusseau Books, Aug. 2018 (cit. on p. 16).

[12] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. 1st. USA: No Starch Press, 2010. ISBN: 1593272200 (cit. on p. 17).

[13] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. «Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems». In: *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*. Boulder, Colorado, USA, Nov. 2006. URL: `http://dunkels.com/adam/dunkels06protothreads.pdf` (cit. on pp. 19, 39).

[14] Jeff Johnston Corinna Vinschen. *Newlib: C Standard Library*. URL: `https://sourceware.org/newlib/` (cit. on p. 22).

[15] Rich Felker. *musl libc*. URL: `https://musl.libc.org/` (cit. on p. 22).

[16] Mentor Graphics Corporation. *ModelSim Command Reference Manual*. Version Software Version 10.4c. URL: `https://www.microsemi.com/document-portal/doc_view/136364-modelsim-me-10-4c-command-reference-manual-for-libero-soc-v11-7` (cit. on p. 23).

[17] Free Software Foundation Inc. *Installing GCC: Configuration*. May 2021. URL: `https://gcc.gnu.org/install/configure.html` (cit. on p. 25).

[18] *PULP RISC-V GNU Toolchain*. Micrel Lab and Multiherman Lab, University of Bologna, Integrated Systems Lab, ETH Zurich. URL: `https://github.com/pulp-platform/pulp-riscv-gnu-toolchain` (cit. on p. 26).

[19] *PULP Runtime*. Micrel Lab and Multiherman Lab, University of Bologna, Integrated Systems Lab, ETH Zurich. URL: `https://github.com/pulp-platform/pulp-runtime` (cit. on p. 26).

[20] Michael J. Clark. *RV8 RISC-V Simulator for x86-64*. URL: `https://michaeljclark.github.io/` (cit. on p. 30).

[21] Michael J. Clark. *Musl 1.1.18 32-bit RISC-V Port*. 2018. URL: `https://github.com/michaeljclark/musl-riscv` (cit. on p. 33).

[22] Hoang M. Le Vladimir Herdt Daniel Große. «Extensible and Configurable RISC-V based Virtual Prototype». In: (2018) (cit. on p. 46).

[23] Hoang M. Le Vladimir Herdt Daniel Große. *RISCV-VP Examples: Simple Scheduler*. 2018. URL: `https://github.com/agra-uni-bremen/riscv-vp/tree/master/sw/simple-scheduler` (cit. on p. 47).

[24] G.L. Peterson. «Myths About the Mutual Exclusion Problem». In: *Information Processing Letters* (June 1981) (cit. on p. 61).