# RISC-V : an FPGA implementation for general purpose prototyping hardware

**Author : Federico Pozzana**

**Supervisor : Guido Masera**
**Co-supervisor : Elia Delledonne**
**Co-supervisor : Fabrizio Fraternali**

A thesis presented for the Master degree in
Electronic engineering


Department of Electronics and Telecommunication
Politecnico di Torino
18 December 2020

# Abstract

The RISCV project began in 2010 at UC Berkely. It set out to provide a flexible, open source instruction set architecture to offer an alternative to proprietary ISAs which require non disclosures agreements and royalties to be used. Currently the RISCV foundation own, maintain and publish intellectual property related to RISCV's definition [1]. Although it started out in academia the RISCV instruction set architecture has been adopted by numerous companies, such as

- SiFive
- Codasip
- lowRISC

and many more [2]. With such strong industrial adoption, RISCV poses itself as a rival to the ARM's hegemony in the microcontroller world.

This study, made in collaboration with Maxim Integrated [3], aims to integrate general purpose peripherals (such as I2C, SPI, UART, etc...) with an available RISCV IP provided by an internal group. This work is intended to be a feasibility study for the aformentioned core; possible applications for the microcontroller could be

- motor control applications
- replace big control finite state machines
- industrial communication applications

The methodology followed these steps

- RISCV IP RTL study
- publicly available toolchain/IDE selection
- compiler performance evaluation
- peripheral dedicated firmware development
- peripherals integration
- develop regression environment for RTL validation
- CORDIC peripheral development
- LINTing of newly developed RTL
- synthesis step with Design Compiler

The conclusions can be observed in the "Conclusion" chapter.

# Acknowledgements

I would like to thank my supervisor, Professor Guido Masera. His knowledge and guidance helped me during the technical challenges that this thesis presented.

I would like to thank my co-supervisors, Elia Delledonne and Fabrizio Fraternali, for their guidance during this master thesis project. They were always present to steer me in the right direction from start to finish.

I would also like to thank the people that have been close to me for all these years. I am particularly grateful to Laura, whom had the arduous task of putting up with me for the past decade.

Last, but not least, I would like to thank my family. Their constant support throughout my university studies have been invaluable.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This thesis, done in collaboration with Maxim Integrated [3], is intended as a feasibility study for an internal version of a RISC-V microcontroller, investigate the area - performance tradeoff for this particular design and to consider if such microcontroller could be used in applications such as

- replacing big control finite state machines
- motor control applications
- industrial communication applications

The starting point of this research is the register transfer level description of the microcontroller. From the RTL description my contribution range from firmware development to logic synthesis passing through peripherals integration, in particular

**IDE setup**

In order to be able to develop C code for the peripheral's drivers an IDE has to be set up. In this case Eclipse [5] has been chosen due to the support for the RISCV toolchain [6]. An in depth overview for the IDE selection and set up process is on chapter 3.

**Benchmarking activity**

To gauge the performance of the microcontroller a benchmarking activity is necessary. Benchmarking is useful in order to understand possible application uses for the design at hand. An exhaustive account for the benchmarking acrivity is present on chapter 5.

**FPGA porting activity**

To gauge the area occupation and to validate the RTL design on hardware an FPGA porting process has been performed. The FPGA board Arty A7 - 100T [7] has been selected for porting/testing the overall design. The complete FPGA porting process is present on chapter 6.

**General purpose peripherals integration**

In order to be able to communicate with the external world peripherals such as UART, SPI and I2C have to be integrated to the microcontroller itself. An RTL description for the previously mentioned peripherals was already present in the Maxim Integrated's IP bank. Chapters 7 and 8 are an in depth overview of the peripheral integration and verification activities respectively.

**Application specific peripheral development**

Maxim's need to find a more efficient way of performing operations such as sine, cosine and square root led to the development of a custom peripheral. An exhaustive description of the development, testing and benckmarking activity for the custom peripheral, based on the CORDIC algorithm, is present on chapter 9.

**Front-end activity**

To have a more precise indication on the area occupation and a clearer understanding on the clock frequency obtainable a synthesis step has been performed. Alongside the synthesis step the front-end includes a LINTing activity for the RTL code. A complete overview for the front-end process is present on chater 10.

### 1.0.1 Tools

Given the wide variety of topics touched during this research a number of software tools have been used, in particular

- Xcelium Logic Simulation from Cadence for RTL simulation
- AscentLint from RealIntent for LINTing
- Vivado Design suite from Xilinx for FPGA synthesis
- Design Compiler from Synopsys for ASIC synthesis
- Eclipse IDE for firmware development

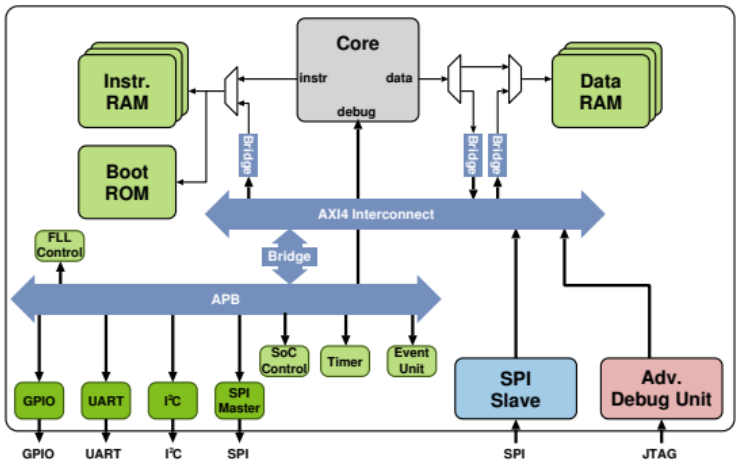
### 1.0.2 PULP

The PULP platform [4] is a research group between ETH Zurich and University of Bologna which develops open hardware compliant with the RISCV instruction set architecture. The group has developed during the last seven years single core, multi core and multi cluster based microcontrollers. For this research project a Maxim's [3] internal revision of the PULPino microcontroller is used. The microcontroller, given as IP, has

- two memory ports (one for a data RAM and one for an instruction ROM)
- a JTAG port (used to debug the microcontroller)
- one AHB interface (used by high speed peipherals)
- one APB interface (used by low speed peripherals)

Figure 1.1 gives an overall structure of the microcontroller.

Figure 1.1: reference microcontroller architecture



### Memory ports

The two memory ports, one used for the ROM and one used for the RAM, are interfaces used by the microcontroller to exchange informations with the two memories. In particular

- the Read-Only-Memory is used to store both program code and constant data
- the Random-Access-Memory is used to store non constant data, like data returned by a function

A code snippet for both memories can be found in the appendix chapter. By specification the memories have to be synchronous to a clock and compliant with the signals coming out of the two interfaces, which are similar for both memories.

| port name | lenght | inuput/output | description |
|---|---|---|---|
| ram_clk | 1 bit | output | RAM clock |
| ram_address | log2(num_bytes/4) bits | output | RAM address |
| ram_ce | 1 bit | output | RAM chip enable |
| ram_wdata | 32 bits | output | RAM write data |
| ram_be | 4 bits | output | RAM byte enable |
| ram_we | 1 bit | output | RAM write enable |
| ram_rdata | 32 bits | input | RAM read data |

Table 1.1: RAM interface portmap

| port name | lenght | inuput/output | description |
|---|---|---|---|
| rom_clk | 1 bit | output | ROM clock |
| rom_address | log2(num_bytes/4) bits | output | ROM address |
| rom_ce | 1 bit | output | ROM chip enable |
| rom_rdata | 32 bits | input | ROM read data |

Table 1.2: ROM interface portmap

The following figures represent a read and a write operation on the RAM memory (the ROM memory have similar access timing, only for the read operation).

Figure 1.2: RAM read operation

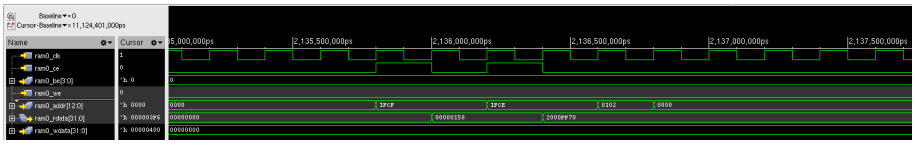


Figure 1.3: RAM write operation

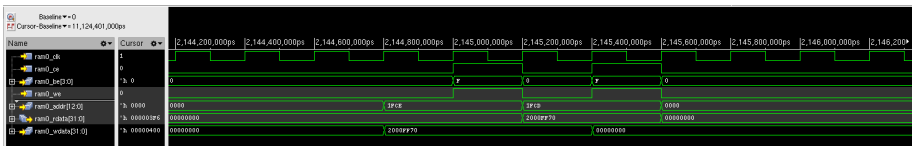
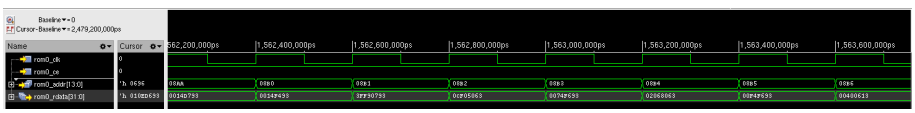


Figure 1.4: ROM read operation



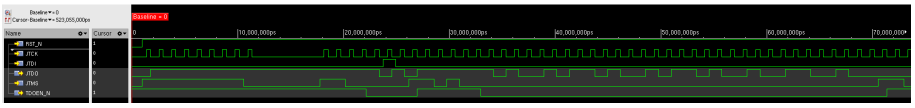
### JTAG port

JTAG is an industry standard for verifying designs and testing PCB's after manufacture [8]. Almost every microcontroller has a JTAG port for debugging purposes. The JTAG port signals follow closely the JTAG standard, in particular

| port name | lenght | inuput/output | description |
|---|---|---|---|
| jtck | 1 bit | input | JTAG clock |
| jtdi | 1 bit | input | JTAG data input |
| jtms | 1 bit | input | JTAG test mode select |
| tdoen_n | 1 bit | output | JTAG data output enable |
| jtdo | 1 bit | output | JTAG data output |

Table 1.3: JTAG interface portmap

Figure 1.5: JTAG operation



### AHB-APB interfaces

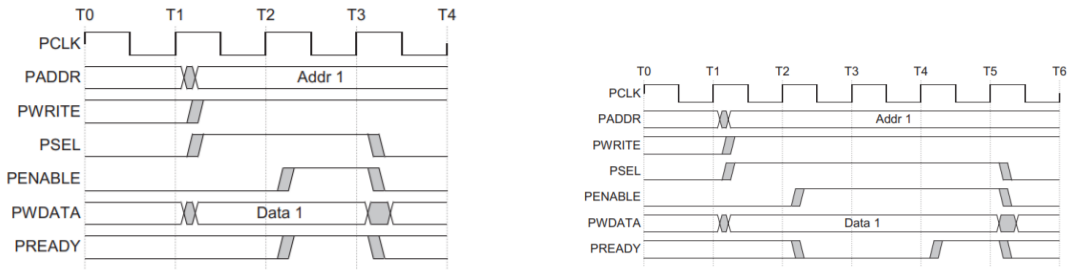
Both AHB and APB interfaces are bus protocols compliant with the AMBA [9] (advanced microcontroller bus architecture) specifications. The two interfaces are used for

- AHB : used for fast peripherals
- APB : used for slow peripherals

The two protocol are similar in the sense that both present an address phase and a subsequent data phase; the difference is the presence of additional features, such as

- burst write
- write without wait states
- extended bus widths (64, 128, .. to 1024 bits)

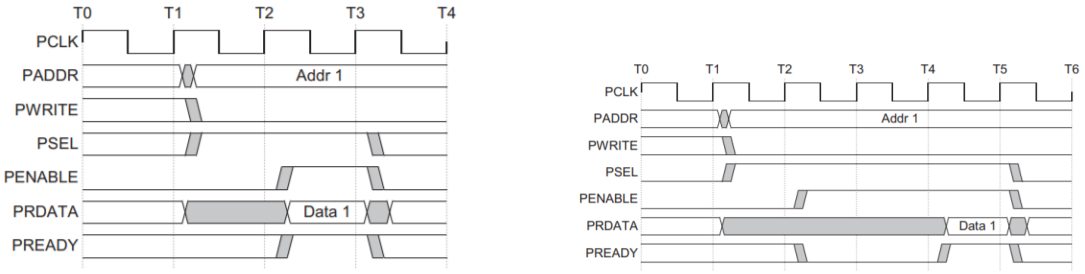
These features make the AHB bus more sutable for high bandwidth, high complexity peripherals. The following figures represent both read and write transactions, with and without wait states. For a more comprehensive view of the APB bus standard refer to [10].




(a) APB write without wait states

(b) APB write with wait states

Figure 1.6: APB write operation




(a) APB read without wait states

(b) APB read with wait states

Figure 1.7: APB read operation

**Microcontroller core**

For this specific microcontroller [11] two different cores can be used

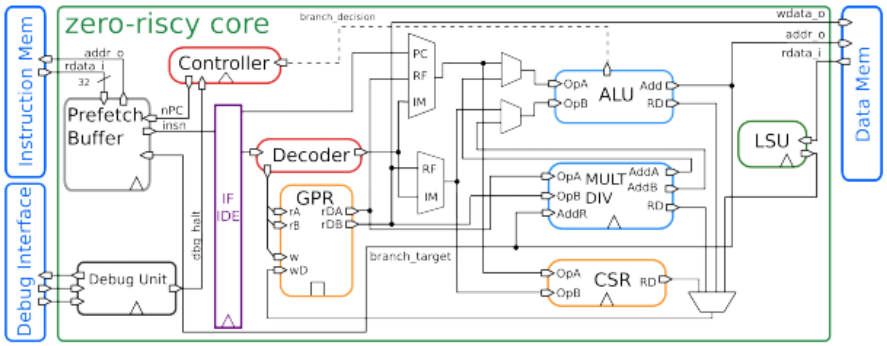
- a 4-stage pipelined core named RI5CY
- a 2-stage pipelined core named ZERO-RISCY (currently known as IBEX)

The two cores present the same external interface and thus are plug-compatible.
Since this project is intended for low overhead applications ZERO-RISCY is used since it provides the same functionalities in term of instruction supported on a smaller footprint. Figure 2.2 gives an overall structure of the ZERO-RISCY core.

The ZERO-RISCY core presents the following features

- parametrized to support either the RV32I or the RV32E instruction set
- RV32C and RV32M instruction set availability
- 2-stage pipeline (IF - ID/EX)
- latch based or flip-flop based register file

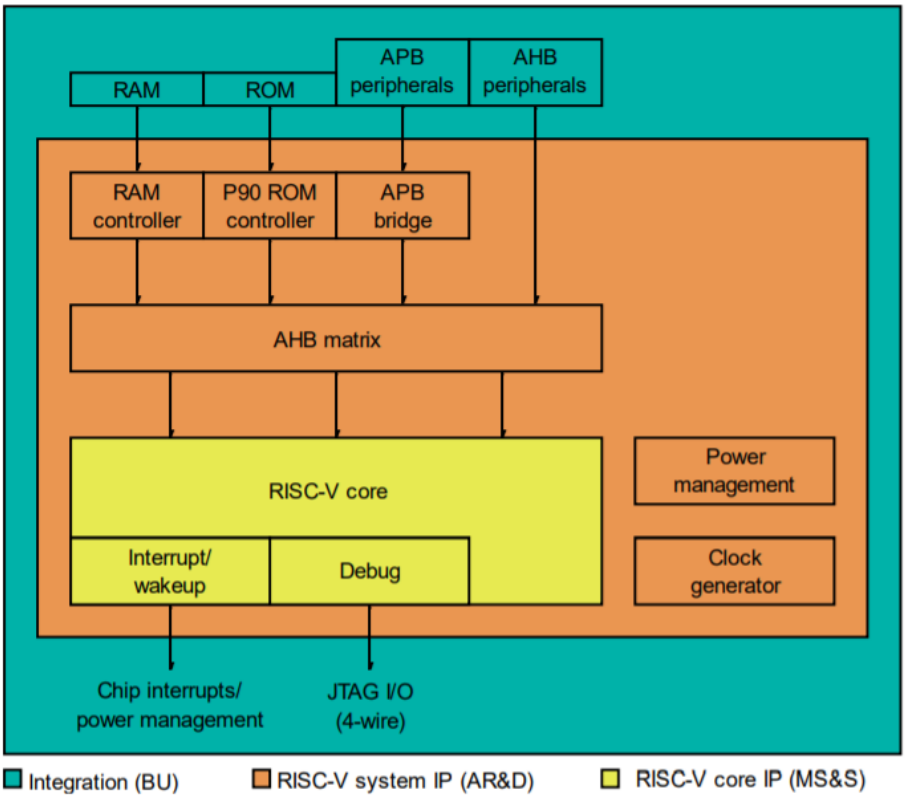
Figure 1.8: ZERO-RISCY architecture




## 1.0.3 RISCV IP

The starting point for this master thesis in terms of IP is the complete microcontroller, with interfaces for the two memories (ROM and RAM) alongside AHB and APB buses. Figure 1.12 represent the delivered IP (orange block). This thesis's work is represented in figure 1.12 by the cyan block.

Figure 1.9: RISCV IP



# Chapter 2

# Memories description

The first step in the integration process is to develop both ROM and RAM memories to be connected to the RISCV IP. In our case both memory controllers have been attached to the ahb bus; this design choice forces the design of both memories to be synchronous. Timing diagram are in figure 2.1 to 2.3. The code snippet relative to both ROM and RAM memories can be found in the appendix chapter.

Both memories have been synthetized on Vivado in order to make sure that they inferred a memory based on BRAM instead of logic cell's registers. This characteristic is crucial for the following reasons

- Using logic cell's registers is inefficient with respect to chip usage
- A requirement for this project is to be able to reprogram the ROM on the synthetized design without re-synthetizing the whole microcontroller on FPGA

### 2.0.1 Portmap and timing diagram

In order to understand the principle of both ROM and RAM memories a brief description of their portmap and timing diagram is necessary.

| port name | lenght | inuput/output | description |
|---|---|---|---|
| ram_clk | 1 bit | input | RAM clock |
| ram_address | log2(num_bytes/4) bits | input | RAM address |
| ram_ce | 1 bit | input | RAM chip enable |
| ram_wdata | 32 bits | input | RAM write data |
| ram_be | 4 bits | input | RAM byte enable |
| ram_we | 1 bit | input | RAM write enable |
| ram_rdata | 32 bits | output | RAM read data |

Table 2.1: RAM portmap

| port name | lenght | inuput/output | description |
|---|---|---|---|
| rom_clk | 1 bit | input | ROM clock |
| rom_address | log2(num_bytes/4) bits | input | ROM address |
| rom_ce | 1 bit | input | ROM chip enable |
| rom_rdata | 32 bits | output | ROM read data |

Table 2.2: ROM portmap

Figure 2.1: RAM read operation

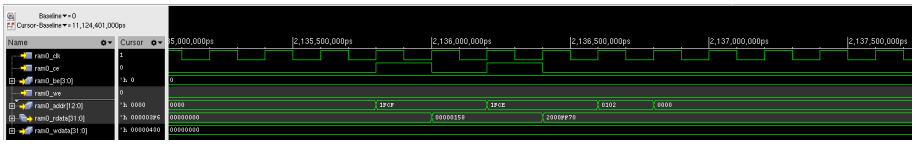


Figure 2.2: RAM write operation

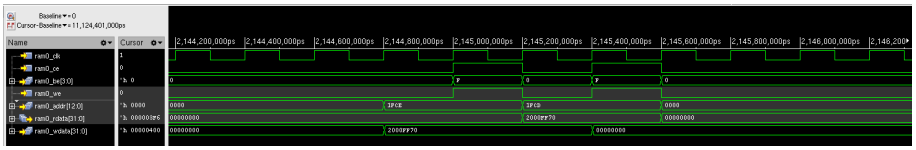
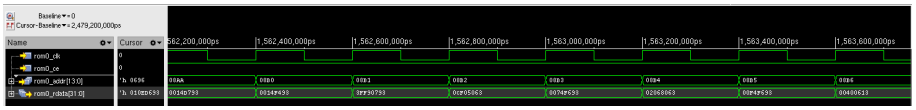


Figure 2.3: ROM read operation



### 2.0.2 Vivado's synthesis

In order to make sure that the code for ROM and RAM memories, available in the appendix chapter, would infer block RAMs and not map the two memories on flip-flops, the Vivado's synthesis report has to checked. The synthesis report confirmed that the RTL code for both memories infers block RAMs.

2. Memory

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| Block RAM Tile | 24 | 0 | 135 | 17.78 |
| RAMB36/FIFO* | 24 | 0 | 135 | 17.78 |
| RAMB36E1 only | 24 | | | |
| RAMB18 | 0 | 0 | 270 | 0.00 |

```
Module rom
Detailed RTL Component Info :
+---Registers :
32 Bit      Registers := 1
+---RAMs :
512K Bit          RAMs := 1
+---Muxes :
2 Input      32 Bit         Muxes := 6
```

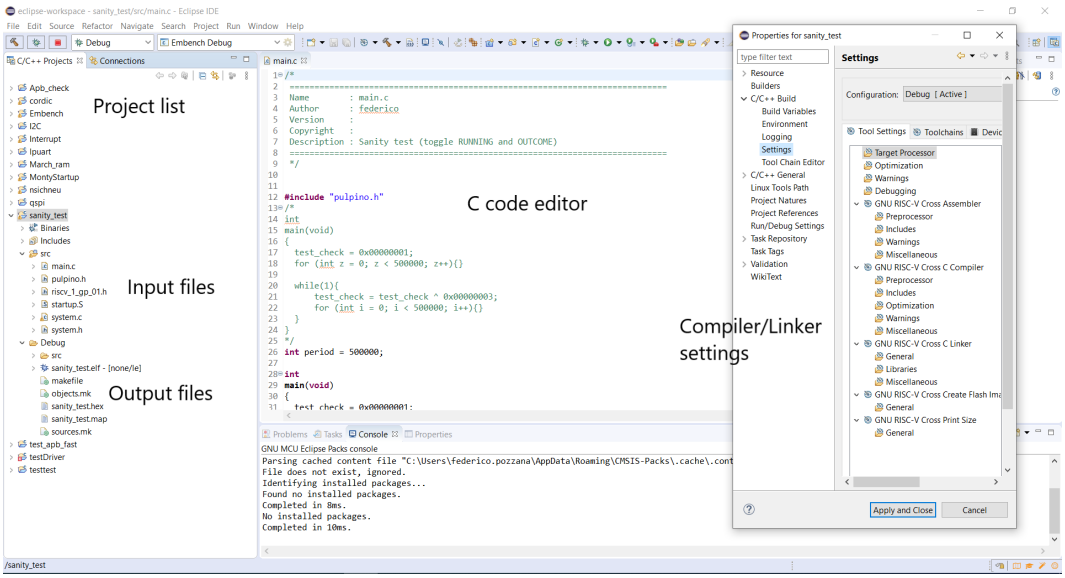
```
Module ram
Detailed RTL Component Info :
+---Registers :
32 Bit      Registers := 1
+---RAMs :
256K Bit          RAMs := 1
+---Muxes :
2 Input      32 Bit         Muxes := 6
```

# Chapter 3

# IDE setup

An integrated development environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development [12]. In this specific case the IDE is a graphical user interface that links every step of firmware development (C code correctness check, compilation, linking, hex file generation).

Figure 3.1: Eclipse IDE




### 3.0.1 IDE

The first step in developing executable code for the microcontroller is to set up the IDE. The IDE of choice is Eclipse due to the following reasons

- open software that can be customized as needed
- RISC-V toolchain integration within the IDE [13]
- ease of integrating startup/linker files (internal version of them were already available)

In particular, for a microcontroller, as toolchain is intended one or more applications capable of taking as input a list of C files and giving as output either a binary or an HEX file. Usually the intermediate steps performed are

- C code correctness check, done by Eclipse
- C code compilation, done by the RISC-V compiler linked to Eclipse

- Linking, done by a Maxim propietary script. Eclipse is capable of pointing to a specific linker file chosen by the user
- HEX file generation, done by Eclipse

In order to prove the correctness of the IDE setup a C program has been developed, further informations are in chapter 3.

In order to fill the ROM with executable code with the

```
initial $readmemh("file.mem", rom);
```

instruction a binary memory file has to be created. Eclipse's output is an HEX file, in the following paragraph are presented

- HEX file format
- HEX to MEM transition

### 3.0.2 HEX file

The output of the compile and linking phase is a .hex [14] file. An example of a line of a .hex file would be :0300300002337A1E. Relatively to the previous hex line example the .hex file format presents the following characteristics

| HEX line breakdown | | |
|---|---|---|
| value | type | description |
| : | start code | start of a new line |
| 03 | byte count | two hex digits indicating the number of bytes in the data fields |
| 0030 | address | four hex digits representing the 16-bit beginning memory address offset of the data |
| 00 | record type | two hex digits defining the meaning of the data field |
| 02337A | data | a sequence of n bytes of data represented by 2n hex digits |
| 1E | checksum | two hex digits used to verify the record has no errors |

The record type can assume values from 00 to 05 with different meanings. In particular if record type is

| Record type breakdown | |
|---|---|
| value | description |
| 00 | the data field contains data |
| 01 | end of file |
| 02 | the data field contains a base address to be added to each subsequent address relative to data type (extended segment address) |
| 03 | CS:IP registers content (only for 80x86 processors) |
| 04 | the data field contains the upper 16 bits for a 32 bits address to be applied to each subsequent data type (extended linear address) |
| 05 | address to be loaded into EIP register (only for 80386 or higher CPU's) |

The checksum field is computed adding each byte in the hex line and then making the two's complement of the LSB of the obtained sum. In our example line is easy to see that the sum of 03 + 00 + 30 + 00 + 02 + 33 + 7A is E2, and its two's complement is 1E.

In order to be able to decode the content of the .hex file an internal script was provided. The script, a .hex file parser, takes as input the .hex file produced by the compiling and linking phase and produces the memory file in a .mem extension. This .mem file is then read by the ROM during simulation. This script have been used for 8-bit microcontroller projects; since the microcontroller is a 32-bit one the internal script has been modified in order to output program data word by word instead of byte by byte.

# Chapter 4

# Initial tests

### 4.0.1 Testbench

The testbench is organized in order to reflect the real world implementation relative to chip hierarchy; in particular figure 3.1 shows the testbench organization. This testbench structure allows to utilize the same stimuli both on RTL simulation and on FPGA simulation. Performing the same test in both cases (RTL and FPGA) allows to spot an error much easier than if the setup needed two different simulations. In particular the testbench has the following blocks

- fpga_digtop is the top entity of the microcontroller. It comprises both the core and the peripherals attached to it
- fpga_top is the top entity of the overall system. It comprises both the fpga_digtop and the analog models of external components present on FPGA, such as a reset button and an oscillator
- tc_fpga_top is the top entity of the stimuli block. This block drives and checks the signals going to and coming from the fpga_top block. During simulation it generates stimuli for the microcontroller and checks its actuations; this allows to perform an auto-check for each simulation. The auto-checking feature in turn makes possible to change RTL code and simulate the result without looking at any specific waveform, but just checking the .log file to see if the test has finished successfully or not.
- tb_fpga_top is the top entity connecting tc_fpga_top and fpga_top

Running and outcome are service signals coming from the microcontroller which indicates the state of the program being tested. In each test program those signals are set at the beginning and end of the program; the module tc_fpga_top then checks the signal outcome at the end of the program (signaled by the running signal) to confirm the successful exit.

### 4.0.2 Initial tests

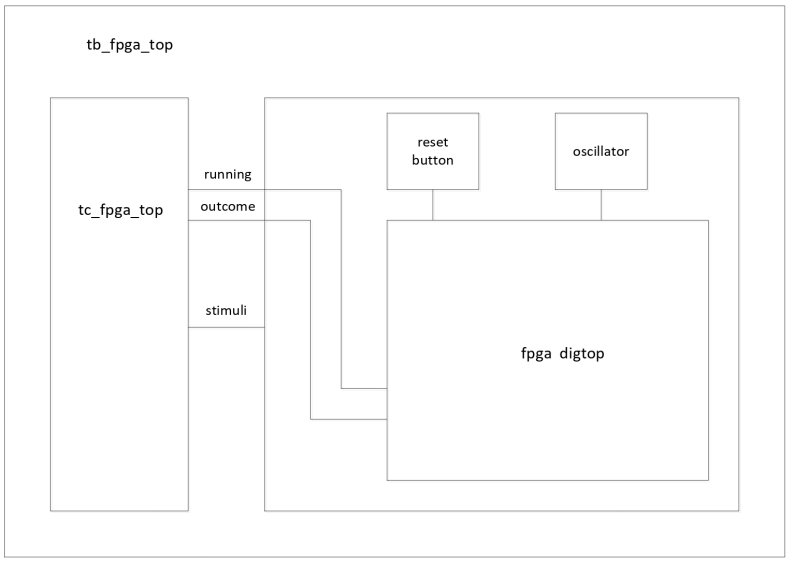
In order to guarantee the correctness of both the toolchain setup and the RISCV IP two simple programs have been developed. They are respectively

- March C RAM test
- Random APB test

To be noted is the auto-checking nature of these tests; in fact at both the start and the end of the program the signals running and outcome are set/cleared depending on the outcome of the program itself. This feature allows us to build a regression list of tests used to check automatically (looking at the simulator log) the correctness of the microcontroller in case of modifications.

The following sections enter in detail for the above mentioned tests.

Figure 4.1: testbench architecture




### March C test

March tests are a group of algorithms used to test memories. This group of tests are characterized by reading and writing test patterns in memory locations in both an increasing and decreasing fashion. The aim for this particular test is to check the connection between the RAM memory and the microcontroller; if the march test fails it means that a bug is present between the microcontroller and the RAM memory. A C program compliant with these specifications has been developed in order to test both the RAM memory and its connection to the microcontroller. A code snippet for this test can be found in the appendix chapter.

The aim of the *March C test* is to perform a systematic read/write test on each memory cell. During this test if an error is spotted the signal test_running is lowered to 0, likewise for the test_outcome signal. Setting to 0 or 1 test_running and test_outcome will set to 0 or 1 the corresponding signal specified in the testbench section. For this test two 32 bit registers are attached to the APB bus used to store informations about the possible error through the writeAPB(address,value) function.

Figure 4.2: March C - RAM ramp

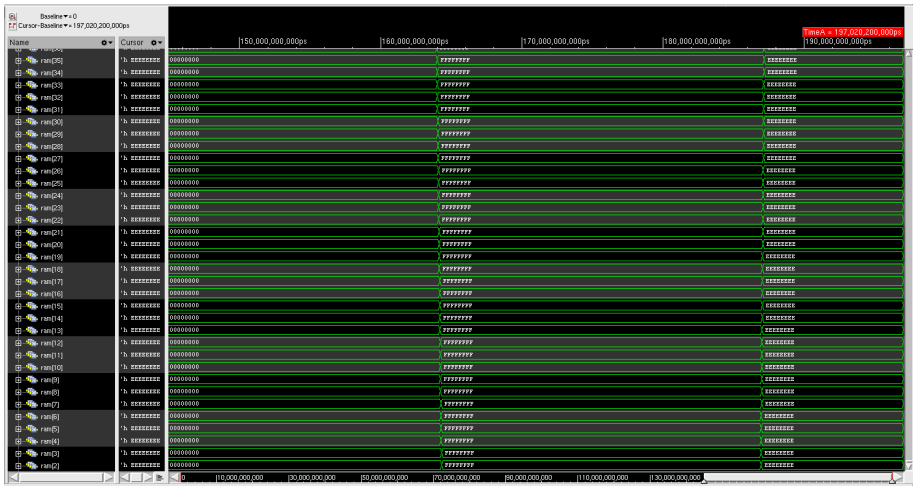


Figure 4.3: March C - RAM read

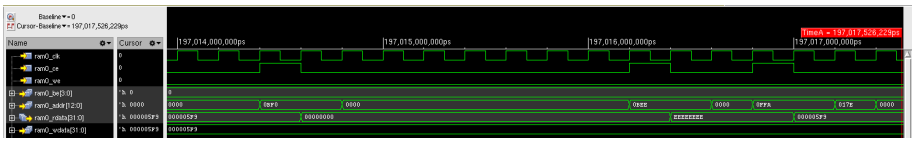
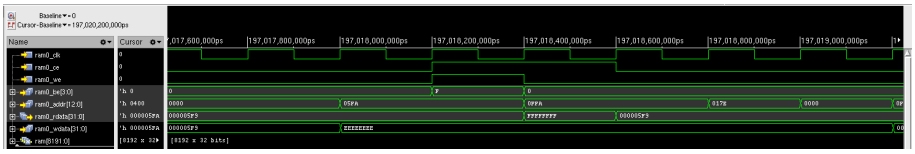


Figure 4.4: March C - RAM write



### Random APB test

Following the reasons of the previous test (check correctness of both the toolchain setup and the RISCV IP) a random test for the APB bus has been developed. A code snippet can be found in the appendix chapter.

In particular the test checks random addresses of the APB bus comprised in an interval defined by the variables start_addr and end_addr. The test writes on the APB bus ten random values on 32 bits at ten random addresses; after the write phase it reads at those random addresses to check the value. This process is repeated, thus providing a more exhaustive test. In order for this test to be effective a register file is attached to the APB bus; the register file holds the random values produced by the write phase and provides them during the read phase. As stated before setting to 0 or 1 test_running and test_outcome will set to 0 or 1 the corresponding signal specified in the testbench section. Figures from 4.5 to 4.7 are the waveforms for the RTL simulation for the APB test. Figures 4.6 and 4.7 show a write operation with values 0x7044FFE7 and 0x01B1B099; the same values are then read from the register file in figure 4.5 (first two results in rf_prdata[31:0]).

Figure 4.5: APB test - RF read

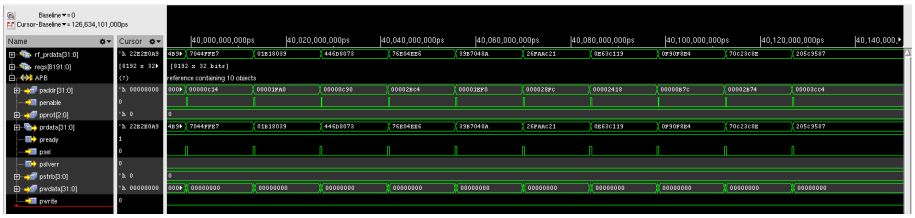


Figure 4.6: APB test - RF write 1

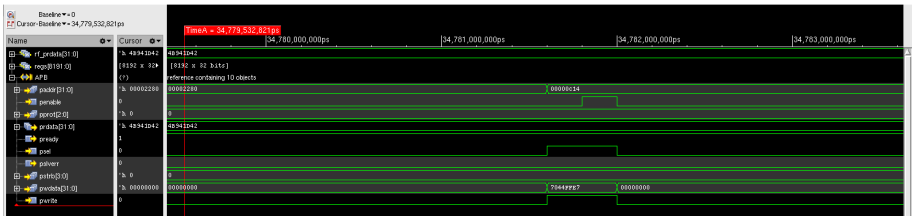
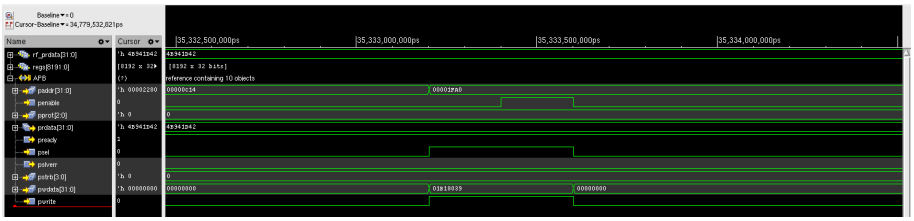


Figure 4.7: APB test - RF write 2



# Chapter 5

# Embench tests

In order to gauge the performances of both the microcontroller and the gcc compiler a stress test is needed. The most common benchmark program to test CPU's performance is the Dhrystone [15]; this benchmark however has some shortcomings

- it features code that is not representative of real life programs
- it is susceptible to compiler optimizations

These two problems are addressed by the Embench [16] open benchmarks. In particular the Embench benchmark collection, developed by both industry and university, provides the following benefits

- it is a collection of 19 benchmark programs, capable of representing real life processor usage
- ease of porting and free to use
- benchamrk programs developed specifically with memory constraints in mind (64 KiB ROM and 64 KiB RAM)

| test | comment | branch | memory | computing |
|---|---|---|---|---|
| crc32 | CRC error checking 32b | high | med | low |
| cubic | cubic root solver | low | med | med |
| edn | general filter | low | high | med |
| huffbench | compress/decompress | med | med | med |
| matmult-int | integer matrix multiply | med | med | med |
| miniver | matrix inversion | high | low | med |
| mont64 | montgomery multiplication | low | low | high |
| nbody | satellite N body, large data | med | low | high |
| nettle-aes | AES encrypt/decrypt | med | high | low |
| nettle-sha256 | SHA256 digest | low | med | med |
| nsichneu | large - petri net | med | high | low |
| picojpeg | JPEG | med | med | high |
| qrduino | QR codes | low | med | med |
| sglib-combined | Simple Generic Library for C | high | high | low |
| slre | regex | high | med | med |
| st | statistics | med | low | high |
| statemate | state machine (car window) | high | high | low |
| ud | LUD composition int | med | low | high |
| wikisort | merge sort | med | med | med |

Table 5.1: Embench tests distribution

Due to these reasons I decided to test the microcontroller performances with the Embench tests.

### 5.0.1 Tests setup

It has been a straightforward process to set up the Embench tests due to the following reasons

- Common main.c file, used to trigger start/stop test
- Extensive documentation available at Embench github page [16]

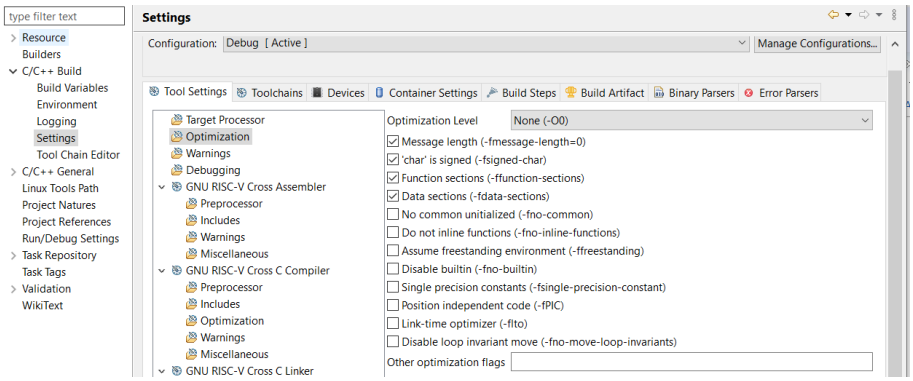
The number of each test run is determined by two parameters, namely LOCAL_SCALE_FACTOR and CPU_MHZ; those two parameters are multiplied and the result is the number of time the benchmark has to be run. Those numbers are not random; in fact, depending on the test, those numbers allow the run time to be approximately 4 seconds for the benchmark microcontroller which is an ARM Cortex-M4. These two parameters are present in each test.c file. The results obtained with the benchmark microcontroller have to be considered with no compiler optimization flag.

To have a clearer grasp on the zero-riscy code size/performance trade-off and GCC compiler preformance each test present in the Embench's collection was performed changing the compiler optimization flags; in particular

- -O0 flag. This is the default flag which optimize for compilation time
- -O3 flag. This flag let the compiler to do an heavy optimization for both code size and performance
- -Os flag. This flag let the compiler to do an heavy optimization for code size only

The optimization flag can be set using the Eclipse IDE as it's represented in figure 5.1.

Figure 5.1: Eclipse optimization flag setting




The following table gives a visual representation of the effectivness of each optimization flag.

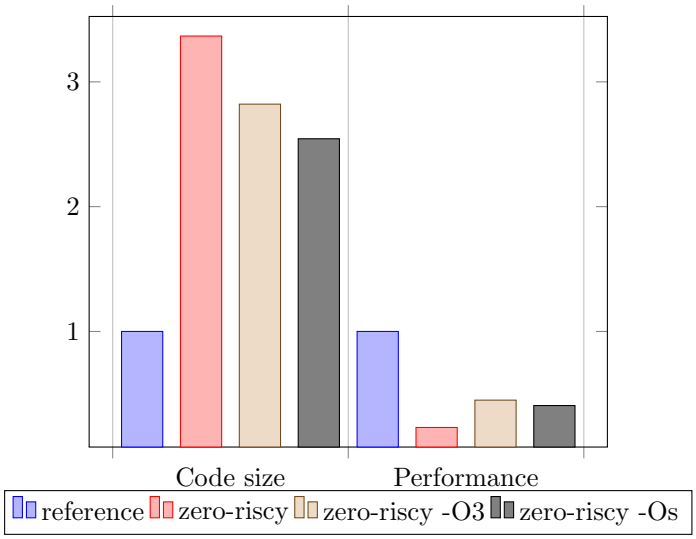
| option | optimization level | execution time | code size | memory usage | compile time |
|---|---|---|---|---|---|
| -O0 | opt for compilation time | + | + | - | - |
| -O1 | opt for code size and execution time | - | - | + | + |
| -O2 | opt for code size and execution time | - - | | + | ++ |
| -O3 | opt for code size and execution time | - - - | | + | +++ |
| -Os | opt for code size | | - - | | ++ |

Table 5.2: GCC optimization flag

### 5.0.2 Results analysis

As it is possible to see from the normalized histogram below, relative to the benchmark microcontroller the zero-riscy microcontroller is both slower and has a larger code size compared to the same test.

In particular the blue bar represent the normalized result, for both code size and performance, for the reference microcontroller. The red, yellow and grey bar represent the result, relative to the reference microcontroller, for no compiler optimization (-O0 flag), heavy optimization for code size and performance (-O3 flag) and heavy optimization for code size (-Os flag).



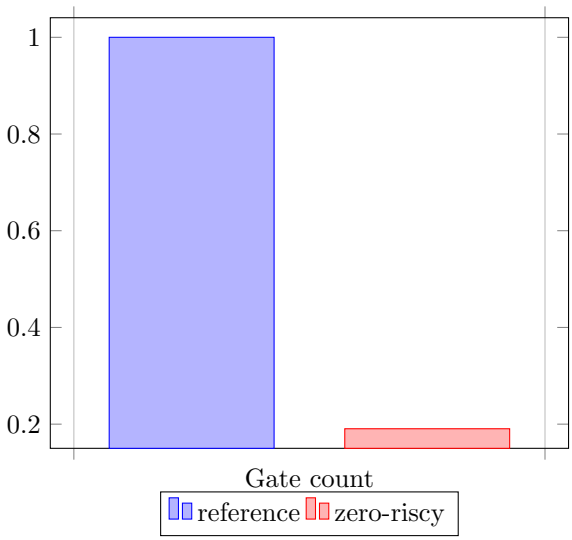

The exact number of code size and performance relative to the benchmark microcontroller are in the table 4.2.

| compiler flag | code size | performance |
|---|---|---|
| -O0 | 336% | 23% |
| -O3 | 282% | 45% |
| -Os | 254% | 41% |

Table 5.3: Results relative to reference

These results, however, do not take into account the size of the microcontroller itself; generally speaking a bigger microcontroller in terms of gate count has better performance. The zero-riscy implementation has appriximately 20kGe [17] while the Cortex M4 has 105kGe [18], thus justifying the worse performance and code size comparison relative to the benchmark microcontroller.

The bar graph below gives a visual representation of the difference in gate count between the reference microcontroller, the Cortex M4, and zero-riscy.




Better results in terms of code size and performance relative to the reference microcontroller could

have been obtained by using a commercial toolchain like IAR Embedded Workbench [19], however this analysis is outside of the scope of this master thesis's work.

### 5.0.3 Simulation

To check the correctness of each test two support signals (running and outcome, explained in chapter 4) were used. The main.c file, in fact, has a start and stop trigger function

```
...
//start_trigger ();
test_check = 0x00000001;
...

...
//stop_trigger ();
correct = verify_benchmark (result);
if(correct) {
        test_check = 0x00000002;
} else {
        test_check = 0x00000000;
}
...
```

Those functions have been substituted with an actuation on running and outcome signals, in particular

- running represent the test's status, 0 for finished and 1 for running
- outcome represent the test's outcome, 0 for fail and 1 for pass

Thanks to the testbench organization (more on it in chapter 4) looking at the log file was the only step needed. A sample of a log file is the following

```
...
test : crc32
test time start in ns is                148500
test time end in ns is                  15853599500
test time in ns is                      15853451000
test time in ms is                      15853
test time in cc is                      15853451
Test passed
Simulation complete via $finish(1) at time 15853699500 NS + 0
./tcs.sv:23 $finish ;
xcelium> assertion -summary -final ;
Summary report deferred until the end of simulation.
xcelium> exit ;
```

# Chapter 6

# FPGA porting process

To validate the whole microcontroller design and the toolchain setup an FPGA porting process is necessary. The overall process can be divided into two parts, namely FPGA board choice and FPGA porting process.

### 6.0.1 Board choice

The FPGA board choice is dependent on the HDL used to describe the RISC-V microcontroller; since, in fact, the HDL used is SystemVerilog the pool of software that could have been used shrunk. For instance looking at Xilinx software offer for FPGA development it's possible to see that

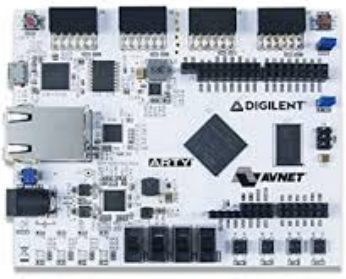
- ISE, support for VHDL/Verilog
- Vivado, support for VHDL/Verilog/SystemVerilog

Using Vivado as FPGA development software places a constraint to the number of boards supported in Vivado itself; in fact only the newer boards (7-series) are natively supported.

Due to the previous constraints the FPGA board of choice is the Arty A7-100T board. Some strengths for this particular board are

- Vivado support
- Many input/output pins available to the end user (more than 50)
- Extensive number of flip-flops (126800 flip-flops)
- 607KB of block RAM available
- USB-UART bridge

Figure 6.1: Arty A7 100T board



### 6.0.2 Porting process

The FPGA porting process has been straight forward thanks to the ease of use of the Vivado software. The porting process can be divided into two distinct sub-processes, namely

- clock instantiation
- xdc (constraint) file development

#### Clock instantiation

The Arty A7 board present a 100 MHz clock on pin E3. In order to provide a lower frequency clock to the microcontroller a clock manager has to be instantiated. Vivado ease this process through the clock wizard, a special menu that produces a Verilog file containing clock tiles capable of delivering an output clock compliant with the specifications given during creation. This module's top entity can be instantiated easily inside the microcontroller top entity. In particular the following is the portmap of the obtained clock manager

```
module clk_wiz_0
 (
  // Clock out ports
  output          clk_out1,
  // Status and control signals
  input           reset,
  output          locked,
 // Clock in ports
  input           clk_in1
 );
```

#### XDC file development

The constraint file developed for the FPGA implementation is available in the appendix chapter, in particular commands used are

- set_property -dict {PACKAGE_PIN pin_num IOSTANDARD LVCMOS33} [get_ports {port_name}]. This command routes an input/output port on the top entity of the microcontroller to a user available FPGA pin.
- create_clock -add -name clock_name -period[ns] period -waveform {start[ns] rising_edge[ns]} [get_ports {clock_port}]. This command informs Vivado that a particular port on the microcontroller top entity has a clock assigned.
- set_property PULLDOWN/PULLUP true [get_ports {port_name}]. This command assign a PULLUP/PULLDOWN to a particular port on the top entity of the microcontroller.

### 6.0.3 FPGA resource utilization

The synthesis and implementation process produced the following resource utilization results

| resource | utilization | available | utilization % |
|---|---|---|---|
| LUT | 4090 | 63400 | 6.45% |
| FF | 2747 | 126800 | 2.17% |
| BRAM | 8 | 135 | 5.93% |
| IO | 44 | 210 | 20.95% |
| BUFG | 3 | 32 | 9.38% |
| MMCM | 1 | 6 | 16.67% |

Table 6.1: FPGA resource utilization

Taking a closer look at the cell usage report it's possible to give a rough estimate on the number of ASIC gates used by the implementation. We have

| cell | count |
|---|---|
| LUT1 | 13 |
| LUT2 | 240 |
| LUT3 | 500 |
| LUT4 | 673 |
| LUT5 | 942 |
| LUT6 | 2352 |

Table 6.2: FPGA implementation cell count

A rough estimate on the equivalency between cell usage and gate count would be 3-8 gates per LUT [20]. This estimate would give

| estimate | gate count |
|---|---|
| 3x | 12.270kGe |
| 8x | 32.720kGe |

Table 6.3: gate count equivalency

This rough estimate is compliant with the informations gathered from the zero-riscy manual [17].

# Chapter 7

# Peripherals integration

The main topic for this master thesis is to integrate general purpose peripherals alongside the RISC-V microcontroller; in particular the required peripherals the be integrated are

- UART
- SPI
- I2C

SystemVerilog models for these peripherals were already available from Maxim Integrated's IPs bank [3]; for this reason the RTL development phase for each peripheral was skipped, leaving the integration activity to

- driver development
- testing phase

The following is a block diagram representing the final result after the peripherals integration step

### 7.0.1 UART

The UART (universal asynchronous receiver-transmitter) is a general purpose peripheral capable of serial communication at programmable speed rates. Usually this peripheral is used for communication and debugging purposes between the integrated circuit and a computer available to the end user. The fact that the UART peripheral lends itself to multiple uses eased the choice for first peripheral to be integrated; in fact this peripheral was used extensively for its debugging capability.

#### Portmap and register organization

In order to be able to integrate successfully any kind of peripheral it's necessary to understand two key points, namely

- peripheral's portmap
- peripheral's register organization

The two following tables give an overview of both the portmap and register organization of the UART peripheral.

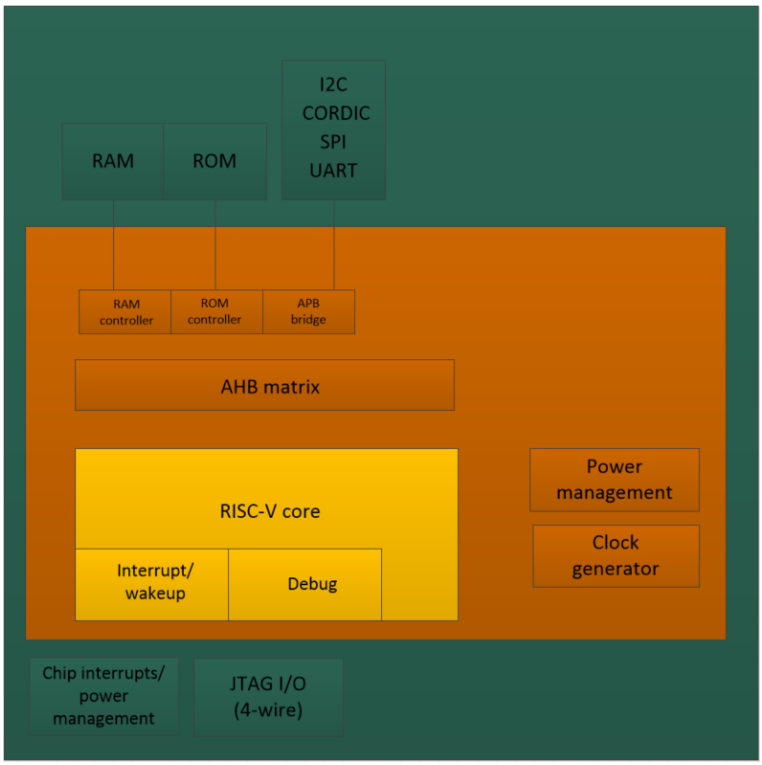
#### Baudrate calculation

The expression that describes the baud rate for the UART peripheral is

$$UART\_CKDIV = SEL\_CLK / Baudrate$$

The input $SEL\_CLK$ is a clock chosen between $CLK$, $CLK\_UART1$, $CLK\_UART2$, $CLK\_UART3$ through bits 16 and 17 of the control register $UART\_CTRL$. A further restriction is that the baudrate has to be at least four times slower than the system clock (input $CLK$).

Figure 7.1: RISC-V system block diagram




| port name | lenght | inuput/output | description |
|---|---|---|---|
| CLK | 1 bit | input | input clock |
| CLK_UART1 | 1 bit | input | secondary clock 1 for baud rate generator |
| CLK_UART2 | 1 bit | input | secondary clock 2 for baud rate generator |
| CLK_UART3 | 1 bit | input | secondary clock 3 for baud rate generator |
| RSTN | 1 bit | input | asynchronous reset |
| SCANMODE | 1 bit | input | SCAN mode input |
| IRQ_Q | 1 bit | output | interrupt request |
| WAKE_A_O | 1 bit | output | asynchronous wake up |
| RX_DMA_REQ_Q | 1 bit | output | inform system DMA that data in RX FIFO is available |
| TX_DMA_REQ_Q | 1 bit | output | inform system DMA that data in TX FIFO is available |
| PSEL | 1 bit | input | APB IP select |
| PENABLE | 1 bit | input | APB IP enable |
| PADDR | 10 bits | input | APB address bus |
| PWDATA | 32 bits | input | APB write data |
| PRDATA | 32 bits | output | APB read data |
| PREADY | 1 bit | output | APB ready |
| PSLVERR | 1 bit | output | APB slave error |
| PWRITE | 1 bit | input | APB write enable |
| RX_I | 1 bit | input | data reception input |
| TX_O | 1 bit | output | data transmission output |
| CTS_I | 1 bit | input | clear to send |
| RTS_O | 1 bit | output | ready to send |

Table 7.1: UART portmap

| register name | address | access | description |
|---|---|---|---|
| UART_CTRL | 0x00 | RW | control register |
| UART_STATUS | 0x04 | R | status register |
| UART_INTEN | 0x08 | RW | interrupt enable control register |
| UART_INTFL | 0x0c | RW | interrupt status flag register |
| UART_CKDIV | 0x10 | RW | clock divider register |
| UART_OSR | 0x14 | RW | over sampling rate register |
| UART_TXFIFO | 0x18 | R | TX FIFO output register |
| UART_PNR | 0x1C | RW | Pin register |
| UART_DATA | 0x20 | RW | FIFO rad/wirte register |
| Reserverd register 0 | 0x24 | R | |
| Reserverd register 1 | 0x28 | R | |
| Reserverd register 2 | 0x2C | R | |
| UART_DMA | 0x30 | RW | DMA configuration register |
| UART_WKEN | 0x34 | RW | wake up enable control register |
| UART_WKFL | 0x38 | RW | wake up status flag register |

Table 7.2: UART register organization

### Driver development

To configure the peripheral it's necessary to be able to read and write the registers present in table 6.2. Being the UART connected to the APB bus the driver development phase can be looked at as a read/write in a specific memory location owned by the APB bus. In C this action is performed by the following code

```
#define PULPINO_BASE_ADDR                0x0
#define SOC_PERIPHERALS_BASE_ADDR        (PULPINO_BASE_ADDR + 0x40000000)
#define UART_BASE_ADDR                   (SOC_PERIPHERALS_BASE_ADDR + 0x4000)

#define REG(x) (*((volatile unsigned int*)(x)))

#define UART_CTRL                        (UART_BASE_ADDR + 0x0000)
#define CTRL_UART                        REG(UART_CTRL)
```

Thanks to these defines it's possible to read/write to the UART CTRL register by reading and writing to CTRL_UART. The previous defines are repeated for each register in the table 6.2.

For a full overview of each function developed during this phase refer to the appendix chapter.

### Testing phase

The testing phase deals with using the drivers developed in the previous phase in order to validate the functionalities of the peripheral. According to the relative datasheet a typical software sequence to establish the UART serial communication is

- configure the baud rate
- configure the oversampling rate
- select the appropriate clock source
- configure the appropriate fifo rx threshold
- enable the internal UART baud clock
- wait for baud clock to be ready
- start the UART communication

In order to test both the rx and tx functionalities of the peripheral two UARTs have been instantiated, connecting the first UART RX input to the second UART TX output and vice versa. With this setup is possible to test extensively both the rx and tx peripheral's capabilities by firmware.

The firmware testing phase can be divided into two distinct phases

- initial peripherals configuration
- initial UART transmission

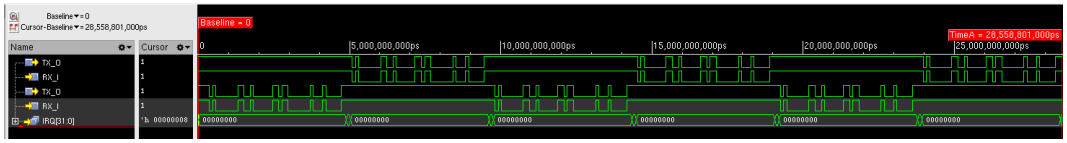
In order to test both rx and tx capabilities the initial peripheral configuration, similar for both peripherals, is composed of

- baud rate configuration
- over sampling rate configuration
- parity, character size and stop bit(s) configuration
- baud clock enable and check
- rx fifo threshold setting (set to 4)
- interrupt on rx fifo threshold reached enabled

- interrupt service routine configured to transmit the same character received back to the other UART

The initial UART transmission is made up of four characters (1-4). This initial transmission triggers the interrupt of the second UART, which is going to transmit back those characters to the first UART, creating a loop of transmissions of characters as seen in figure 6.1.

Figure 7.2: UART testing phase




To definitely confirm that the peripheral has been configured correctly a final testing phase on FPGA was needed.

On the Arty A7 board it's possible to connect directly the RX and TX UART's signals to two FPGA pins routed into the USB-UART bridge which, in turn, is connected to the micro-usb connector allowing to convert USB packets to serial data. After installing FTDI drivers and a serial terminal (such as Tera Term [21]) it's possible to use the host PC to send/receive serial data to/from the FPGA board.

For this particular testing phase the C code developed would

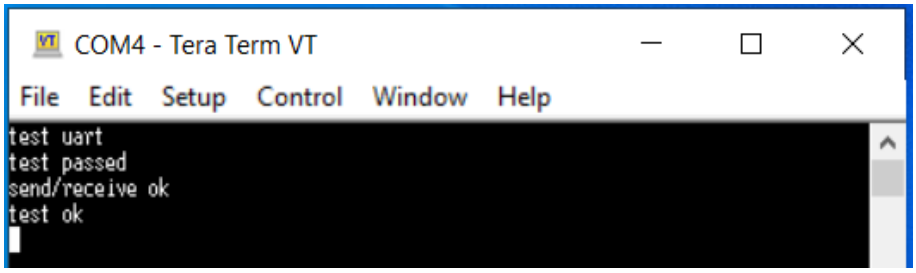
- set the correct baudrate (tested for 9600 and 19200)
- set the over sampling rate
- configure the parity, character size and stop bit(s)
- set the rx fifo threshold to 1
- inside the while(1) perform the uart_scanf() function

The uart_scanf() function enables the interrupt generated on rx fifo threshold reached and wait for the interrupt service routine to complete. The interrupt service routine checks each character coming to the UART, if the character received is

- CR (carriage return) or LF (line feed) ends the interrupt service routine
- any other character is appended to a char array called instruction

As soon as the interrupt service routine completes the uart_scanf() function sends through the uart_printf() function the instruction received to the serial terminal as seen on figure 6.2.

Figure 7.3: UART FPGA testing phase




The combination of these two tests (simulation and FPGA) are sufficient to determine the correctness of both the UART IP connection to the RISCV microcontroller and the driver development.

### 7.0.2 SPI

The SPI (serial peripheral interface) is a general purpose peripheral capable of serial communication between a *master* and one or multiple *slave* devices. The *master* control the SPI bus composed of the following signals

| port name | lenght | description |
|---|---|---|
| slck | 1 bit | SPI clock |
| ss | 1 bit | SPI selection signal |
| mosi | 1 bit | SPI master out slave in |
| miso | 1 bit | SPI slave in master out |

Table 7.3: SPI bus signals

#### Portmap and register organization

As highlighted before it is of paramout importance to to understand the peripheral's portmap and the peripheral's register organization to integrate successfully any peripheral. The two following tables give an overview of

- SPI's portmap
- SPI's register organization

| port name | lenght | inuput/output | description |
|---|---|---|---|
| CLK | 1 bit | input | input clock |
| RSTN | 1 bit | input | asynchronous reset |
| IRQ_Q | 1 bit | output | interrupt request |
| WAKE_A_O | 1 bit | output | wake up request |
| RX_DMA_REQ_Q | 1 bit | output | inform system DMA that data in RX FIFO is available |
| TX_DMA_REQ_Q | 1 bit | output | inform system DMA that data in TX FIFO is available |
| PSEL | 1 bit | input | APB IP select |
| PENABLE | 1 bit | input | APB IP enable |
| PADDR | 10 bits | input | APB address bus |
| PWDATA | 32 bits | input | APB write data |
| PRDATA | 32 bits | output | APB read data |
| PREADY | 1 bit | output | APB ready |
| PSLVERR | 1 bit | output | APB slave error |
| PWRITE | 1 bit | input | APB write enable |
| SCLK | 1 bit | output | serial clock to slaves (used in master mode) |
| SSEL_OUT | 8 bits | output | slave select to slaves (used in master mode) |
| SDO | 4 bits | output | serial data out (used in master mode) |
| SDO_OE | 4 bits | output | serial data output enable (used in master mode) |
| SSEL_IN | 1 bit | input | slave select from master (used in slave mode) |
| SDI | 4 bits | input | serial data in (used in slave mode) |
| SCLK_IN | 1 bit | input | serial clock from master (used in slave mode) |

Table 7.4: SPI portmap

| register name | address | access | description |
|---|---|---|---|
| SPI_DATA | 0x00 | RW | data register |
| SPI_CTRL1 | 0x04 | RW | control register |
| SPI_CTRL2 | 0x08 | RW | control register |
| SPI_CTRL3 | 0x0c | RW | control register |
| SPI_CTRL4 | 0x10 | RW | control register |
| SPI_BRG_CTRL | 0x14 | RW | baud rate generator control register |
| reserved | 0x18 | R | reserved register |
| SPI_DMA | 0x1C | RW | DMA register |
| SPI_IRQ | 0x20 | RW | interrupt status register |
| SPI_IRQE register | 0x24 | RW | interrupt enable register |
| SPI_WAKE register | 0x28 | RW | wakeup status register |
| SPI_WAKEE register | 0x2C | RW | wakeup enable register |
| SPI_STAT | 0x30 | R | status register |
| reserved | 0x34 | R | reserved register |

Table 7.5: SPI register organization

### Baudrate calculation

Similarly to the UART peripheral it is possible to scale the system clock (input $CLK$) to have a slower serial clock for each SPI transmission. To scale the system clock bits 16 to 19 of the register $BRG\_CTRL$ are used; the system clock is scaled by $2^{scale\_factor}$ with a maximum scale value of $2^8$ (any bigger scale factor is illegal and interpreted as $2^8$).

### Driver development

Similarly to the previous peripheral it's necessary to be able to read and write the registers present in table 6.5. Being the SPI connected to the APB bus the driver development phase can be looked at as a read/write in a specific memory location owned by the APB bus. In C this action is performed by the following code

```
#define PULPINO_BASE_ADDR                0x0
#define SOC_PERIPHERALS_BASE_ADDR        (PULPINO_BASE_ADDR + 0x40000000)
#define SPI_BASE_ADDR                    (SOC_PERIPHERALS_BASE_ADDR + 0x2000)

#define REG(x) (*((volatile unsigned int*)(x)))

#define SPI_DATA                         (SPI_BASE_ADDR + 0x0000)
#define DATA_SPI                         REG(SPI_DATA)
```

Thanks to these defines it's possible to read/write to the SPI DATA register by reading and writing to DATA_SPI. The previous defines are repeated for each register in the table 6.5.

For a full overview of each function developed during this phase refer to the appendix chapter.

### Testing phase

The testing phase deals with using the drivers developed in the previous phase in order to validate the functionalities of the peripheral. According to the relative datasheet a typical software sequence to establish the SPI serial communication is

- enable SPI peripheral
- enable tx and rx fifos
- select *master* or *slave* mode
- select clock phase and polarity
- set the number of bytes for each transmission

- start the SPI communication

In order to test both the rx and tx functionalities of the peripheral two SPIs have been instantiated with the following connections

- first SPI sclk_in with second SPI sclk
- first SPI ss_in with second SPI ss_out
- first SPI sdi with second SPI sdo

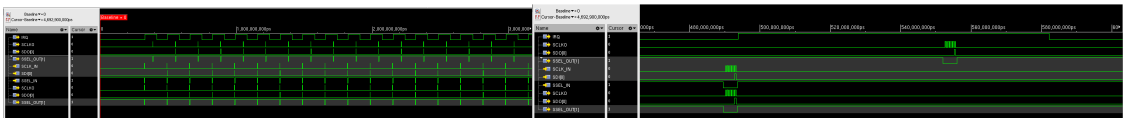
The configuration for both SPIs is composed of

- enable SPI peripheral
- enable tx and rx fifos
- select clock phase and polarity
- set the number of bytes for each transmission
- *master* mode selection for second SPI
- *slave* mode selection for first SPI
- interrupt fifo threshold set to one for the first SPI

This test is based on the fact that each SPI transmission from the second SPI triggers the interrupt service routine of the first SPI. The interrupt service routine performs

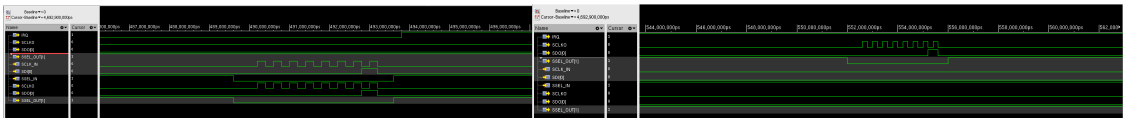
- reads the data received
- transmit back the data read

To start the test the SPI transmission is placed in a while(1) loop, thus making the test indefinitely long. After each SPI transmission from the second SPI peripheral the data to be transmitted is incremented.
Figure 6.3 and 6.4 are proof of the behavior described.



(a) SPI loop test (b) SPI loop test transaction

Figure 7.4: Overall SPI loop test



(a) second SPI write (b) first SPI write

Figure 7.5: SPI loop test - write focus

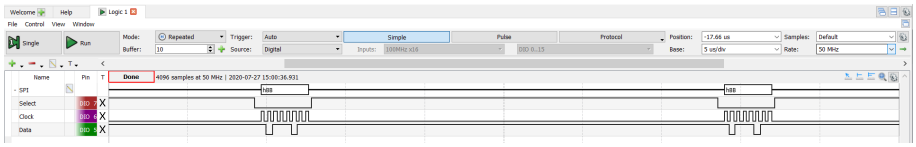
To definitely confirm that the peripheral has been configured correctly a final testing phase on FPGA was needed. The test performed on FPGA is directed to check the correctness of an SPI write transaction; a program consisting of an SPI write inside a while(1) loop have been chosen for this purpose.

To monitor the SPI traffic an oscilloscope is needed; the oscilloscope of choice is the *Analog Discovery 2* [22]. It has the feature of having a bus analyzer, capable of analyzing the most common communication protocols such as

- UART
- SPI
- I2C
- CAN
- 1WIRE

As already introduced, this test performs an SPI write inside a while(1) loop with a static data, equal to 0xbb, being transmitted. This behavior is displayed in Figure 6.5.

Figure 7.6: SPI FPGA testing phase



The combination of these two tests (simulation and FPGA) are sufficient to determine the correctness of both the SPI IP connection to the RISCV microcontroller and the driver development.

### 7.0.3 I2C

The I2C (inter-integrated circuit) is a general purpose peripheral capable of serial communication between one or multiple *master* and one or multiple *slave* devices. The *master* control the I2C bus composed of the following signals

| port name | lenght | description |
|---|---|---|
| sda | 1 bit | serial data |
| scl | 1 bit | serial clock |

Table 7.6: I2C bus signals

#### Portmap and register organization

As highlighted before it is of paramout importance to to understand the peripheral's portmap and the peripheral's register organization to integrate successfully any peripheral. The two following tables give an overview of

- I2C's portmap
- I2C's register organization

| port name | lenght | inuput/output | description |
|---|---|---|---|
| CLK | 1 bit | input | input clock |
| RSTN | 1 bit | input | asynchronous reset |
| IRQ_Q | 1 bit | output | interrupt request |
| WAKE_A_O | 1 bit | output | wake up request |
| RX_DMA_REQ_Q | 1 bit | output | inform system DMA that data in RX FIFO is available |
| TX_DMA_REQ_Q | 1 bit | output | inform system DMA that data in TX FIFO is available |
| PSEL | 1 bit | input | APB IP select |
| PENABLE | 1 bit | input | APB IP enable |
| PADDR | 10 bits | input | APB address bus |
| PWDATA | 32 bits | input | APB write data |
| PRDATA | 32 bits | output | APB read data |
| PREADY | 1 bit | output | APB ready |
| PSLVERR | 1 bit | output | APB slave error |
| PWRITE | 1 bit | input | APB write enable |
| SCL_PAD_OE | 1 bit | output | scl pad output enable |
| SDA_PAD_OE | 1 bit | output | sda pad output enable |
| SCL_PAD_DO | 1 bit | output | scl pad data out |
| SDA_PAD_DO | 1 bit | output | sda pad data out |
| SDA_PAD_IN | 1 bit | input | sda pad in |
| SCL_PAD_IN | 1 bit | input | scl pad in |

Table 7.7: I2C portmap

### Baudrate calculation

Similarly to the other peripherals it is possible to set the $SCL$ frequency to a divided value of the system clock. In this particular case it is necessary to set both the low pulse duration and high pulse duration of the $SCL$ clock relative to the system clock. The expression (similarly for the high period)

$$SCL_{low} = SYS\_CLK * (CLK\_LOW\_REG + 1)$$

determines the duration of the low pulse duration of the $SCL$ clock. The minimum value for $CLK\_LOW\_REG$ and $CLK\_HIGH\_REG$ is one (set through $I2CCKL$ and $I2CCKH$ registers), thus giving a maximum $SCL$ frequency of $SYS\_CLK/4$.

### Driver development

Similarly to the previous peripheral it's necessary to be able to read and write the registers present in table 6.8. Being the I2C connected to the APB bus the driver development phase can be looked at as a read/write in a specific memory location owned by the APB bus. In C this action is performed by the following code

```
#define PULPINO_BASE_ADDR                0x0
#define SOC_PERIPHERALS_BASE_ADDR        (PULPINO_BASE_ADDR + 0x40000000)
#define I2C_BASE_ADDR                    (SOC_PERIPHERALS_BASE_ADDR + 0x1000)

#define REG(x) (*((volatile unsigned int*)(x)))

#define I2C_CTRL                         (I2C_BASE_ADDR + 0x0000)
#define CTRL_I2C                         REG(I2C_CTRL)
```

Thanks to these defines it's possible to read/write to the I2C CTRL register by reading and writing to CTRL_I2C. The previous defines are repeated for each register in the table 6.8.

For a full overview of each function developed during this phase refer to the appendix chapter.

| register name | address | access | description |
|---|---|---|---|
| I2C_CTRL | 0x00 | RW | control register |
| I2C_STATUS | 0x04 | RW | status register |
| I2C_INT0 | 0x08 | RW | interrupt 0 register |
| I2C_INTE0 | 0x0c | RW | interrupt enable 0 register |
| I2C_INT1 | 0x10 | RW | interrupt 1 register |
| I2C_INTE1 | 0x14 | RW | interrupt enable 1 register |
| I2C_FIFO_CFG | 0x18 | R | fifo configuration register |
| I2C_RX_CFG_REG | 0x1C | RW | rx fifo configuration register |
| I2C_RX_REG | 0x20 | RW | rx register register |
| I2C_TX_CFG_REG | 0x24 | RW | tx configuration register |
| I2C_TX_REG | 0x28 | RW | tx register |
| I2C_DATA | 0x2C | RW | data register |
| I2C_MASTER | 0x30 | RW | master register |
| I2C_CLK_LOW | 0x34 | RW | clock low register |
| I2C_CLK_HIGH | 0x38 | RW | clock high register |
| I2C_HS_MODE | 0x3C | RW | high speed mode register |
| I2C_TIMEOUT | 0x40 | RW | timeout register |
| reserved | 0x44 | RW | reserved register |
| I2C_DMA | 0x48 | RW | DMA register |
| I2C_SLAVE | 0x4C | RW | slave register |

Table 7.8: I2C register organization

**Testing phase**

The testing phase deals with using the drivers developed in the previous phase in order to validate the functionalities of the peripheral. According to the relative datasheet a typical software sequence to establish the I2C serial communication is

- enable I2C peripheral
- select *master* or *slave* mode
- set the I2C clock frequency through clock high/low registers
- set the number of bytes for each transmission (in case of read)
- fill the tx fifo with the address of the slave device (in case of master mode)
- start the I2C communication

In order to test both the rx and tx functionalities of the peripheral two I2Cs have been instantiated with the following connections

- first and second I2C scl line with the global scl bus line
- first and second I2C sda line with the global sda bus line

The configuration for both I2Cs is composed of

- enable I2C peripheral
- select *master* or *slave* mode (one *master* and one *slave*)
- set the I2C clock frequency through clock high/low registers for the *master* peripheral
- set the number of bytes for each transmission for the *master* peripheral
- start the I2C communication

The test performed are

Figure 7.7: I2C tx test

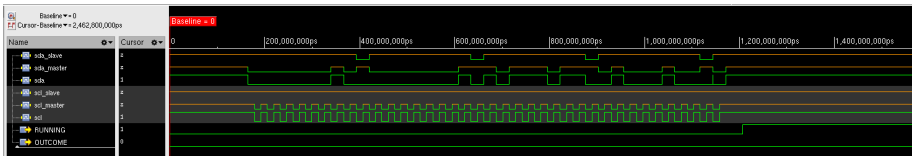
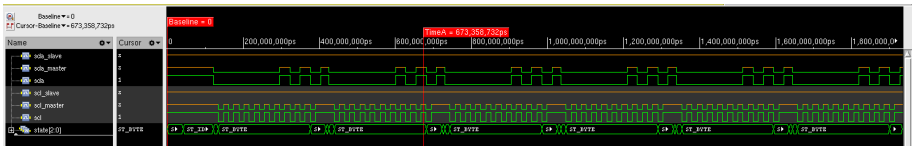



Figure 7.8: I2c only master




- tx three bytes from master to slave
- tx communication with only the master peripheral on the bus

As it's possible to observe in the first case the *slave* acknowledges the *master* transmission request, while in the second case the *master* is forced to end the transmission due to the fact that the acknowledge is missing.

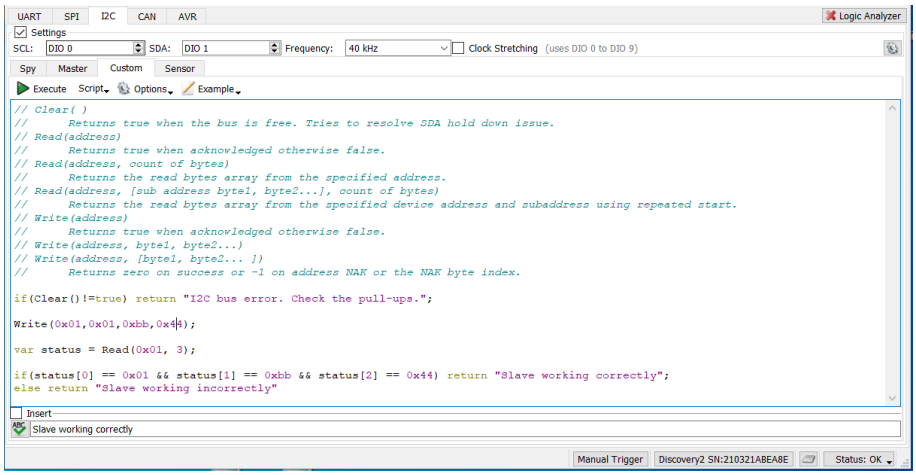
The second test (only master) has been replicated on FPGA to check the correctness of the I2C peripheral in master mode. The result is the same as in the RTL simulation

Figure 7.9: I2C FPGA master only


To definitely confirm that the peripheral has been configured correctly a final testing phase on FPGA was needed. The test performed on FPGA is directed to check the correctness of an I2C read transaction. The *Analog Discovery 2* [22] oscilloscope is used to generate I2C traffic such as a *master* I2C peripheral. In this particular test the initial I2C transaction is a write; this write transaction put in the I2C peripheral's (synthesized on FPGA) rx fifo three bytes which are read during the following read transaction started by the oscilloscope. The data received is then checked and a message is printed. Proof for this test is in Figure 6.9.

The combination of these tests (simulation for both *master* and *slave* mode, FPGA tests for both *master* and *slave* mode) determine the correctness of both the I2C IP connection to the RISCV microcontroller and the driver development.

Figure 7.10: I2C FPGA testing phase

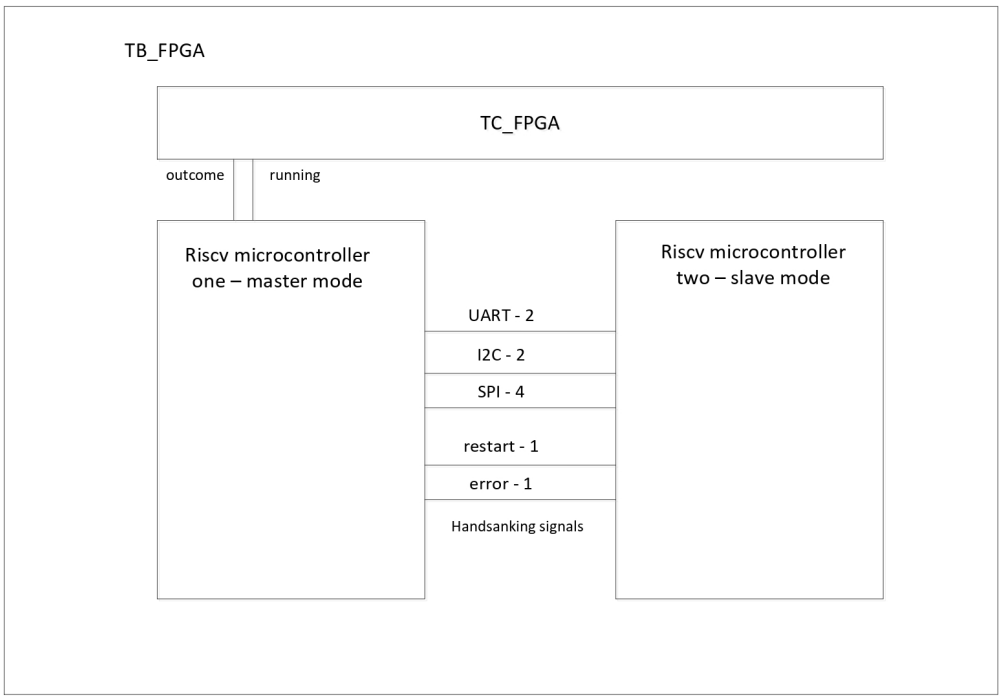


# Chapter 8

# Peripherals stress test

To test extensively the behavior of the peripherals integrated (UART, SPI and I2C) presented in the previous chapther a more thorough test is needed. For this stress test at the top level two microcontrollers are instantiated; connecting together the general purpose peripherals is possible to test both master and slave modes. Two GPIOs are also connected to two IRQ inputs for handshaking purposes. Figure 8.1 gives a visual representation of the testbench organization

Figure 8.1: Testbench organization




To test extensively the general purpose peripherals four tests have been developed, namely

- I2C master write
- I2C master read
- UART communication
- SPI communication

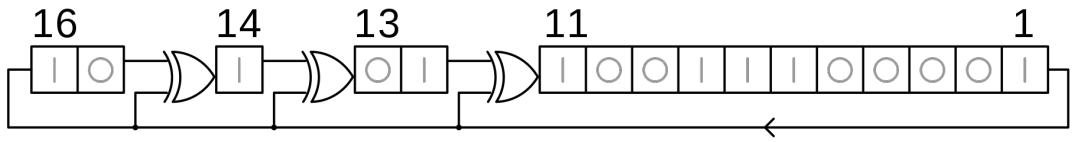
The following subsections explain in further details each test purpose and operation.

### 8.0.1 LFSR

To effectively test a communication of data between any peripherals a random data generator is needed in order to guarantee that the communication is not going to fail for a particular data being transmitted and received; a LFSR (linear feedback shift register) has been used for this purpose. An LFSR is a shift register whose input bit is a linear function of its previous state [23]. Figure 8.2 gives a visual representation of an LFSR. An LFSR is used as

- pseudo-random number generator
- pseudo-noise sequencer

Figure 8.2: LFSR




The initial value of the LFSR is called seed, while the bits that influence the input (bits 11, 13, 14 and 16 in figure 8.2) are called taps. Since the register operation is deterministic the values produced by the LFSR are also deterministic. Likewise the number of possible state is finite, thus the LFSR must re-enter in a repeating cycle. By a carefull choice of initial seed and taps the period of the repeating cycle can assume the maximum value of $2^n - 1$, where n is the number of bits present in the LFSR (16 in figure 8.2). Both hardware and software implementation exist; in this particular case a software implementation is needed to generate pseudo-random data to be transmitted via UART, SPI and I2C. A software implementation of an LFSR is as follows

```
HALF_WORD lfsr_16(HALF_WORD lfsr, HALF_WORD mask){
int lsb;

lsb = lfsr & 1;
lfsr >>= 1;

if (lsb == 1){
        lfsr ^= mask;
}

return lfsr;
}
```

With the values for the initial state of the lfsr 0xACE1 and the mask 0xB400 the LFSR is able to traverse each possible state, thus having a lenght of $2^{16} - 1$. This kind of LFSR is called maximum lenght LFSR. These exact values are used in each peripheral stress test in order to guarantee the maximum code coverage possible.

### 8.0.2 I2C master write

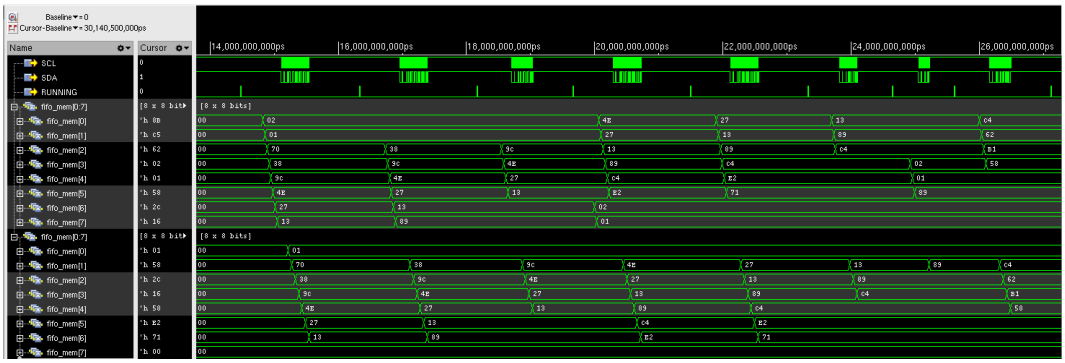
The I2C master write test's purpose is to test two conditions

- Correctness of the write operation of the I2C peripheral in master mode
- Correctness of the read operation of the I2C peripheral in slave mode

In synthesis the test can be divided in two parts, the master operation and the slave operation. The first microcontroller, in charge of the master operation, writes a random number of bytes (between two and seven, depending on the lfsr state) to the slave; the second microcontroller, in charge of the slave operation, reads those bytes and checks the correctness of the trasnmission against the internal lfsr state. A code snippet for both operations can be found in the appendix chapter.

Figure 8.3 represent a series of I2C transactions between the master and the slave; as it is possible to see the master microcontroller sends via I2C a random number of bytes to the slave peripheral. The slave then reads the bytes received, checks them, and, if a correct corrrespondance between data recieved and the internal state of the LFSR is found, sends a pulse to the master microcontroller through the RUNNING output pin.

Figure 8.3: I2C master write - RTL




A breakdown of the firmware of both microcontrollers is as follows

Master firmware divided into main.c, irq_0.c and irq_1.c

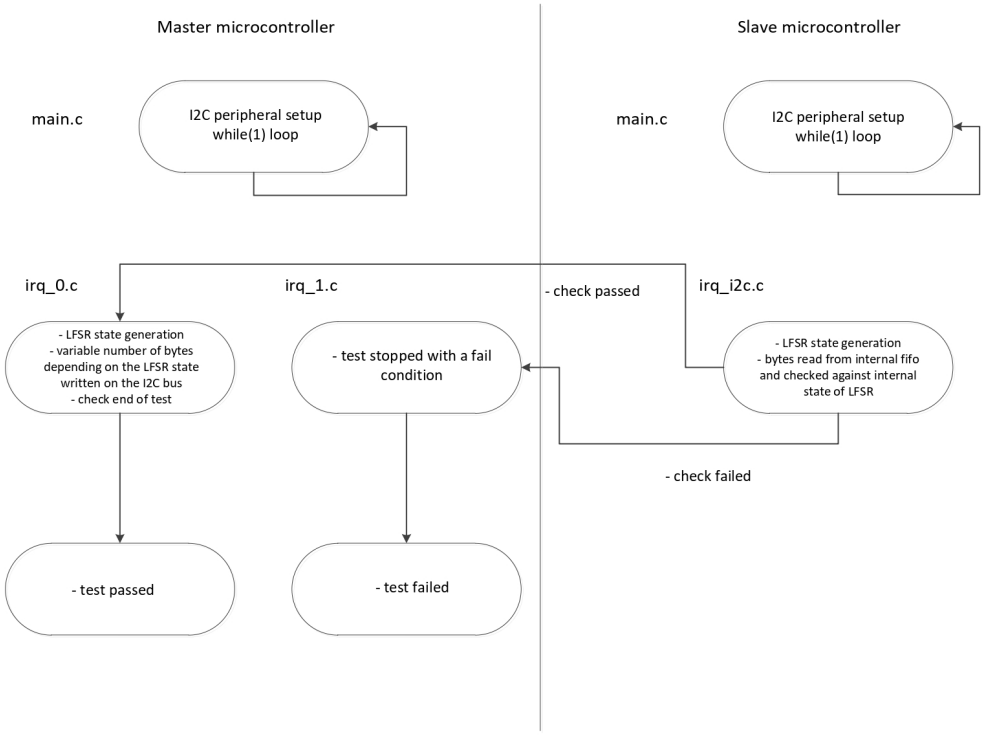
- In main.c the I2C peripheral is enabled and the master mode is selected
- In irq_0.c a new state of the LFSR is generated. Depending on the last three bits of the LFSR state a random number of bytes (between two and seven) is written on the I2C bus. irq_0.c is also responsible to check the end of test; if the LFSR state correspond to the initial one the test is stopped with a pass condition
- In irq_1.c the test is stopped with a fail condition

Slave firmware divided into main.c and irq_i2c.c

- In main.c the I2C peripheral is enabled and the slave mode is selected
- In irq_i2c.c a new LFSR state is generated to match the LFSR state present in the master microcontroller. The number of bytes present in the internal I2C fifo is retrieved; the corresponding number of bytes is read from the fifo and checked against the internal state of the LFSR. In case of correct check the slave microcontroller generated a pulse on an output pin connected to the irq_0 of the master microcontroller (thus restarting the write operation). In case of incorrect check the slave microcontroller generate a pulse on a second output pin connected to the irq_1 of the master microcontroller (thus stopping with an error the test)

Figure 8.4 represent the state diagram of the operations of both microcontroller.

Figure 8.4: I2C master write - state diagram

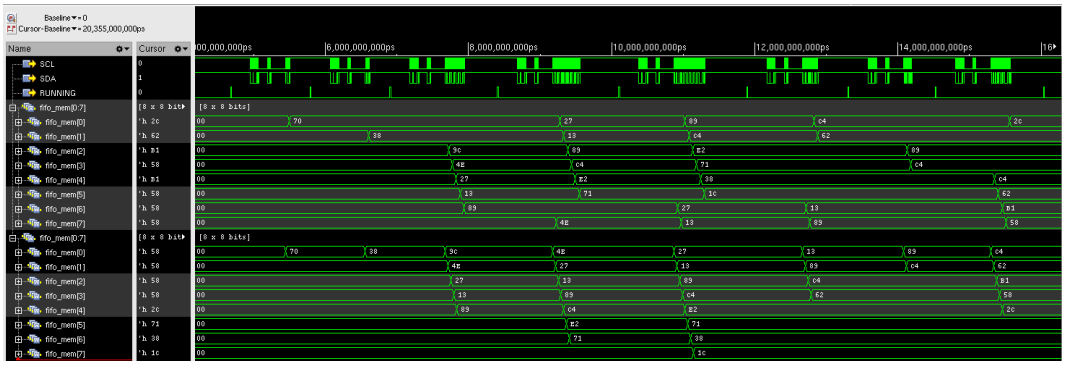



### 8.0.3 I2C master read

The I2C master read test's purpose is to test two conditions

- Correctness of the read operation of the I2C peripheral in master mode
- Correctness of the write operation of the I2C peripheral in slave mode

Figure 8.5: I2C master read - RTL




In synthesis the test can be divided in two parts, the master operation and the slave operation. The first microcontroller, in charge of the master operation, starts a read transaction. The second microcontroller, in charge of the slave operation, aknowledges the read request coming from the master and writes a random number of bytes (between one and eight, depending on the lfsr state). The master, at the end of the transaction, checks the bytes recieved against the internal lfsr state. A code snippet for both operations can be found in the appendix chapter.

Figure 8.4 represent a series of I2C transactions between the master and the slave; as it is possible

to see the master microcontroller sends a I2C read request to the slave microcontroller. The slave microcontroller then sends to the master a random number of bytes dependent on the internal state of the LFSR; the master then checks the data recieved and, in case of a successful check, starts a new read request to the slave.

A breakdown of the firmware of both microcontrollers is as follows

Master firmware divided into main.c , irq_i2c.c and irq_0.c

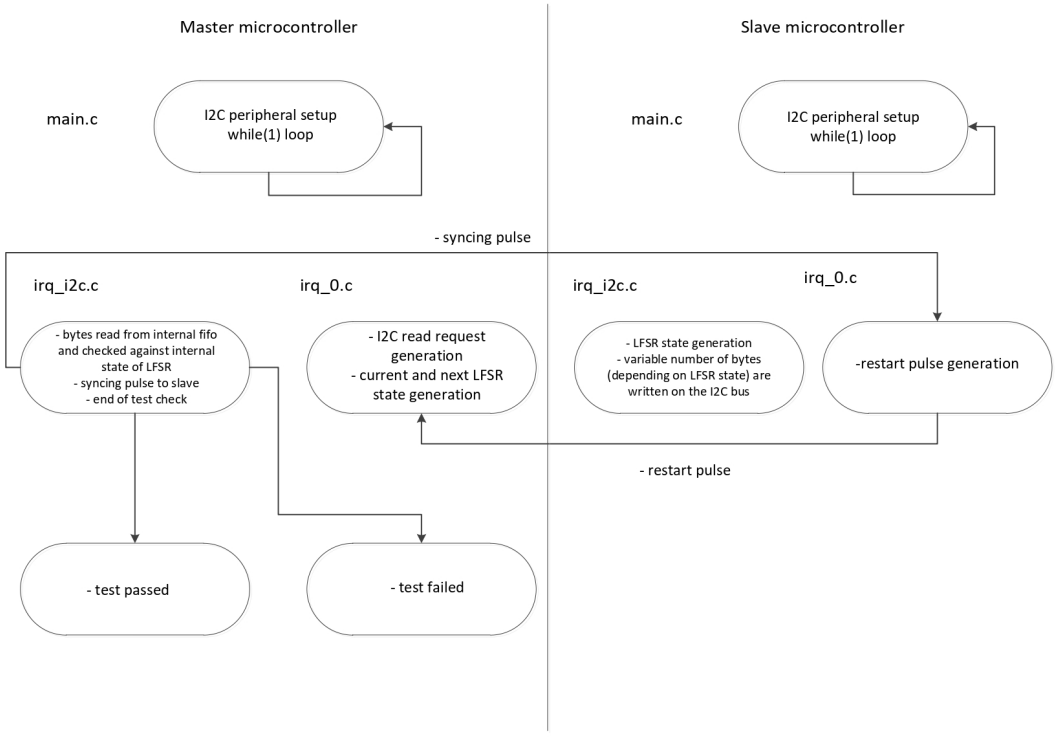
- In main.c the I2C peripherals is enabled and the master mode is selected
- In irq_i2c.c the data received is checked against the internal state of the LFSR. In case of unsuccesful check the test is stopped with a fail, otherwise the I2C peripheral is prepared for the next operation and a pulse is raised on an output pin connected to the slave microcontroller
- In irq_0.c an I2C read request is generated. The curent and next LFSR state are generated, to be used in irq_i2c.c code

Slave firmware divided into main.c , irq_i2c.c and irq_0.c

- In main.c the I2C peripherals is enabled and the slave mode is selected. A pulse is sent through an output pin connected to the master's irq_0 to start the test
- In irq_i2c.c the internal LFSR state is generated, to match the one present in the master microcontroller. Depending on the last three bits of the LFSR state a random number of bits (between one and eight) is sent to the I2C bus
- In irq_0.c a pulse is sent through an output pin connected to the master's irq_0 to start the test again

Figure 8.6 represent the state diagram of the operations of both microcontroller.

Figure 8.6: I2C master read - state diagram



### 8.0.4 UART communication

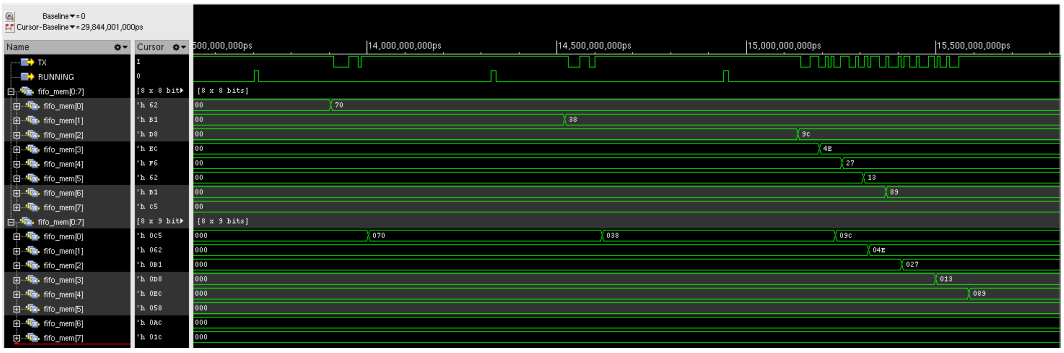
The UART communication test's purpose is to test the following

- Correctness of the TX operation of the UART peripheral
- Correctness of the RX operation of the UART peripheral

In synthesis the test can be divided in two parts, the TX operation and the RX operation. The first microcontroller, in charge of the TX operation, writes a random number of bytes (between one and eight, depending on the lfsr state) to the slave; the second microcontroller, in charge of the RX operation, reads those bytes and checks the correctness of the trasnmission against the internal lfsr state. A code snippet for both operations can be found in the appendix chapter.

Figure 8.7 represent a series of UART transactions between the master and the slave; as it is possible to see the master microcontroller sends a number of bytes trough the UART to the slave microcontroller. The slave microcontroller then checks the data recieved against the internal state of the LFSR; in case of successful check a pulse through the RUNNING output pin is sent back to the master microcontroller which starts a new UART communication.

Figure 8.7: UART communication - RTL




A breakdown of the firmware of both microcontrollers is as follows

Master firmware divided into main.c, irq_0.c and irq_1.c

- In main.c the UART peripheral is initialized with the right baud-rate, number of bits per trasnmission, parity and stop bit configuration
- In irq_0.c a new LFSR state is computed; depending on the last three bits of the LFSR state a random number of bytes (between one and eight) is sent to the slave microcontroller. This section of code also checks for the end of test condition; if the end condition is met the master microcontroller end the test with a pass condition. At the end of the UART transmission a pulse on an output pin is sent to the slave microcontroller
- In irq_1.c the master microcontroller ends the test with a fail result

Slave firmware divided into main.c and irq_0.c

- In main.c the UART peripheral is initialized with the right baud-rate, number of bits per trasnmission, parity and stop bit configuration. An initial pulse is sent out through an output pin connected to irq_0 routine of the master microcontroller to start the test
- In irq_0.c the new LFSR's state is computed; the data received is checked against the internal LFSR's state. If the check is unsuccesful the output connected the master's irq_1 routine is raised, otherwise a pulse is sent through the output pin connected to the master's irq_0 routine, thus restarting the UART communication

Figure 8.8: UART communication - state diagram

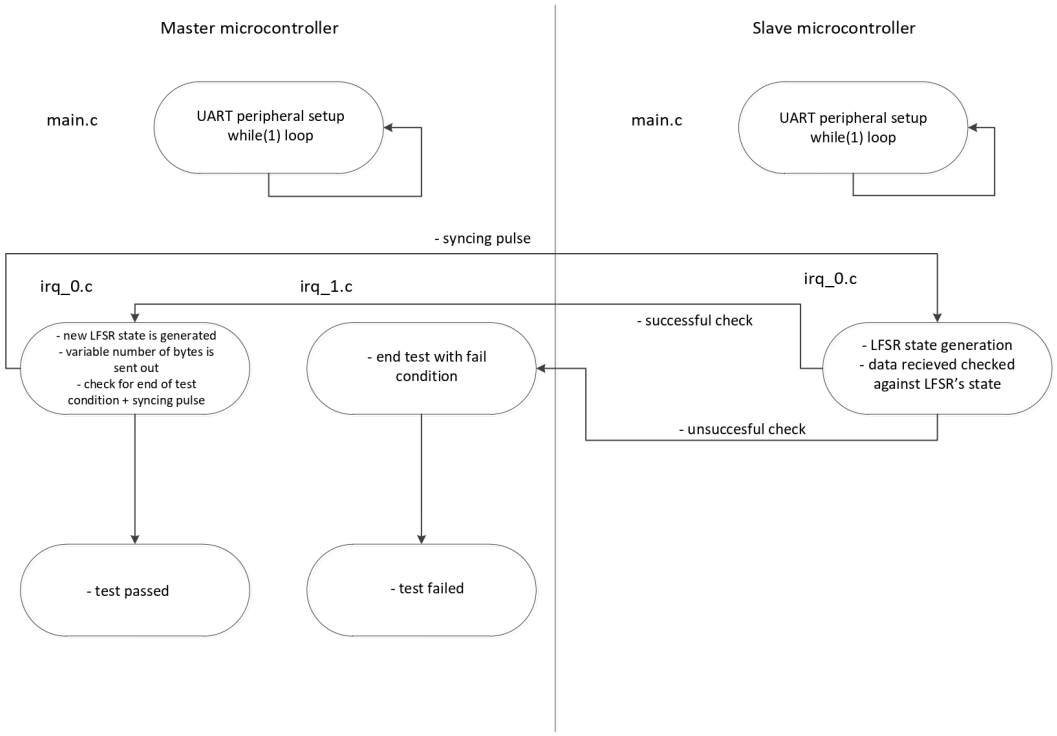



Figure 8.6 represent the state diagram of the operations of both microcontroller.

The test has been repeated for the following operation modes

- baud rate 115200, 8 bits for each data transmission, 1 stop bit
- baud rate 9600, 5 bits for each data transmission, 1 stop bit
- baud rate 38499, 6 bits for each data transmission, 1 stop bit

### 8.0.5 SPI communication

The SPI communication test's purpose is to test the following

- Correctness of the TX operation of the SPI peripheral in master mode
- Correctness of the RX operation of the SPI peripheral in slave mode

In synthesis the test can be divided in two parts, the TX operation and the RX operation. The first microcontroller, in charge of the TX operation, writes a random number of bytes to the slave; the second microcontroller, in charge of the RX operation, reads those bytes and checks the correctness of the trasnmission against the internal lfsr state. A code snippet for both operations can be found in the appendix chapter.

Figure 8.9 represent a series of SPI transactions between the master and the slave; as it is possible to see the master microcontroller sends a number of bytes trough the SPI to the slave microcontroller. The slave microcontroller then checks the data recieved against the internal state of the LFSR; in case of successful check a pulse through the RUNNING output pin is sent back to the master microcontroller which starts a new SPI communication.

A breakdown of the firmware of both microcontrollers is as follows

Master firmware divided into main.c, irq_0.c and irq_1.c

- In main.c the SPI peripheral is initialized
- In irq_0.c a new LFSR state is computed; depending on the LFSR's state a random number of bytes/half words or words (depending on the test) is sent to the slave microcontroller. This section of code also checks for the end of test condition; if the end condition is met the master microcontroller end the test with a pass condition
- In irq_1.c the master microcontroller ends the test with a fail result

Slave firmware divided into main.c and irq_spi.c

- In main.c the SPI peripheral is initialized
- In irq_spi.c a new LFSR state is computed; the data recieved thourgh SPI is checked against the internal state of the LFSR. In case of unsuccesful check an output pin connected to the master's irq_1 routine is raised, thus ending the test with a fail condition. This section of code alse generated a pulse through a pin connected to the master's irq_0 routine, thus restarting a new SPI transaction

Figure 8.9: SPI communication - RTL

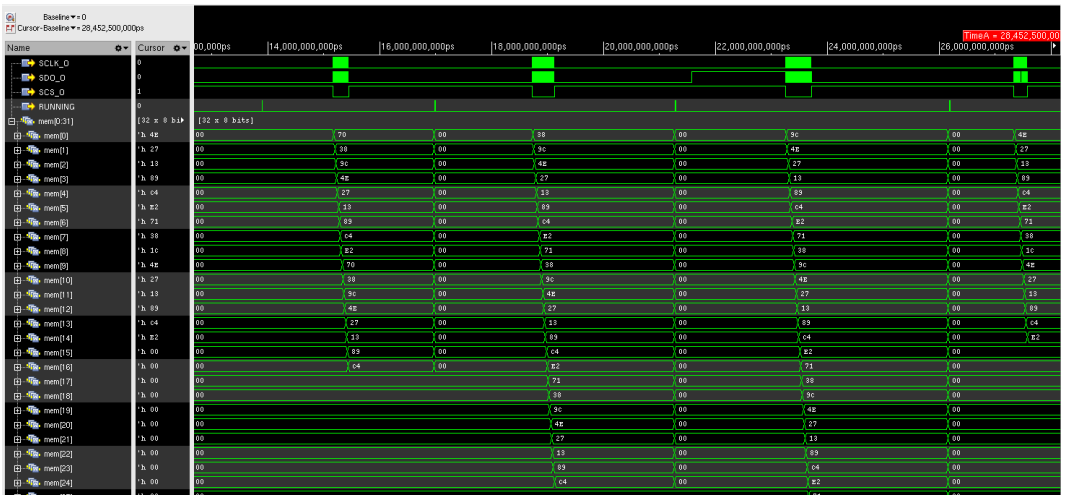
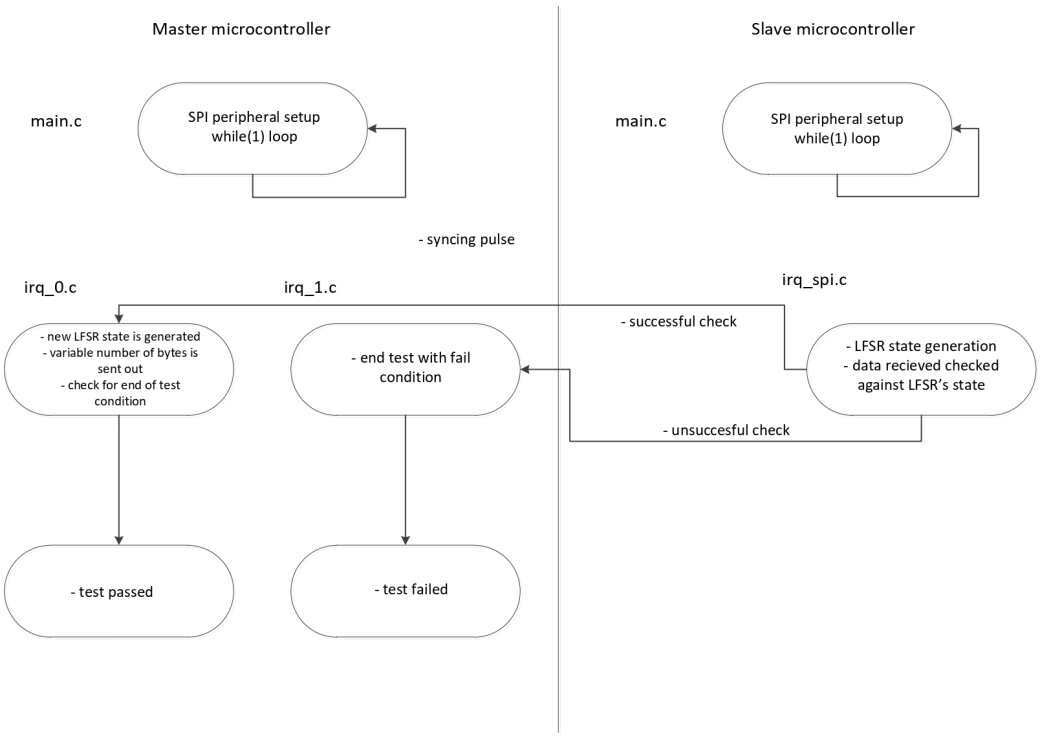



Figure 8.10 represent the state diagram of the operations of both microcontroller.

The test has been repeated for the following operation modes

- SPI peripheral in byte mode (TX and RX of data byte by byte). The first microcontroller sends a number of bytes between one and thirtytwo (full lenght of internal fifo).
- SPI peripheral in half word mode (TX and RX of data half word by half word). The first microcontroller sends a number of half word between one and sixteen (full lenght of internal fifo).
- SPI peripheral in word mode (TX and RX of data word by word). The first microcontroller sends a number of word between one and eight (full lenght of internal fifo).

Figure 8.10: SPI communication - state diagram



# Chapter 9

# Cordic

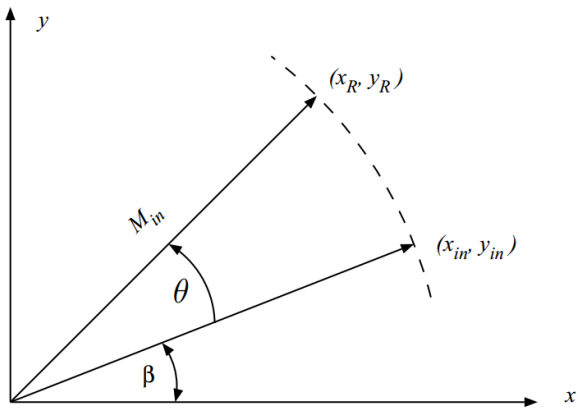
### 9.0.1 Cordic algorithm

The CORDIC algorithm (COordinate Rotation Digital Computer) is an efficient algorithm to compute

- trigonometric functions
- square roots
- hyperbolic functions
- multiplications and divisions
- exponentials and logarithms

A common use of this algorithm is in case of a microcontroller with no multiplication unit since it requires only additions, subtractions, bit-shifts and look up tables. To understand the theory behind the CORDIC algorithm it's necessary to take a step back. Suppose having a vector with coordinates $x_{in}$ $y_{in}$ and rotating it by an angle $\theta$, thus getting to the point of coordinates $x_r$ $y_r$.

Figure 9.1: Vector rotation




This rotation is represented by the following equations

- $x_r = x_{in}cos(\theta) - y_{in}sin(\theta)$
- $y_r = x_{in}sin(\theta) + y_{in}cos(\theta)$

Choosing $y_{in} = 0$ and $x_{in} = 1$ the previous equations become

- $x_r = cos(\theta)$
- $y_r = sin(\theta)$

Thus by knowing the value of $x_r$ and $y_r$ it's possible to calculate sine and cosine of an angle through a rotation. The previous general equations can be transformed into

- $x_r = cos(\theta)(x_{in} - y_{in}tan(\theta))$
- $y_r = cos(\theta)(x_{in}tan(\theta) + y_{in})$

So far no simplifications have been obtained, however the CORDIC algorithm is based on three premises

- rotating by one input angle is the same as rotating for several smaller angles
- the value of smaller angles can be chosen such that $tan(\theta_i) = 2^{-i}$ (angles stored in a LUT)
- the objective is to rotate an initial vector in order to align it with a new vector (which has an angle relative to the x axis of which sine and cosine have to be computed)

With the second premise the multiplication can be replaced by a bit shift (much easier in hardware). The first premise (dividing the original shift in a series of smaller shifts) provide the negative feedback mechanism of the CORDIC algorithm. An arbitrary angle is obtained with a successive rotation by the angles stored in the LUT; at each iteration the difference between the new vector angle and the angle in the LUT is recorded. The sequence of vector rotations can be represented by a decision vector

$z^{i+1} = z^i$ - $d_i \tan^{-1}2^{-i}$

where z represent the difference between the angle of the input vector and the vector being rotated; z is an angle accumulator. $d_i$ represent the direction of the current rotation, based on the value of the current z. Due to the initial choice for the elementary angles $\tan^{-1}2^{-i}$ is equal to the series of angles stored in the LUT. With the following assumptions

- value of smaller angles chosen such that $tan(\theta_i) = 2^{-i}$
- ignoring the constant gain factor ($cos(\theta)$) since for a sufficient number of iterations the cosine contribution becomes a constant equal to 0.6073

The three equations representing the values of x and y coordinates of the rotating vector and the decision vector become

- $x^{i+1} = x^i$ - $d_i 2^{-i} y^i$
- $y^{i+1} = y^i$ + $d_i 2^{-i} x^i$
- $z^{i+1} = z^i$ - $d_i \tan^{-1}2^{-i}$

### 9.0.2 Numerical example

The three equations in the previous sub section might seem not intuitive, a numerical example is necessary to have a clearer understanding of the CORDIC algorithm. To sum up the inputs of the CORDIC algorithm

- *x_in* is the initial location on the x coordinated of the rotating vector
- *y_in* is the initial location on the y coordinated of the rotating vector
- *z_in* is the angle of which sine and cosine are being computed

To compute sine and cosine the initial vector, as explained in the previous sub section, has to have coordinate (1,0). However at the end of the algorithm it's still necessary to divide the results by $cos(\theta)$; the cosine term act as gain (both $x$ and $y$ values are affected by it). The scaling operation can be avoided by passing the value 0.6072 instead of 1 for the x coordinate for the initial rotating vector.

| iteration | $d_i$ | $x\_in$ | $y\_in$ | $z\_in$ |
|---|---|---|---|---|
| - | - | 0.6072 | 0 | 70° |
| 0 | 1 | 0.6072 | 0.6072 | 25° |
| 1 | 1 | 0,3036 | 0,9108 | $-1.5651°$ |
| 2 | -1 | 0,5313 | 0,8349 | 12.4711° |
| 3 | 1 | 0,4269 | 0,9013 | 5.3461° |
| 4 | 1 | 0,3706 | 0,9279 | 1.7698° |
| 5 | 1 | 0,3415 | 0,9395 | $-0.0201°$ |
| | | $x\_r$ | $y\_r$ | $z\_r$ |

Table 9.1: CORDIC numerical example

The following example shows a numerical example for the computation of sine and cosine of an angle of 70°.

The real value of $cos(70)$ is 0.342, while the real value of $sin(70)$ is 0.9396. The values computed with the CORDIC algorithm approximate the real values closely even after only 5 iterations. For other input angles the approximation after such a small number of iteration is going to be less effective; increasing the number of iterations increases the result's precision.

### 9.0.3 CORDIC generalized equations

Sub section 9.0.1 presents a CORDIC's operative mode in which the input vector (with coordinates (1,0)) is rotated in order to be aligned to the vector which sine and cosine have to be computed. This operative mode is called rotation mode. The CORDIC algorithm has an additional operative mode, called vectoring mode; in vectoring mode the input vector is rotated by whichever angle in order to align it with the x axis. The vectoring mode allows the CORDIC algorithm to compute new functions, such as

- square root
- $tan^{-1}$
- $tanh^{-1}$

Moreover, depending on which angles are stored in the LUT, a number of additional operation can be performed (both in rotation and vectoring mode). As shown in the previous sub section, to compute sine and cosine the LUT is filled with angles such as

$\theta^i = \tan^{-1}(2^{-i})$ * 360/2*$\pi$

This choicce of angles put the CORDIC in circular mode. The other two options are linear mode and hyperbolic mode. The LUT is filled with angles such as

$\theta^i = 2^{-i}$ * 360/2*$\pi$ for the linear mode

$\theta^i = \tanh^{-1}(2^{-i})$ * 360/2*$\pi$ for the hyperbolic mode

The generalized equations describing the possible operations in both operative mode (rotation and vectoring mode) and with each angle choice (circular, linear and hyperbolic mode) are as follows

- $x^{i+1} = x^i$ - $\mu d_i 2^{-i} y^i$
- $y^{i+1} = y^i + d_i 2^{-i} x^i$
- $z^{i+1} = z^i$ - $d_i e^i$

The values of $\mu$, $d_i$ and $e^i$ change in case we are in rotation or vectoring mode, circular, linear or hyperbolic rotations. A complete overview of every combination of rotation mode and rotation type is the following

| | $x\_out$ | $y\_out$ | $z\_out$ |
|---|---|---|---|
| Circular | $K(x\cos(z) - y\sin(z))$ | $K(y\cos(z) + x\sin(z))$ | 0 |
| Linear | $x$ | $y + (x * z)$ | 0 |
| Hyperbolic | $K^*(x\cosh(z) - y\sinh(z))$ | $K^*(y\cosh(z) + x\sinh(z))$ | 0 |

Table 9.2: Rotation mode, $d_i$=$sign(z^i)$

| | $\mu$ | $e^i$ | $d_i$ |
|---|---|---|---|
| Circular | 1 | $\tan^{-1}2^{-i}$ | $sign(z^i)$ |
| Linear | 0 | $2^{-i}$ | $sign(z^i)$ |
| Hyperbolic | -1 | $\tanh^{-1}2^{-i}$ | $sign(z^i)$ |

Table 9.3: Rotation mode, $d_i$=$sign(z^i)$

| Vectoring mode, $d_i$=$-sign(y^i)$ | | | |
|---|---|---|---|
| | $x\_out$ | $y\_out$ | $z\_out$ |
| Circular | $\sqrt{K(x^2 + y^2)}$ | 0 | $z + \tan^{-1}(y/x)$ |
| Linear | $x$ | 0 | $z + (y/x)$ |
| Hyperbolic | $\sqrt{K^*(x^2 - y^2)}$ | 0 | $z + \tanh^{-1}(y/x)$ |

Table 9.4: Vectoring mode, $d_i$=$-sign(y^i)$

| | $\mu$ | $e^i$ | $d_i$ |
|---|---|---|---|
| Circular | 1 | $\tan^{-1}2^{-i}$ | $sign(y^i)$ |
| Linear | 0 | $2^{-i}$ | $sign(y^i)$ |
| Hyperbolic | -1 | $\tanh^{-1}2^{-i}$ | $sign(y^i)$ |

Table 9.5: Vectoring mode, $d_i$=$-sign(y^i)$

As it's possible to see from the two previous tables, depending on the values fed to the inputs (represented as $x$, $y$ and $z$) the outputs (represented as $x\_out$, $y\_out$ and $z\_out$) can express trigonometric functions, square roots, multiplication etc.. For instance to compute sine, cosine and square root

| | mode | $x_in$ | $y_in$ | $z_in$ |
|---|---|---|---|---|
| sine | Rotation - Circular | 0.6072 | 0 | angle |
| cosine | Rotation - Circular | 0.6072 | 0 | angle |
| square root | Vectoring - Hyperbolic | x + 0.25 | x - 0.25 | 0 |

Table 9.6: sine, cosine and square root inputs

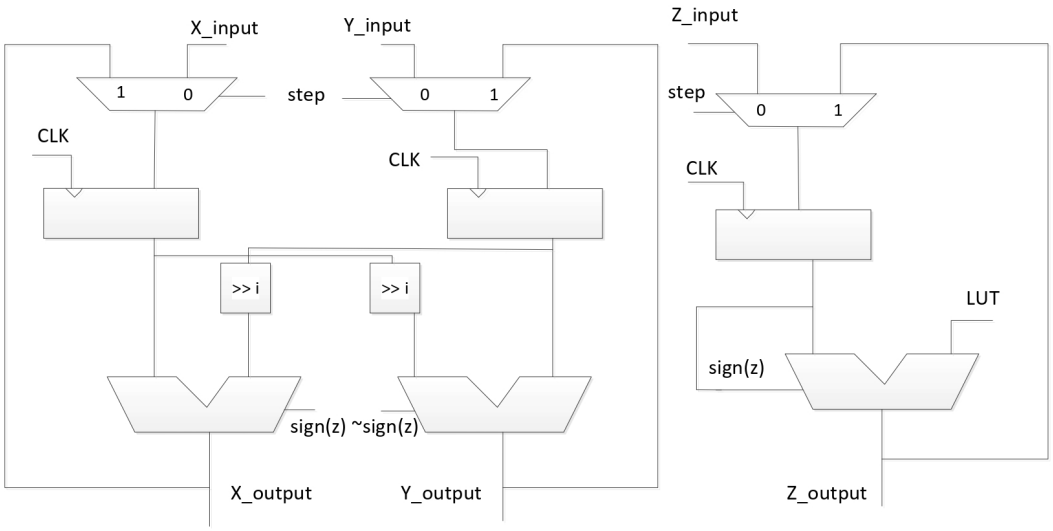
For the sine/cosine operation at the end of the CORDIC algorithm $x\_out$ is going to hold the value of the cosine of the initial angle while $y\_out$ is going to hold the value of the sine of the initial angle. Similarly for the square root operation at the end of the CORDIC algorithm $x\_out$ is going to hold the value of the square root of $x$. Similar input combinations can be found for other operations.

### 9.0.4 Hardware implementations

Before diving into the two implementations considered for the CORDIC algorithm it is necessary to give a visual representation of the equations presented in the previous subsection. As it is possible to see the following equations

- $x^{i+1} = x^i$ - $\mu d_i 2^{-i} y^i$
- $y^{i+1} = y^i + d_i 2^{-i} x^i$
- $z^{i+1} = z^i$ - $d_i e^i$

Figure 9.2: CORDIC architecture - Rotation mode



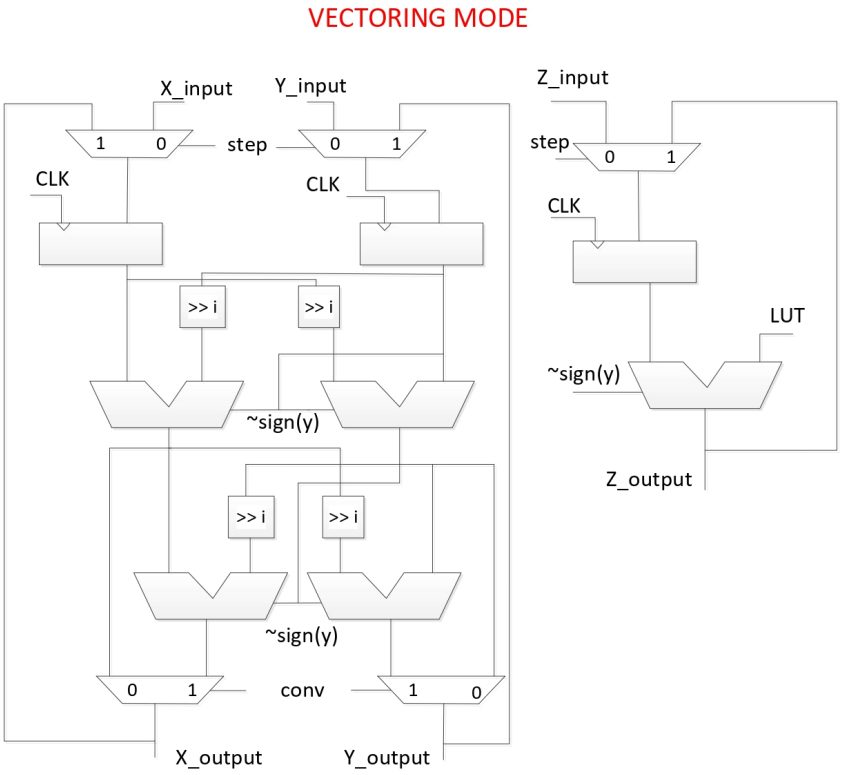

present a series of sum/subtractions and shifts to converge to the final result. To understand the possible implementations of the algorithm a single stage has to be observed; figure 9.1 present a single stage depending on the mode the CORDIC peripheral is put in. In fact in vectoring mode the shift-add stage has to be repeated for the steps that adheres to the equation $k^{i+1} = 3 * k^i + 1$, starting from $k^0 = 1$ [24]. Those iterations are $k = 4$, $k = 13$, $k = 40$, $k = 121$ etc.

Both implementations, folded and unfolded, make use of the above blocks; rotation mode and vectoring mode CORDIC architecture.

For the folded implementation a single stage is used; in particular the stage represented for the vectoring mode is used. Thanks to the ***conv*** signal (generated with particular timing, refer to subsection 9.0.3 for further explanations on the complete CORDIC peripheral design) the next iteration's result is chosen between the result after a single shift-add computation or the one generated with a double shift-add computation for convergence reasons. The final result is fed back to the input multiplexer in order to compute each successive result. At the end of the algorithm X_output and Y_output are sampled as they are the final CORDIC algorithm results.

For the unfolded implementation each stage is repeated, with each register functioning as pipelining registers for *X_output*, *Y_output* and *Z_output* coming from the previous stage; unlike the folded

Figure 9.3: CORDIC architecture - Vectoring mode




implementation both X_output, Y_output and Z_output are not fed back to the input multiplexer but are fed to the pipelining registers. Since the additional shifting present in the vectoring mode stage is required only at certain iterations, the stage used for each iteration in the generate loop is chosen according to the iteration itself. For instance for the first stage (iteration $i = 1$) a simple rotation mode stage is sufficient since the additional shift/add present in the vectoring mode stage is necessary for other iterations. For the fourth stage (iteration $i = 4$) the vectoring mode stage has to be used for convergence reasons. Figure 9.4 gives a visual representation of the CORDIC architecture in the unfolded version.

To evaluate the trade-off area/performance for the CORDIC algorithm implementations, one folded and one unfolded, are taken into considerations. As it's possible to see from figures 9.2 and 9.3, representing a single stage in rotation and vectoring mode, the two implementations differ vastly in area and performances, in particular tables 9.7 to 9.9 give a comparison between area and performance parameters betrween the folded and the unfolded implementation

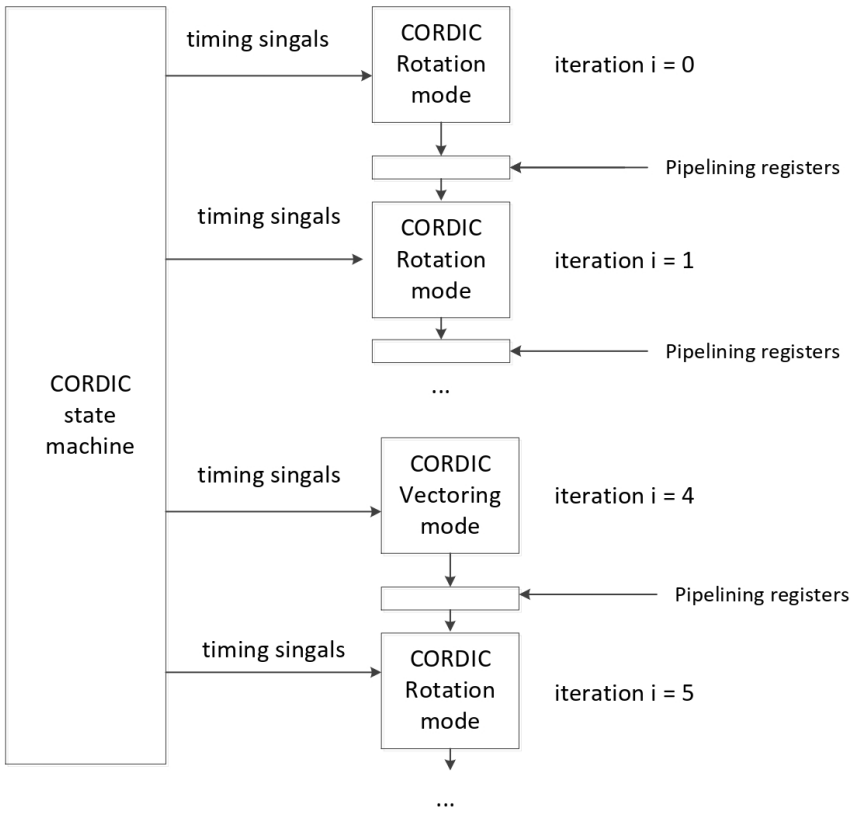
| Folded adders | Unfolded adders | Folded shifters | Unfolded shifters |
|---|---|---|---|
| 5 | 3 * (depth - conv_stage) + 5 * conv_stage | 4 | 2 * (depth - conv_stage) + 4 * conv_stage |

Table 9.7: Area parameters

| Folded LUTs | Unfolded LUTs | Folded registers | Unfolded registers |
|---|---|---|---|
| 1 | 1 | 3 | 3 * depth |

Table 9.8: Area parameters

Figure 9.4: CORDIC unfolded architecture




| Folded number of cc for operation | Unfolded number of cc for operation | Folded max achievable throughput | Unfolded max achievable throughput |
|---|---|---|---|
| depth | depth | 1/depth ops/cc | 1 ops/cc |

Table 9.9: Performance parameters

Due to the excessive area cost of the unfolded implementation, the folded implementation has been chosen as preferred CORDIC core implementation. Table 9.10 gives the area parameter, after being synthesized on FPGA, for both core implementations with 15 iterations and data width for $x$, $y$ and $z$ of 32 bits

| | Folded | Unfolded |
|---|---|---|
| Slice LUTs | 465 | 2865 |
| Slice registers | 96 | 1421 |

Table 9.10: CORDIC core area comparison

### 9.0.5 Complete CORDIC peripheral design

The CORDIC peripheral designed needed to be compatible with the APB bus (similarly to the other peripherals examined in the previous chapter) in order to be able to communicate easily with the RISC-V microcontroller. The complete CORDIC peripheral is thus composed of

- The CORDIC core, as seen in the previous subsection
- The APB wrapper, a digital block capable of acting as a bridge between the CORDIC core and the APB bus

An overall picture can be observed in figure 9.10.
As highlighted before the APB wrapper is responsible for two operations, namely

- Communications with the APB bus through a bank of registers
- Controlling the CORDIC core through control signals

#### CORDIC register organization

As done for the previous peripherals designing the peripheral's register organization allows us to communicate effectively with the, in this specific case, CORDIC core. The register organization is the following

| register name | address | access | description |
|---|---|---|---|
| CORDIC_CTRL | 0x00 | RW | control register |
| CORDIC_STATUS | 0x04 | R | status register |
| CORDIC_X_DATA | 0x08 | RW | x register data in |
| CORDIC_Y_DATA | 0x0C | RW | y register data in |
| CORDIC_Z_DATA | 0x10 | RW | z register data in |
| CORDIC_X_OUT | 0x14 | R | x register data out |
| CORDIC_Y_OUT | 0x18 | R | y register data out |

Table 9.11: CORDIC register organization

Every CORDIC register is 32 bits wide, however, depending on the register, only a subset is used. Tables 9.12 to 9.15 give a bit by bit representation of each CORDIC register content

| register name | bit 0 | bit 1 | bits 2-31 |
|---|---|---|---|
| CORDIC_CTRL | rot mode start | vector mode start | unused |

Table 9.12: CORDIC register organization - CORDIC control register

| register name | bit 0 | bit 1 | bits 2-31 |
|---|---|---|---|
| CORDIC_STATUS | status | status clear | unused |

Table 9.13: CORDIC register organization - CORDIC status register

| register name | bit 0-31 |
|---|---|
| CORDIC_X_DATA | x data in |
| CORDIC_Y_DATA | y data in |
| CORDIC_Z_DATA | z data in |

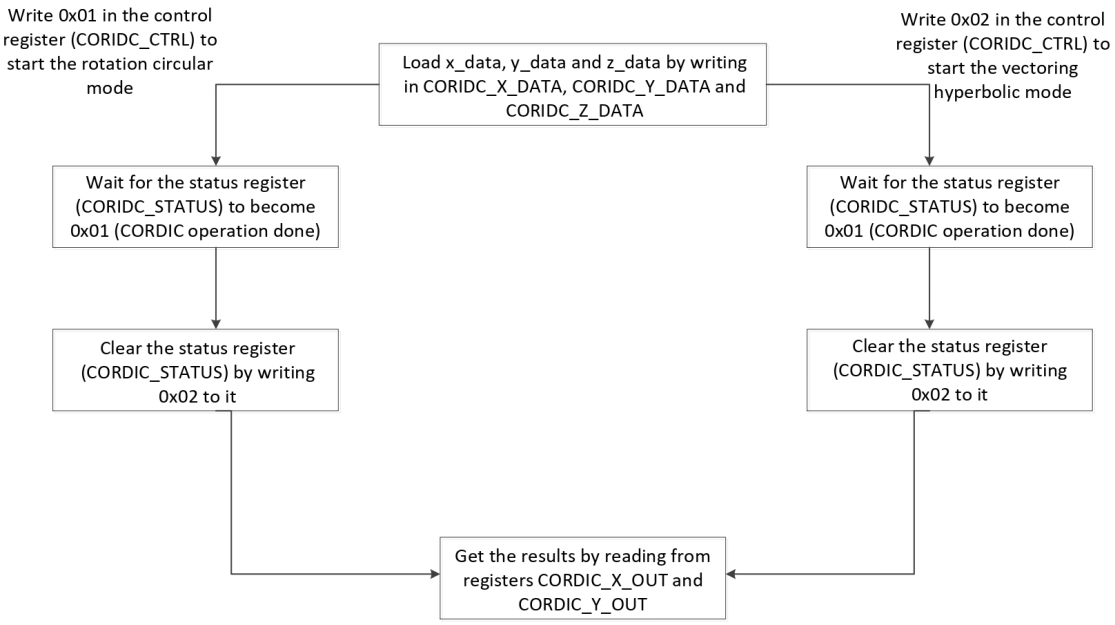
Table 9.14: CORDIC register organization - CORDIC x, y and z data in

A typical sequence of operations for the CORDIC peripheral is represented in figure 9.5.

| register name | bit 0-31 |
|---|---|
| CORDIC_X_OUT | x data out |
| CORDIC_Y_OUT | y data out |

Table 9.15: CORDIC register organization - CORDIC x and y data out

Figure 9.5: CORDIC operation




### CORDIC state machine

To control the CORDIC core a state machine has been implemented; the purpose for this particular state machine is to control, with the right timing, the *control* signals to

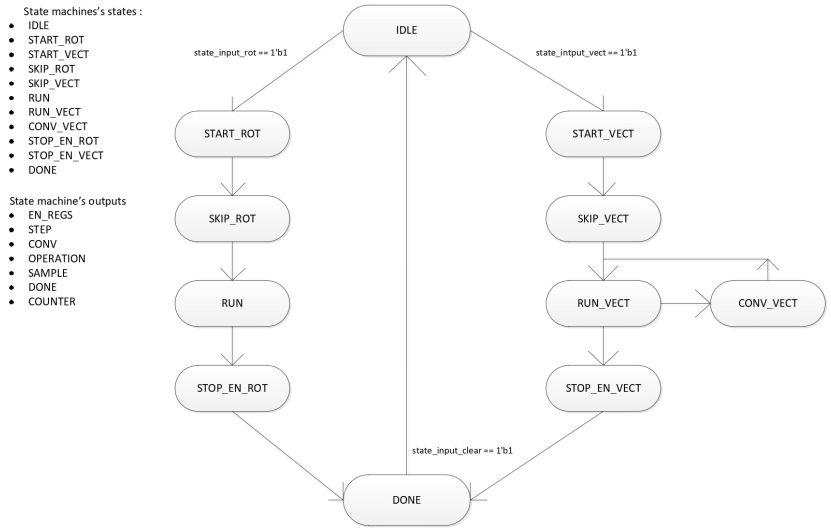
- Handle muxes control input present in the CORDIC core according to the mode selected
- Generate the sampling and done signals used by the APB wrapper itself to effectively communicate with the RISC-V microcontroller about its state of operations

The complete state machine's operation is represented in figure 9.3

The output signals coming from the state machine serve the following purpose

- EN_REGS : registers enable going to the CORDIC stage
- STEP : signal used to control the multiplexer between X_input and X_output
- CONV : signals used to control the multiplexer between the standard X_next result and the result produced for convergence reason at iteration $k^{i+1} = 3 * k^i + 1$
- OPERATION : signals used to determine the operation mode (rotation or vectoring)
- SAMPLE : signal used by the APB wrapper to sample the data coming from the CORDIC core

Figure 9.6: State machine's operation



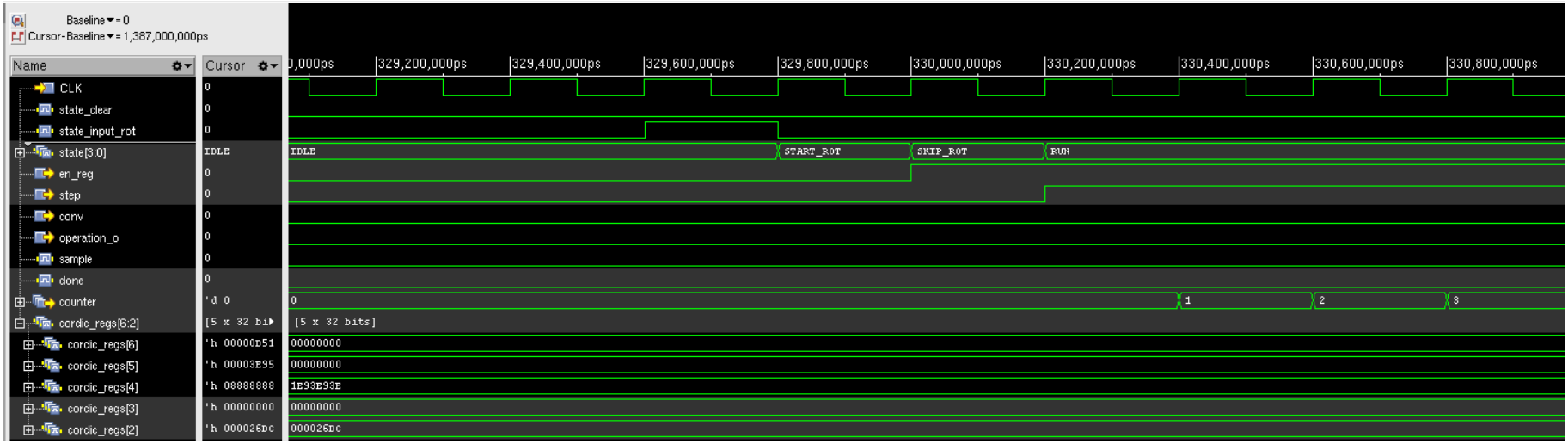

- DONE : signal used by the APB wrapper to indicate that the current computation has finished
- COUNTER : integer used to indicate the current iteration of the CORDIC algorithm. The CORDIC core uses this integer to shift by the right amount the current data.

The states of the state machine are as follows

- IDLE : idle state, is the starting state at reset and after each computation
- START_ROT : a CORDIC computation in rotation mode is started
- SKIP_ROT : first iteration of a computation in rotation mode
- RUN : CORDIC algorithm in rotation mode
- STOP_EN_ROT : last iteration for a computation in rotation mode
- START_VECT : a CORDIC computation in vectoring mode is started
- SKIP_VECT : first iteration of a computation in vectoring mode
- RUN_VECT : CORDIC algorithm in vectoring mode
- CONV_VECT : CORDIC algorithm in vectoring mode during a convergence iteration
- STOP_EN_VECT : last iteration for a computation in vectoring mode
- DONE : computation done

In figures from 9.4 to 9.6 is an example of the state machine operation. As an example only the rotation mode has been represented (left branch of the diagram in figure 9.3). A code snippet of the whole state machine is in the appendix chapter.
As it's possible to see in figure 9.4 the rotation mode is activated by an high pulse of the signal *state_input_rot* while on $IDLE$ state; the rotation mode starts by activating the registers through $EN\_REGS$ while in $START\_ROT$ state, asserting $STEP$ one clock cycle later (while in $SKIP\_ROT$ state) and then activating the counter (while in $RUN$ state). In figure 9.5 it's possible to see the final part of the computation; during the $DEPTH - 1$ stage the state changes to $STOP\_EN\_REG$, where an high pulse of $SAMPLE$ is activated (thus sampling the result at the following clock cycle). During the next clock cycle (thus in $DONE$ state) the result is sampled and stored in *cordic_regs*[5] and *cordic_regs*[6]. The state is stuck to $DONE$ until an high pulse of *state_clear* which reset the state machine to the $IDLE$ state (as in figure 9.6).

Figure 9.7: State machine initial operation



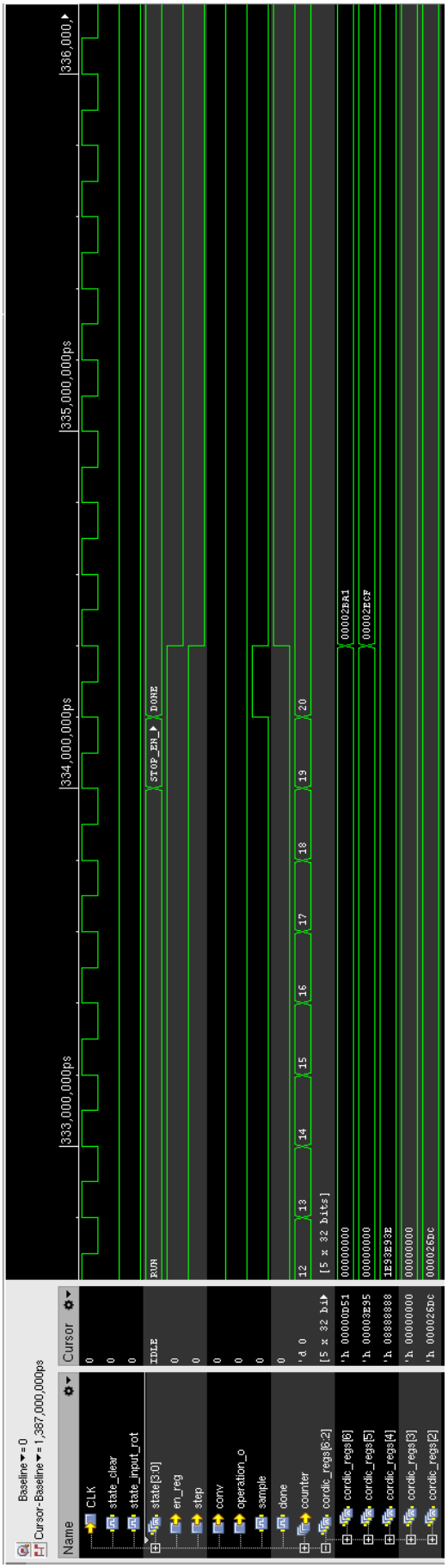



Figure 9.8: State machine intermediate operation

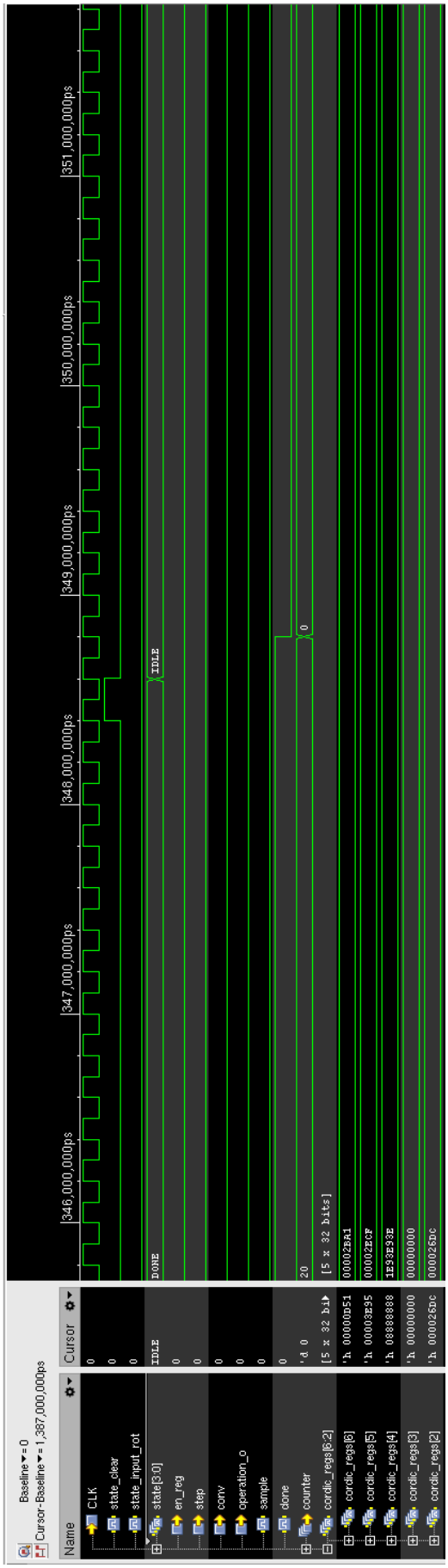
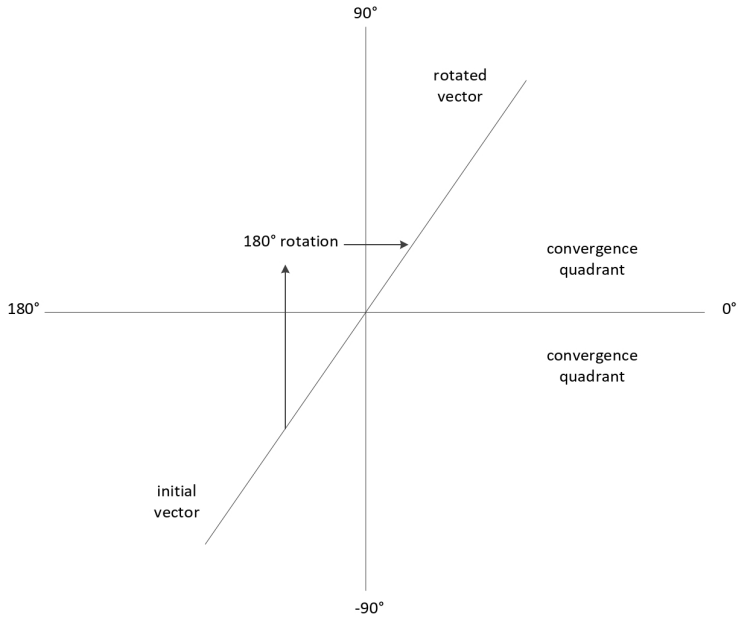



Figure 9.9: State machine final operation

### CORDIC pre and post processing

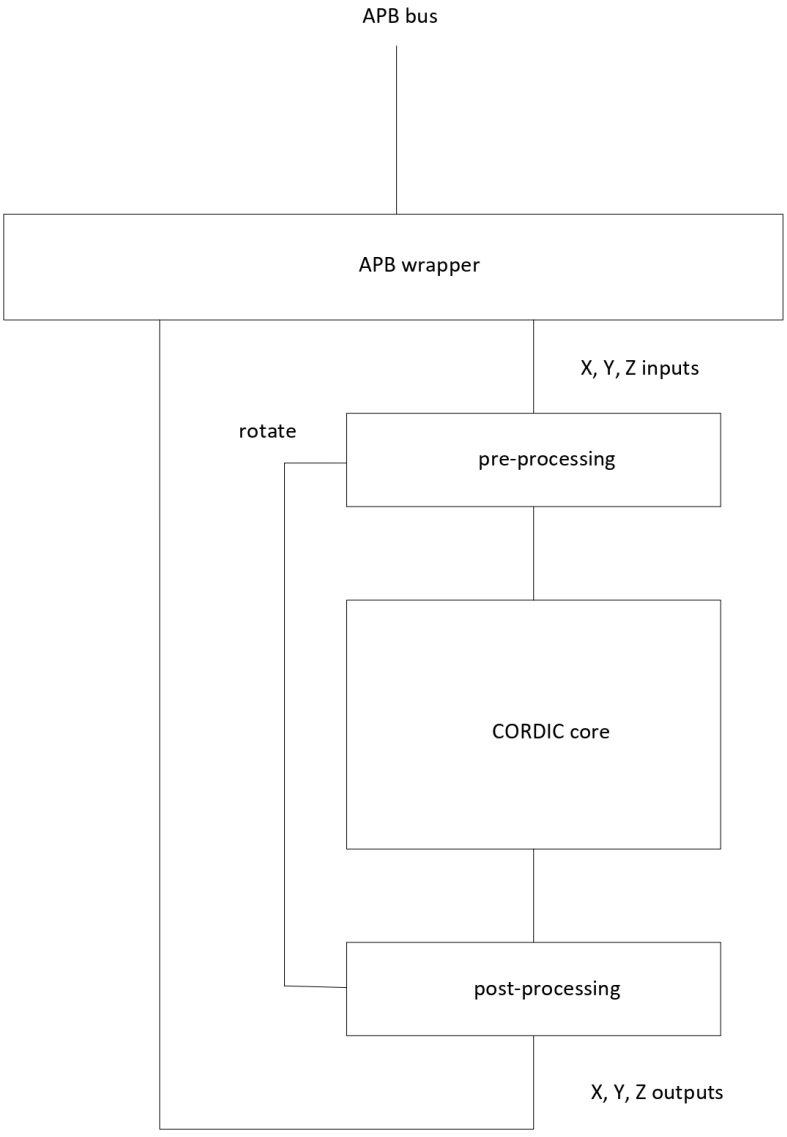
The CORDIC core embed a pre and post processing block; these two blocks are used during the sine-cosine operations in order to place the angle received as input in the convergence interval for the algorithm itself. The CORDIC algorithm, in fact, for the sine-cosine operation converges only if the input angle is between 90° and -90°. This is due to the fact that the sum of the angles stored in the LUT amounts to around 100° (99.86° for the first 12 entries in the LUT). Limiting the convergence range to 90° -/- -90° ease the rotation decision.

Figure 9.10: Cordic rotation




The pre-processing block looks at the input angle and rotates it by 180° if it's outside the restricted convergence interval. If the input angle gets shifted the pre-processing block raises a signal going to the post processing block. The post-processing block is responsible of outputting the results for sine and cosine; if the pre-processing block raises the *rotate* signal the post-processing block outputs the negated results obtained from the CORDIC core. Figure 9.10 gives a visual representation of the convergence interval and the rotation operation for a vector outside of it.

Figure 9.11: CORDIC complete architecture



### 9.0.6 Data representation

A fixed point integer representation for x and y data has been adopted. This particular choice, explained further in the next two subsections, has one advantage and one drawback, namely

- Ease the design phase allowing the use of integer adder/subtractor instead of floating point ones (saving area and complexity in the process)
- To get a *real*, as in floating point, representation of the results a further floating point division is needed. This division is handled by the microcontroller core

To be noted is that this floating point division is achieved by multiple operations, not necessarily involving division/multiplication operations. The GCC compiler is able in fact to translate a floating point division in a series of integer operations.

For the z data, or the data representing the angle, a different approach has been adopted, called binary scaling [25]. The binary scaling approach is a way of representing angles in such a way that the addition/subtraction operations would work in two's complement. The idea behind binary scaling is to use 32 bits to represent 360 degrees. In particular

| 0° | 90° | 180° | 270° |
|---|---|---|---|
| 0x00000000 | 0x40000000 | 0x80000000 | 0xC0000000 |

Table 9.16: CORDIC angle representation

This type of angle representation gives a similar resolution than a floating point one considering that the angles interval is (very) limited.

#### Cosine and sine operations

As highlighted in the first section, the CORDIC peripheral can be used to compute sine and cosine of an angle; this is achieved through

- Load the angle in register z
- Load register x with 1/K (scaling factor)
- Load register y with 0
- Start the CORDIC core in circular rotation mode

At the end of the computation the value of Z_output is going to converge to zero, while X_output and Y_output are going to be cosine and sine of the angle placed in register z respectively.

As explained in the previous subsection, this particular peripheral has an integer fixed point representation for the numbers in X and Y registers. The following example (on 16 bits, but the idea behind it is applicable to any number of bits) shows how the number 1/K (namely 0.607253) is represented.

| bit 15 | bit 14 | bits 13-0 |
|---|---|---|
| sign bit | integer part bit | fractional part bits |

Table 9.17: CORDIC data representation

To represent the number 1.75 we would have

This kind of representation allows us to retain the maximum resolution on the final result. As explained before the final result (an integer number in this case represented in 16 bits) have to be divided by 16384 (2 to the 14th, least significant bit of the integer part) to obtain a floating point number in the range of +1/-1 representing sine and cosine of the initial angle. In our specific case the number 1/K (0.607253) can be approximated to

This number would then be loaded into the X_data_in register to start the sine/cosine computation.

| bit 15 | bit 14 | bit 13 | bit 12 | bits 11-0 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 |

Table 9.18: 1.75 representation

| bit 15 | bit 14 | bit 13 | bit 12 | bit 11 | bit 10 | bit 9 | bit 8 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |

Table 9.19: 0.607253 representation

| bit 7 | bit 6 | bit 5 | bit 4 | bit 3 | bit 2 | bit 1 | bit 0 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |

Table 9.20: 0.607253 representation

**Square root operation**

To compute the square root of a number$v$ it is needed to

- Load register X_DATA_IN with *v + 0.25*
- Load register Y_DATA_IN with *v - 0.25*
- Start the CORDIC core in hyperbolic vectoring mode

The CORDIC algorithm for the square root is particularly sensitive to inputs outside the range 2/0.5, this is the reason why an initial pre-processing on the input $v$ is needed. This pre-processing is needed to find the value $n$, which is the number of leading zeros in the binary representation of $v$, needed to verify the equation $v = u * 2^n$ with the value of $u$ comprised between 0.5 and 2. This relation is useful since at the end of the CORDIC algorithm with the pre-processing step it's possible to obtain the value of the square root of $v$ since $\sqrt{v} = \sqrt{u}\ 2^{n/2}$. Given this last equation it's easy to see the reason for a further condition on $n$; $n$ has to be an even number.

The following example (on 16 bits, but the idea behind it is applicable to any number of bits) clarifies the steps taken to compute the square root of an integer number $v$. Let's suppose $v = 7$, which in binary has a representation of 111. The first step is to find the number of bits of $v$, which in our case is three. Since the number $u$ that we are going to find at the end of this pre-processing phase has to be comprised between 0.5 and 2 our 16 bits integer representation is going to be

| bit 15 | bit 14 | bits 13 | bits 12-0 |
|---|---|---|---|
| sign bit | integer part bit | integer part bit | fractional part bits |

Table 9.21: CORDIC data representation

After finding the number $n$ the binary representation of the number $v$ has to be shifted left following this rule

- If even shift left by *15 - n - 2*
- If odd shift left by *15 - n - 1*

This rule guarantees that the value of *v + 0.25* doesn't overflow and changes accidentally the sign bit. In our representation the value of 0.25 is

At this point the shifted value of 0.25 is added and subtracted to $v$, thus generating the values to be loaded into the registers X_DATA_IN and Y_DATA_IN. At the end of the CORDIC computation with the before mentioned values in register X_DATA_OUT is present *u / K*, where K is the scaling factor (namely 1.2075). This value is processed as follows, depending on the parity of $n$ computed in the first pre-processing step

| bit 15 | bit 14 | bits 13 | bits 12 | bit 11 | bits 10-0 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 |

Table 9.22: 0.25 representation

- If even the result is shifted left by *n/2*
- If odd the result is shifted left by *(n-1)/2*

This intermediate result (corresponding to $\sqrt{v}$ / $K$) has to be multiplied by the scaling factor (1.2075). As in the cosine/sine operation this result (now corresponding to $\sqrt{v}$) has to be divided by 8192 (2 to the 13th, least significant bit in the integer part).

### 9.0.7 Testing phase

To test the correctness of the CORDIC peripheral an automatic test has been developed. This test compares the results for sine, cosine and square root obtained through the CORDIC peripheral against the results obtained through math.h C library. A code snippet for this test can be found in the appendix chapter. Every result have been printed using the UART peripheral, and the printed results have been redirected to a log file thanks to the serial monitor Tera Term. This process has been repeated for different number of total iterations of the CORDIC algorithm, in particular 10, 15 and 20.

To guarantee that the comparison between cordic and math.h results is a reasonable one the same comparison between math/h's results and excel's results (taken as golden standard) is performed. Having a reasonably small diference between the two methods allows to consider math.h's results as the golden standard.

The following tables and histograms give a visual representation of the comparison results

| median_sin | average_sin | median_cos | average_cos | median_sqrt | average_sqrt |
|---|---|---|---|---|---|
| 1,1270 $10^{-4}$ | 1,2970 $10^{-4}$ | 1,1978 $10^{-4}$ | 1,4219 $10^{-4}$ | 8,6524 $10^{-2}$ | 8,2140 $10^{-2}$ |

Table 9.23: CORDIC 20 results

| median_sin | average_sin | median_cos | average_cos | median_sqrt | average_sqrt |
|---|---|---|---|---|---|
| 1,1270 $10^{-4}$ | 1,2970 $10^{-4}$ | 1,0398 $10^{-4}$ | 1,2305 $10^{-4}$ | 8,5147 $10^{-2}$ | 8,0540 $10^{-2}$ |

Table 9.24: CORDIC 15 results

| median_sin | average_sin | median_cos | average_cos | median_sqrt | average_sqrt |
|---|---|---|---|---|---|
| 2,8213 $10^{-4}$ | 3,3816 $10^{-4}$ | 9,0920 $10^{-5}$ | 3,3816 $10^{-4}$ | 1,2590 $10^{-1}$ | 1,2475 $10^{-1}$ |

Table 9.25: CORDIC 10 results

| median_sin | average_sin | median_cos | average_cos | median_sqrt | average_sqrt |
|---|---|---|---|---|---|
| 2,9126 $10^{-8}$ | 4,4959 $10^{-8}$ | 3,0738 $10^{-8}$ | 4,9017 $10^{-8}$ | 2,0013 $10^{-6}$ | 2,0391 $10^{-6}$ |

Table 9.26: math.h results

Given the fact that the absolute and the percent error between math.h's results and excel's results differ by a factor of $10^{-8}$ for the absolute error of sine and cosine and by a factor of $10^{-6}$ for the percent error of the square root the approximation above (considering math.h's and excel's results as the golden standard) holds true.

Figure 9.12: CORDIC 20 sin results

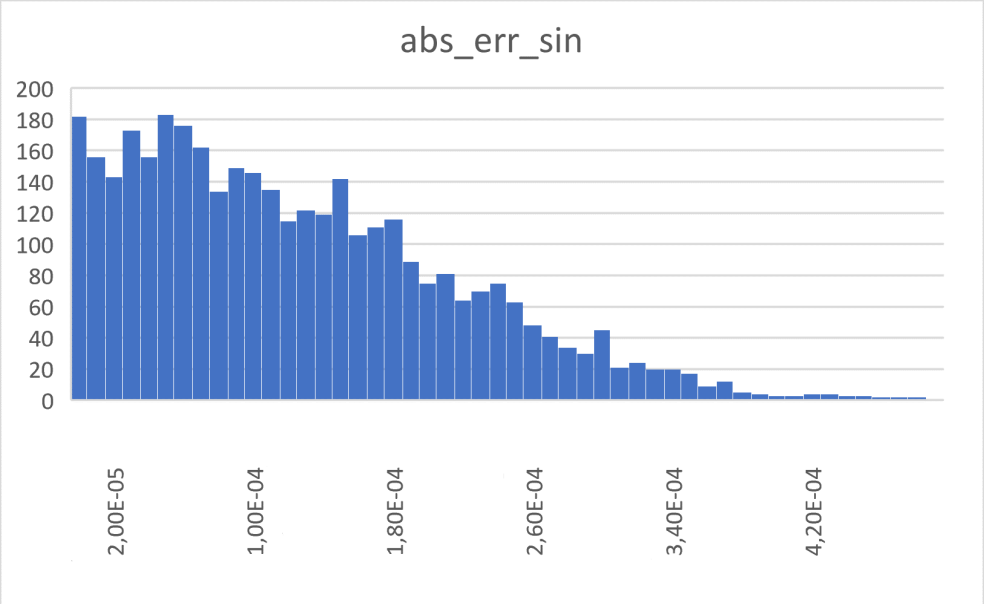



Figure 9.13: CORDIC 20 cos results

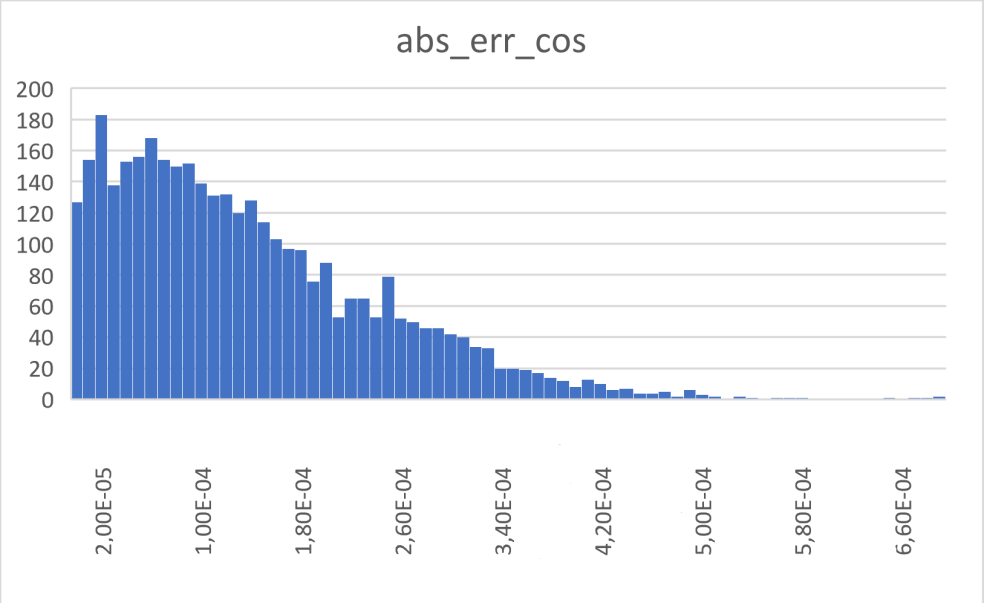


Figure 9.14: CORDIC 20 sqrt results

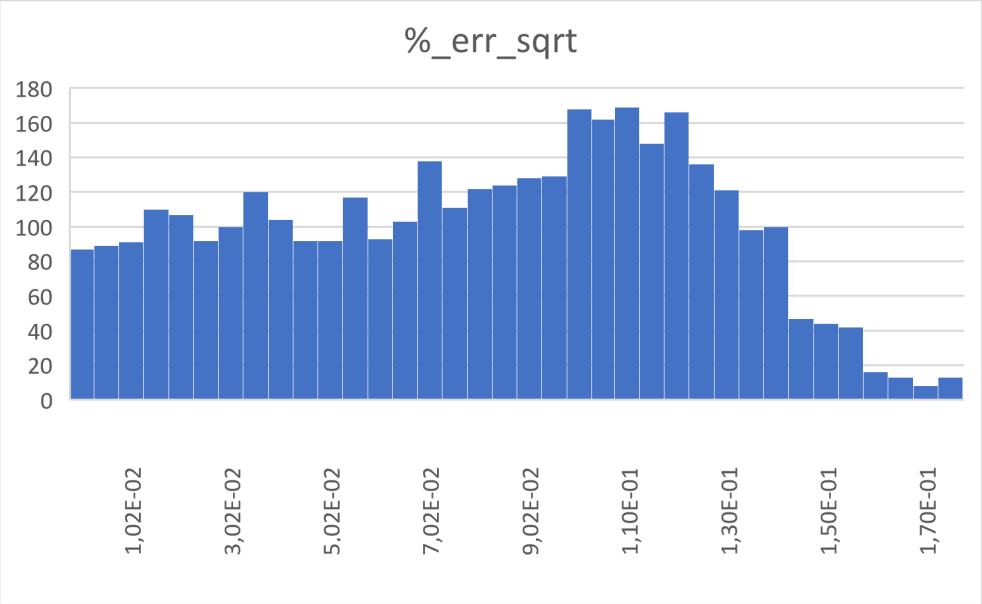



Figure 9.15: CORDIC 15 sin results

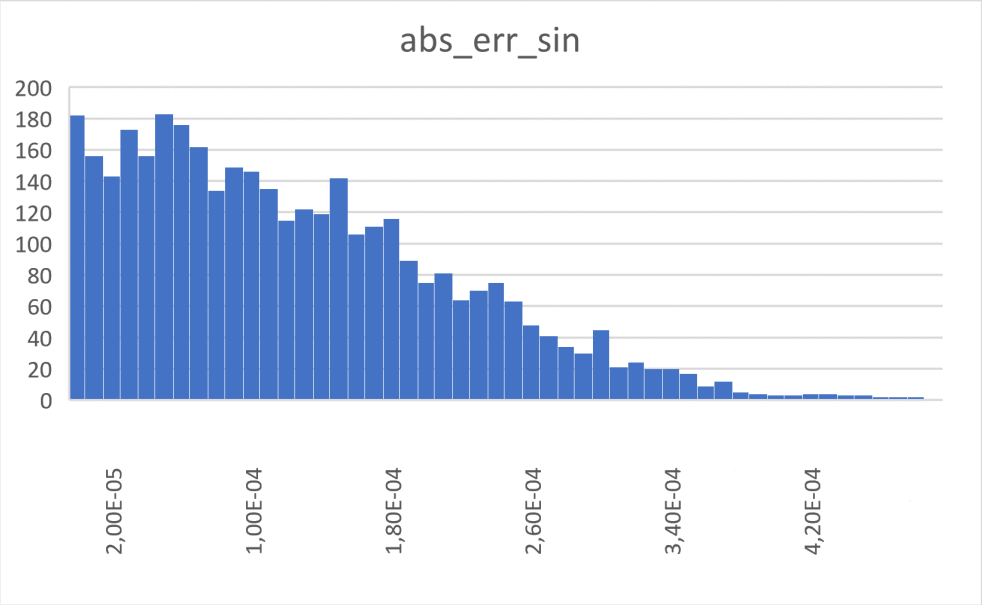


Figure 9.16: CORDIC 15 cos results

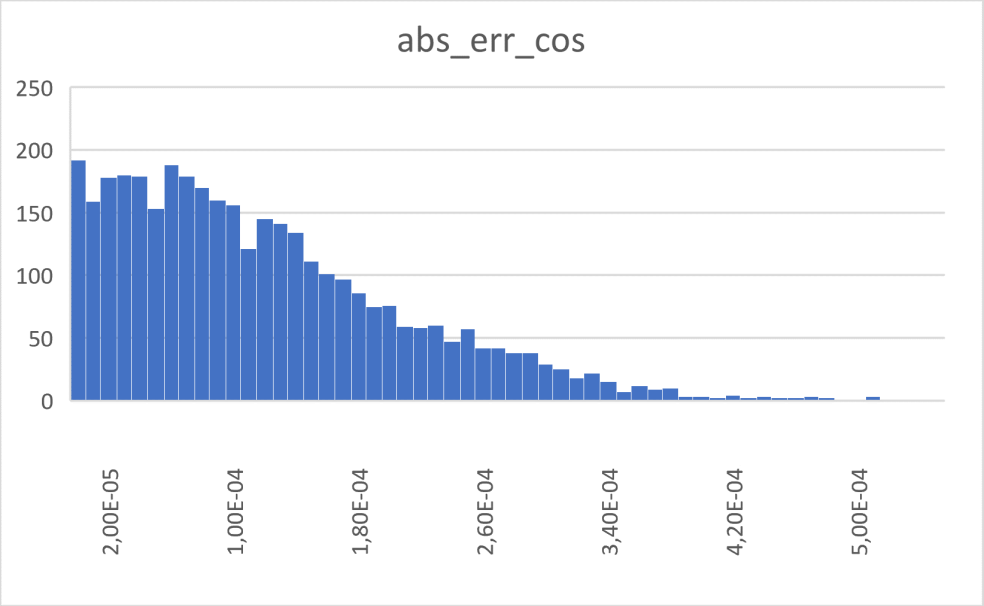



Figure 9.17: CORDIC 15 sqrt results

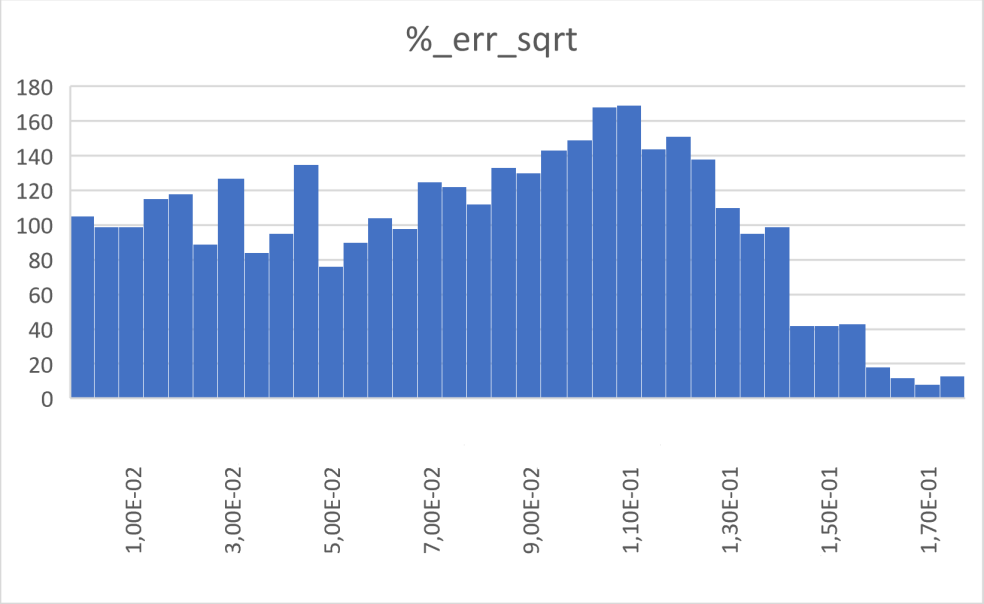


Figure 9.18: CORDIC 10 sin results

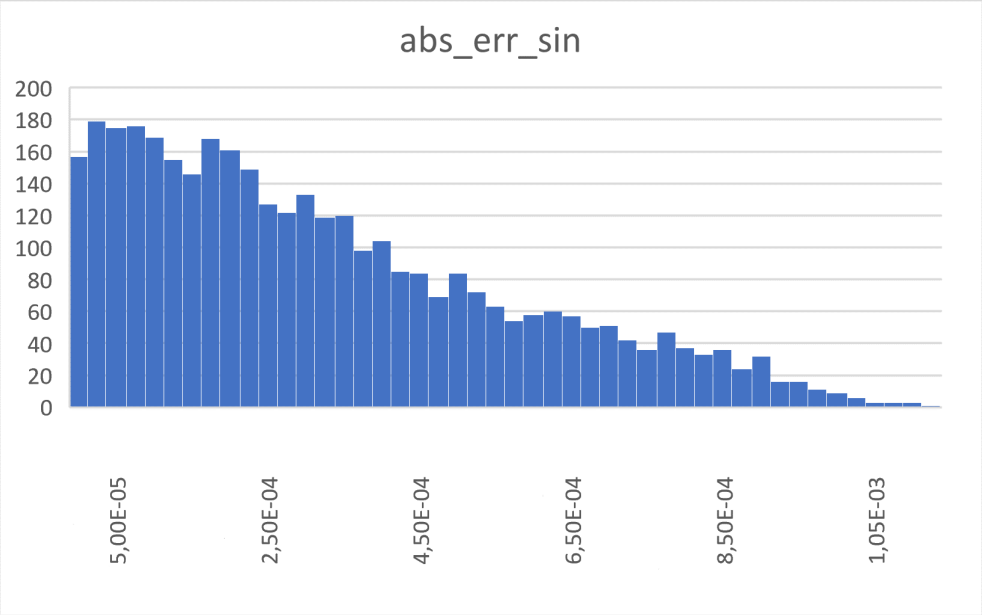



Figure 9.19: CORDIC 10 cos results

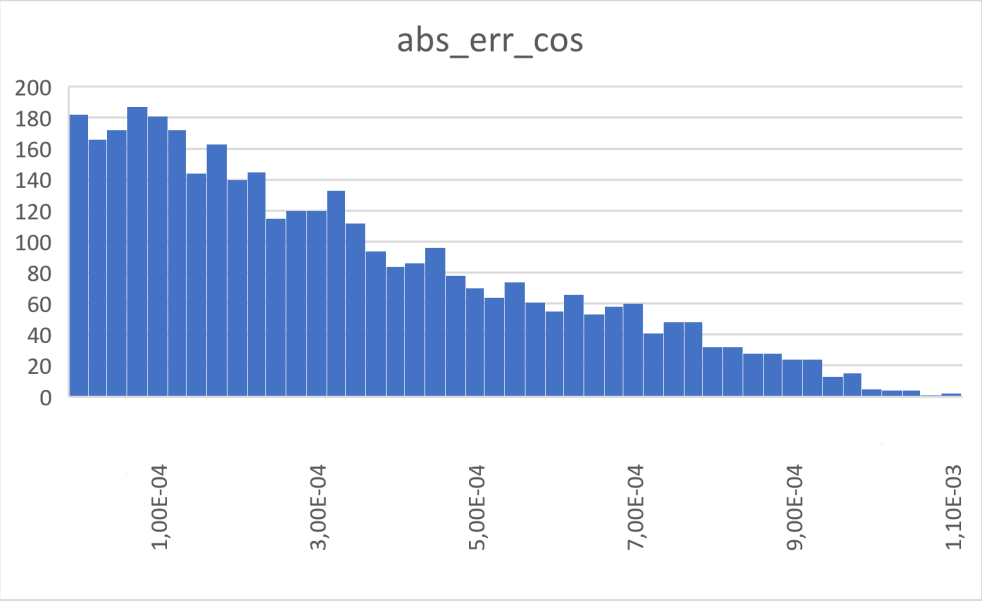


Figure 9.20: CORDIC 10 sqrt results

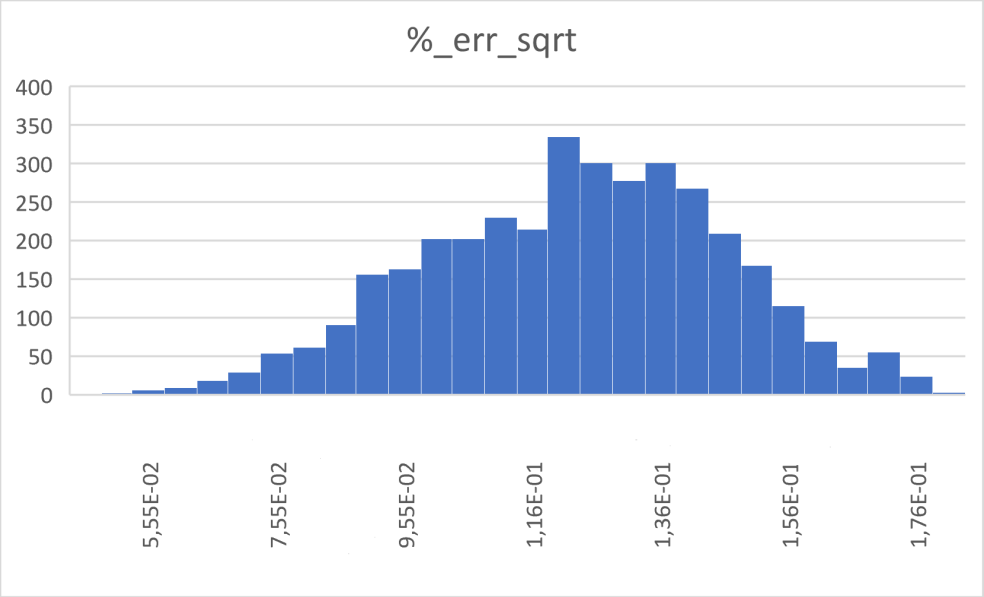



Figure 9.21: math.h sin results

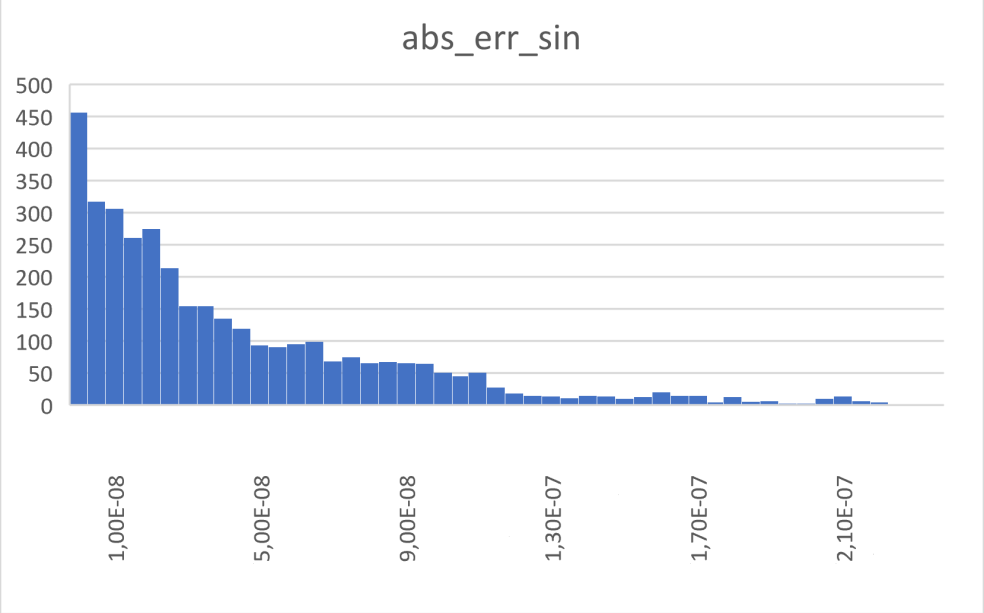


Figure 9.22: math.h cos results

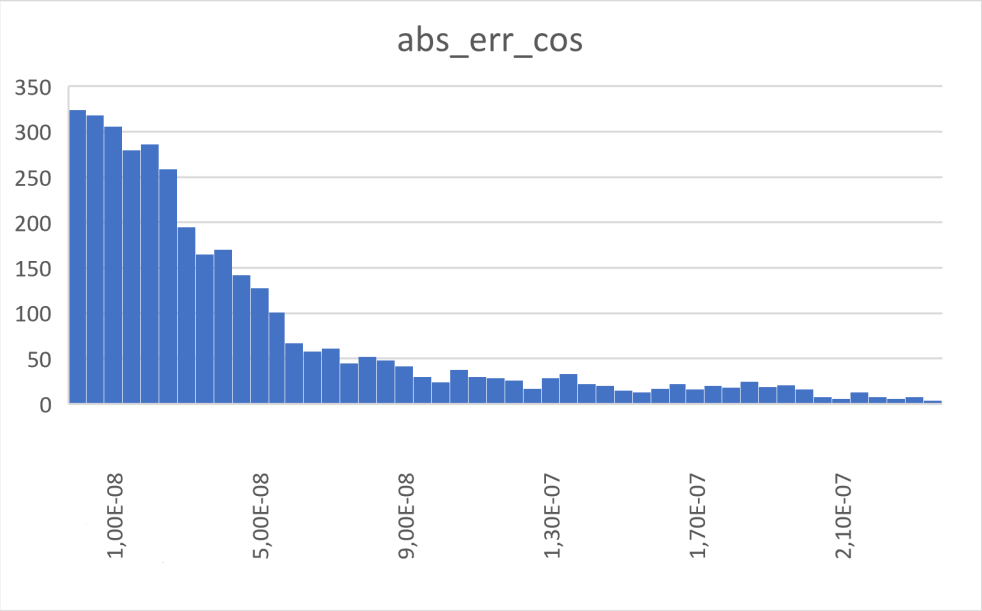
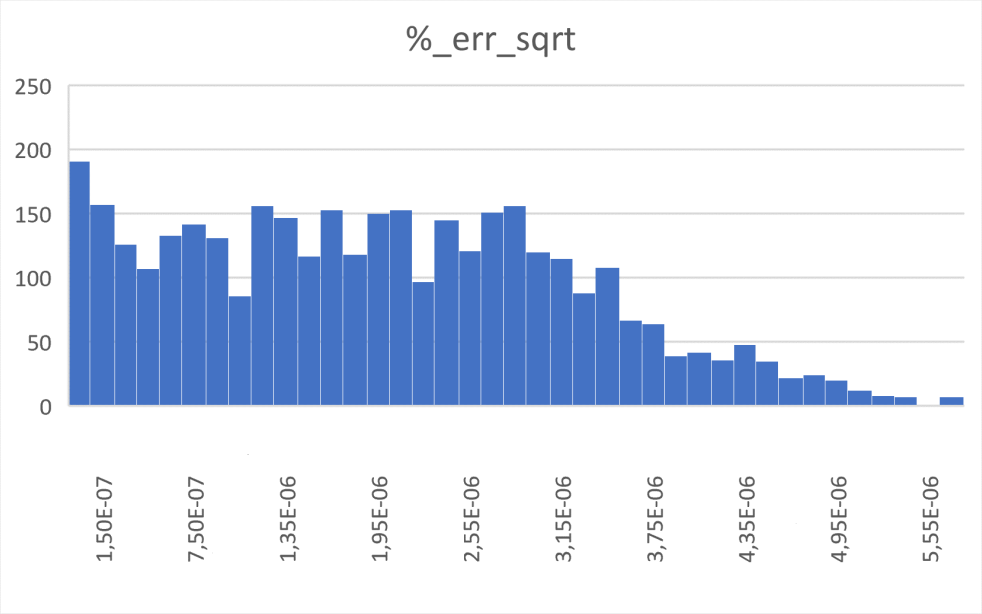



Figure 9.23: math.h sqrt results



The graphs show that the histograms for the 15 and 20 iteration implementations present similar results; the difference in performance can be seen looking at tables 9.23 and 9.24. The 20 iterations implementation has slightly worse median and average results. This is due to the fact that continuing the CORDIC algorithm once the convergence is reached tend to impact the final result, thus degrading it. Unsurprisingly the 10 iteration implementation has the worst median and average results for each operation out of the three implementations tested (not enough steps have been performed for the result to converge to the correct one).

**math.h comparison**

As highlighted in the previous subsection the initial correctness test has been performed comparing the CORDIC results to the math.h outputs; a further comparison between the CORDIC and math.h approach for the sine, cosine and square root operation is necessary. This additional comparison gauge the performance of a custom peripheral approach against a software one. The results for time and memory occupation are taken for

- A sine operation followed by a cosine operation of the same angle
- A square root operation

Both operations are repeated in case of *IMC* and *IC* instructions sets; this means that in the *IC* case the multiplication unit is not used.

| CORDIC time [cc] | CORDIC ROM | math.h time [cc] | math.h ROM |
|---|---|---|---|
| 3130 | 9 kB | 24417 | 19 kB |

Table 9.27: sine and cosine operations *IMC*

| CORDIC time [cc] | CORDIC ROM | math.h time [cc] | math.h ROM |
|---|---|---|---|
| 6136 | 9.5 kB | 57135 | 19.5 kB |

Table 9.28: sine and cosine operations *IC*

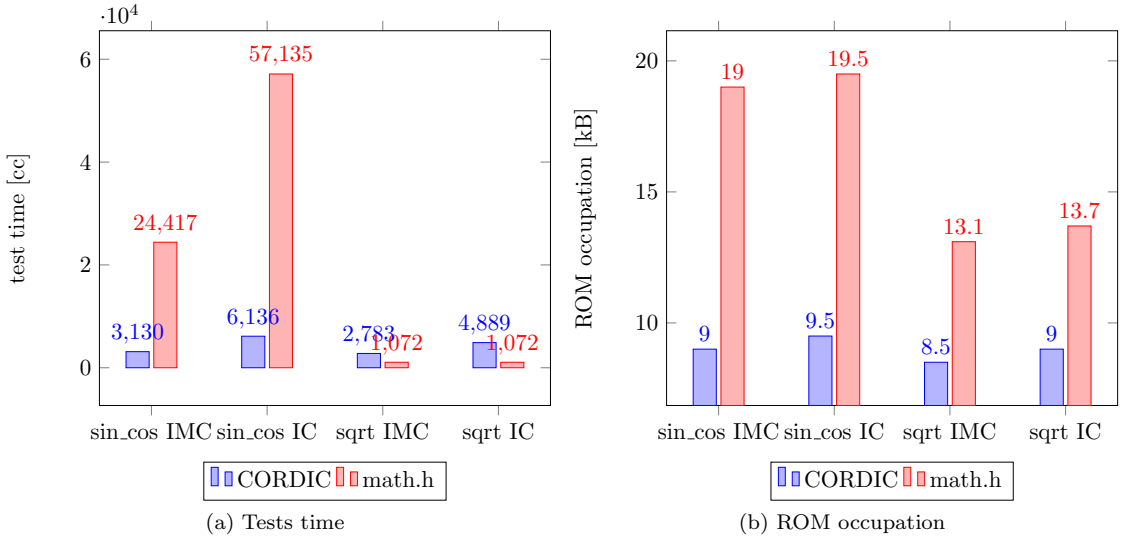
| CORDIC time [cc] | CORDIC ROM | math.h time [cc] | math.h ROM |
|---|---|---|---|
| 2783 | 8.5 kB | 1072 | 13.1 kB |

Table 9.29: square root operation *IMC*

| CORDIC time [cc] | CORDIC ROM | math.h time [cc] | math.h ROM |
|---|---|---|---|
| 4889 | 9 kB | 1072 | 13.7 kB |

Table 9.30: square root operation *IC*

As it's possible to see every implementation done through the CORDIC approach has a clear edge on the memory occupation side; in fact for both tests, sine-cosine and square root, in both instruction set conditions, *IMC* and *IC*, the CORDIC approach presents a smaller ROM occupation (in case of the sine-cosine operations tests the CORDIC ROM occupations amount to half of the software approach). In terms of test time the CORDIC approach presents a clear advantage in the sine-cosine tests (test time is between 8 to 9 times faster), however for the square root test the software approach is between 3 to 5 times faster. Figure 9.6 gives a visual representation for the previous results




(a) Tests time

(b) ROM occupation

Figure 9.24: CORDIC - software approach comparison

### 9.0.8 Further improvements

A shortcoming of this CORDIC peripheral implementation is the fact that the data is represented in an integer fixed point representation. This is necessary for the CORDIC algorithm to work (the algorithm works on integer numbers), however poses a significant drawback; the data has to be converted back to floating point by software. A further improvement would be to place a *float to int* and an *int to float* converter both in the pre and post processing blocks. This would allow to place floating point data in the input registers (useful for computing the square root of a floating point number) and to retrieve an already converted to *float* number, thus ease the microcontroller of the conversion itself.

# Chapter 10

# Front-end

The final activity of this master thesis is the front-end. This part is divided into

- Linting of newly developed RTL code
- Synthesis with DesignCompiler

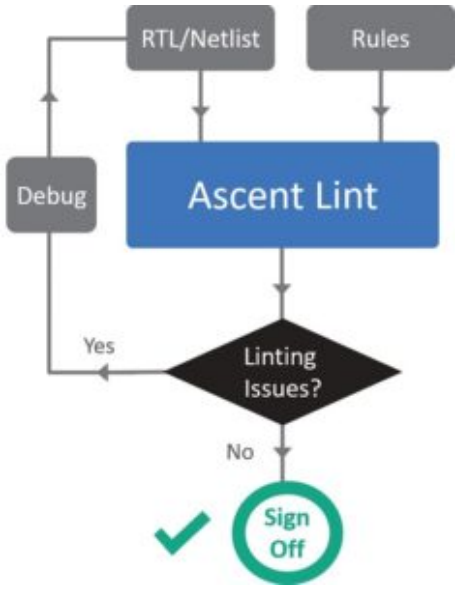
### 10.0.1 Linting

The linting activity has been performed with AscentLint [26], a tool from RealIntent. AscentLint takes as input the RTL to be analyzed alongside a set of predetermined rules, checks the RTL and produces a log with the part of the RTL code that do not match the input set of rules. With RealIntent it is possible to check

- coding style
- language construct usage
- synthesizable RTL

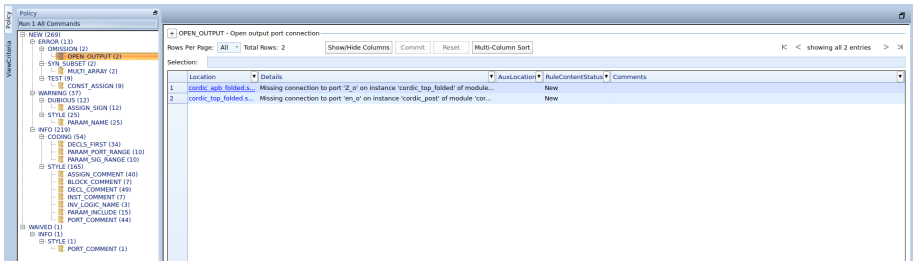
This activity, run at RTL stage, detect possible bugs in order to minimize the number of design iterations which could occur due to bug detection downstream in the flow. Figure 10.1 gives a visual representation of the RealIntent flow.

Figure 10.1: RealIntent flow



RealIntent has an integrated debug environment named iDebug that facilitates the error correction activity. Figure 10.2 gives a visual representation of iDebug.

Figure 10.2: iDebug

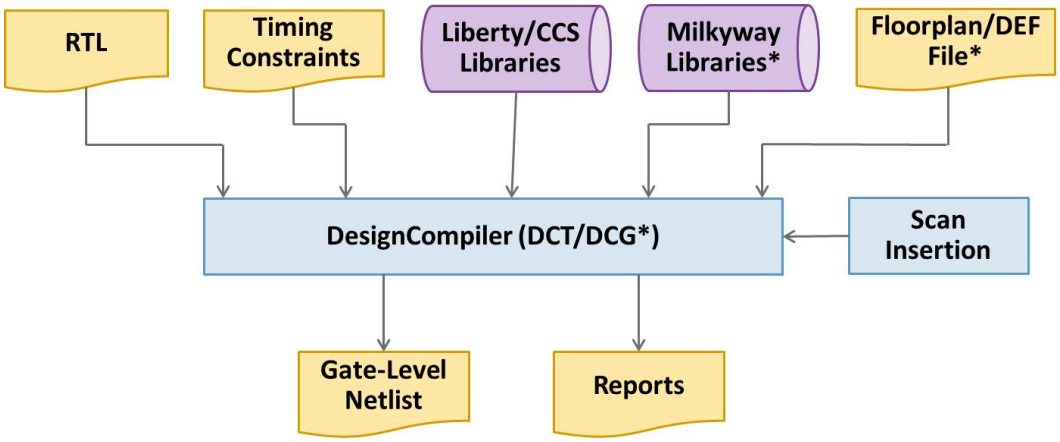


### 10.0.2 Synthesis

The synthesis step has been performed using Design Compiler, a tool from Synopsis; its purpose is to, starting from the RTL design, output the design's gate level netlist. The synthesis has been performed pointing to a Maxim'x propietary library of gates. As represented in figure 10.3 in order to get to a final gate level netlist Design Compiler needs the following inputs

- RTL description
- Timing contraints
- Liberty/CCS libraries

Figure 10.3: Design Compiler inputs/outputs




#### RTL description

The RTL collection of files given as input to Design Compiler are the files used during the MCU's simulation activity performed to validate both the microcontroller core and the peripherals integration step. In particular the collection of files comprise

- MCU's RTL files
- UART, SPI, I2C RTL files
- CORDIC RTL files

### Timing constraints

The timing contraints are applied to input and output ports, in particular

- Clock frequency, jitter and slope transition are specified
- Input delays relative to the system clock are specified
- Output delays relative to the system clock are specified

### Liberty/CCS libraries

Since the MCU's design is made of two RAM memories (data and instruction RAM), the liberty (.lib) files for both memories had to be generated. The liberty files give a characterization for timing and area for the memories; this characterization is necessary for Design Compiler in order to give an estimate for the design's performance and area parameters.

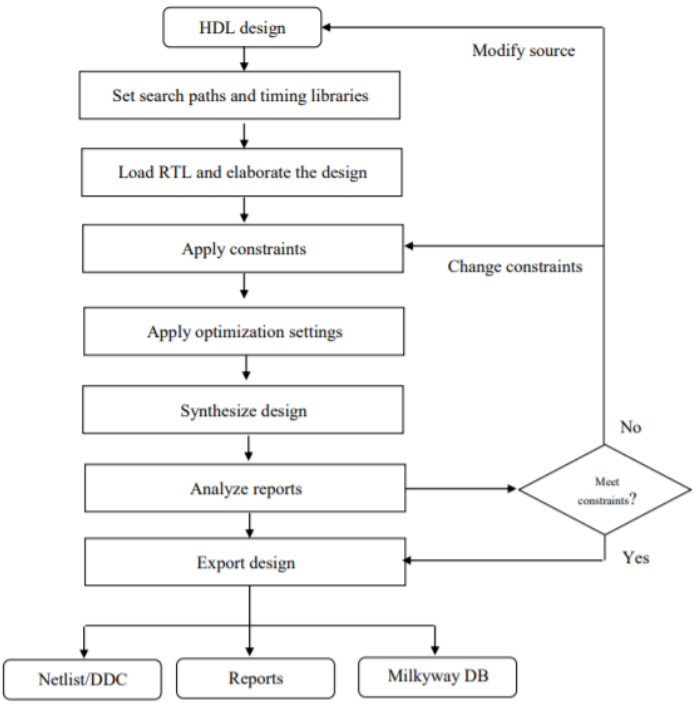
### Synthesis script

To perform the sythesis step an internal script has been provided. The following points are a rough summary of each step performed by the script

- Loading each design file (.sv, .lib) + analyze step to check the correctness of each input file
- Applying timing constraints
- Initial hierarchical syntesis
- Final flattened syntesis
- Fianl report annotation
- Final netlist annotation

Figure 10.4 gives a representation of a typical design compiler flow.

Figure 10.4: Design Compiler flow



### Reports

After the flattened synthesis step reports for area and timing are annotated. The final result of the synthesis step are

| frequency [MHz] | total area [$\mu m^2$] | comb area [$\mu m^2$] | non comb area [$\mu m^2$] | black boxes area [$\mu m^2$] |
|---|---|---|---|---|
| 25 | 3088125 | 163856 | 150139 | 2774130 |

Table 10.1: synthesis reports

A couple of points have to be noted for the results present in table 10.1; the aim of this preliminary synthesis step is to gauge the area occupation for the MCU. This is the reason why the maximum clock frequency is 25 MHz.

Looking at the area parameter is possible to note that around 90% of the area is taken by the black boxes area; the black boxes in this particular design are the two RAM memories. The size of the RAM memories has been chosen such that they would be big enouch to contain data and instruction of the most memory expensive test performed, which is the CORDIC precision test. The size used in this preliminary synthesis is not important since the area parameter of importance is relative to the MCU.

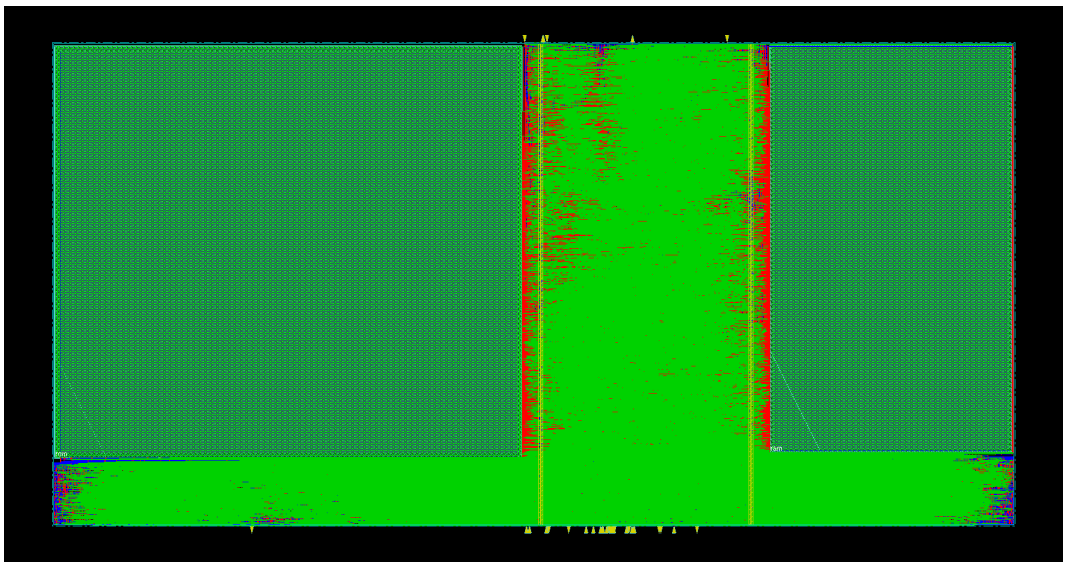
The total area of the MCU, both combinational and non combinational, amount to around 0.3 $mm^2$. The area of the MCU includes

- RISC-V microcontroller
- UART, SPI, I2C peripherals
- CORDIC peripheral

### Place and Route

A preliminary place and route has been performed by a Maxim's layout engineer. The result of the place and route process is

Figure 10.5: MCU place and route



# Chapter 11

# Conclusion

In conclusion the objectives set out in the abstract have been met, in particular

- Toolchain bring up for RISCV
- Benchmarking activity
- General purpose peripherals integration
- CORDIC peripheral development
- Front-end activity

Compared to a state machine designed for a specific application the RISCV IP microcontroller is a more flexible solution in a low area occupation. The benchmarking results confirm that the performances of the RISCV microcontroller (with the zero-riscy core) is not suitable for floating point or compute intensive DSP applications, rather for general purpose I/O applications.

The presence of two busess (AHB and APB) allows the development of application specific peripherals to make up for the lack in performances at the expense of additional area.

A stronger industrial adoption of the RISCV instruction set can only benefit the movement in terms of

- Availability of new and improved IDEs
- Availability of new and improved RISCV based microcontroller designs

Possible future developments for this project are

- Completing the front-end activity by performing STA
- Production of a test chip

# Appendix A

# Project sources

## A.1 Memories RTL model

### A.1.1 ROM RTL model

```
//fully synchronous rom model
//numByte is the number of Bytes of the memory
module rom #(parameter numByte = 512*1024)
            (input logic rom0_clk,
             input logic rom0_ce,
             input logic[($clog2(numByte)-2)-1:0] rom0_addr,
             output logic[31:0] rom0_rdata);

//1 MiB = 4 * 256 KiB => each memory location is made of 32 bits
logic [31:0] rom[(1<<(($clog2(numByte))-2))-1:0];

//rom initialization
initial $readmemb("file.mem", rom);

//read capability
//read the complete word if rom0_ce is high
always_ff @(posedge rom0_clk) begin
  if (rom0_ce)
    rom0_rdata <= rom[rom0_addr];
end

endmodule
```

### A.1.2 RAM RTL model

```
//fully synchronous ram model
//numByte is the number of Bytes of the memory
module ram #(parameter numByte = 128*1024)
            (input logic ram0_clk,
             input logic ram0_ce,
             input logic ram0_we,
             input logic[3:0] ram0_be,
             input logic[($clog2(numByte)-2)-1:0] ram0_addr,
             input logic[31:0] ram0_wdata,
             output logic[31:0] ram0_rdata);

//128 KiB = 4 * 32 KiB => each memory location is made of 32 bits
logic[31:0] ram[(1<<(($clog2(numByte))-2))-1:0];

//write capability
//a memory location is written if
//ram0_ce and ram0_we are high
//the corresponding bit in ram0_be is high
always_ff @(posedge ram0_clk) begin
  if (ram0_ce) begin
    if (ram0_we)
      if (ram0_be[0])
        ram[ram0_addr][7:0] <= ram0_wdata[7:0];
      if (ram0_be[1])
        ram[ram0_addr][15:8] <= ram0_wdata[15:8];
      if (ram0_be[2])
        ram[ram0_addr][23:16] <= ram0_wdata[23:16];
      if (ram0_be[3])
        ram[ram0_addr][31:24] <= ram0_wdata[31:24];
    end
end

//read capability
//read the complete word if ram0_ce is high
always_ff @(posedge ram0_clk) begin
  if (ram0_ce)
    ram0_rdata <= ram[ram0_addr];
end

endmodule
```

## A.2 State machine RTL model

```
//input to state machine
logic state_input_rot, state_input_vect, state_clear;

assign state_input_rot = PENABLE & PSEL & PWRITE & PWDATA[0] &
 (PADDR == 32'h00000000);
assign state_input_vect = PENABLE & PSEL & PWRITE & PWDATA[1] &
 (PADDR == 32'h00000000);
assign state_clear = PENABLE & PSEL & PWRITE & PWDATA[1] &
 (PADDR == 32'h00000004);

//state machine
parameter IDLE          = 4'b0000,
START_ROT               = 4'b0001,
START_VECT              = 4'b0010,
RUN                     = 4'b0011,
DONE                    = 4'b0100,
SKIP_ROT                = 4'b0101,
STOP_EN_VECT            = 4'b0110,
RUN_VECT                = 4'b0111,
CONV_VECT               = 4'b1000,
SKIP_VECT               = 4'b1001,
STOP_EN_ROT             = 4'b1010;

logic [3:0] state, next_state;
logic out_rot, out_vect, sample, done;

integer counter_s;
logic en_reg_s, step_s, conv_s;
logic operation_s;

always_comb //(state, state_input, state_clear)
begin
case(state)
IDLE :
begin
if(state_input_rot == 1'b1)
        next_state = START_ROT;
else if(state_input_vect == 1'b1)
        next_state = START_VECT;
else
        next_state = IDLE;
end
START_ROT :
begin
        next_state = SKIP_ROT;
end
SKIP_ROT :
begin
        next_state = RUN;
end
START_VECT :
begin
        next_state = SKIP_VECT;
end
SKIP_VECT :
begin
```

```
                next_state = RUN_VECT;
        end
        RUN :
        begin
        if(counter == DEPTH - 2)
                begin
                        next_state = STOP_EN_ROT;
                end
        else
        begin
                next_state = RUN;
        end
        end
        RUN_VECT :
        begin
                if(counter == DEPTH - 2)
                begin
                        next_state = STOP_EN_VECT;
                end
        else if(counter == 4 - 1)
        begin
                next_state = CONV_VECT;
        end
        else if(counter == 13 - 1)
        begin
                next_state = CONV_VECT;
        end
        else
        begin
                next_state = RUN_VECT;
        end
        end
        CONV_VECT :
        begin
                if(counter == DEPTH - 2)
                begin
                        next_state = STOP_EN_VECT;
                end
        else
        begin
                next_state = RUN_VECT;
        end
        end
        STOP_EN_VECT :
        begin
                next_state = DONE;
        end
        STOP_EN_ROT :
        begin
                next_state = DONE;
        end
        DONE :
        begin
        if(state_clear == 1'b1)
                next_state = IDLE;
        else
                next_state = DONE;
        end
```

```
default :
        next_state = IDLE;
endcase
end

always_ff @(posedge CLK or negedge RSTn)
begin
if (RSTn == 1'b0)
begin
        counter_s   <= 6'd0;
        out_rot     <= 1'b0;
        out_vect    <= 1'b0;
        sample      <= 1'b0;
        done        <= 1'b0;
        ////////////////
        en_reg_s    <= 1'b0;
        step_s      <= 1'b0;
        conv_s      <= 1'b0;
        operation_s <= 1'b0;
end
else
case(state)
IDLE :
begin
        counter_s   <= 6'd0;
        out_rot     <= 1'b0;
        out_vect    <= 1'b0;
        sample      <= 1'b0;
        done        <= 1'b0;
        ////////////////
        en_reg_s    <= 1'b0;
        step_s      <= 1'b0;
        conv_s      <= 1'b0;
        operation_s <= 1'b0;
end
START_ROT :
begin
        counter_s   <= 6'd0;
        out_rot     <= 1'b1;
        out_vect    <= 1'b0;
        sample      <= 1'b0;
        done        <= 1'b0;
        ////////////////
        en_reg_s    <= 1'b1;
        step_s      <= 1'b0;
        conv_s      <= 1'b0;
        operation_s <= 1'b0;
end
SKIP_ROT :
begin
        counter_s   <= 6'd0;
        out_rot     <= 1'b1;
        out_vect    <= 1'b0;
        sample      <= 1'b0;
        done        <= 1'b0;
        ////////////////
        en_reg_s    <= 1'b1;
        step_s      <= 1'b1;
```

```
            conv_s      <= 1'b0;
            operation_s <= 1'b0;
        end
        START_VECT :
        begin
            counter_s   <= 6'd0;
            out_rot     <= 1'b0;
            out_vect    <= 1'b1;
            sample      <= 1'b0;
            done        <= 1'b0;
            ////////////////
            en_reg_s    <= 1'b1;
            step_s      <= 1'b0;
            conv_s      <= 1'b0;
            operation_s <= 1'b1;
        end
        SKIP_VECT :
        begin
            counter_s   <= counter_s + 1;
            out_rot     <= 1'b0;
            out_vect    <= 1'b1;
            sample      <= 1'b0;
            done        <= 1'b0;
            ////////////////
            en_reg_s    <= 1'b1;
            step_s      <= 1'b1;
            conv_s      <= 1'b0;
            operation_s <= 1'b1;
        end
        RUN :
        begin
            counter_s   <= counter + 1;
            out_rot     <= 1'b0;
            out_vect    <= 1'b0;
            sample      <= 1'b0;
            done        <= 1'b0;
            ////////////////
            en_reg_s    <= 1'b1;
            step_s      <= 1'b1;
            conv_s      <= 1'b0;
            operation_s <= 1'b0;
        end
        RUN_VECT :
        begin
            counter_s   <= counter + 1;
            out_rot     <= 1'b0;
            out_vect    <= 1'b0;
            sample      <= 1'b0;
            done        <= 1'b0;
            ////////////////
            en_reg_s    <= 1'b1;
            step_s      <= 1'b1;
            conv_s      <= 1'b0;
            operation_s <= 1'b1;
        end
        CONV_VECT :
        begin
            counter_s   <= counter + 1;
```

```
            out_rot     <= 1'b0;
            out_vect    <= 1'b0;
            sample      <= 1'b0;
            done        <= 1'b0;
            ////////////////
            en_reg_s    <= 1'b1;
            step_s      <= 1'b1;
            conv_s      <= 1'b1;
            operation_s <= 1'b1;
        end
        STOP_EN_VECT :
        begin
            counter_s   <= counter + 1;
            out_rot     <= 1'b0;
            out_vect    <= 1'b0;
            sample      <= 1'b1;
            done        <= 1'b0;
            ////////////////
            en_reg_s    <= 1'b1;
            step_s      <= 1'b1;
            conv_s      <= 1'b0;
            operation_s <= 1'b1;
        end
        STOP_EN_ROT :
        begin
            counter_s   <= counter + 1;
            out_rot     <= 1'b0;
            out_vect    <= 1'b0;
            sample      <= 1'b1;
            done        <= 1'b0;
            ////////////////
            en_reg_s    <= 1'b1;
            step_s      <= 1'b1;
            conv_s      <= 1'b0;
            operation_s <= 1'b0;
        end
        DONE :
        begin
            counter_s   <= counter_s;
            out_rot     <= 1'b0;
            out_vect    <= 1'b0;
            sample      <= 1'b0;
            done        <= 1'b1;
            ////////////////
            en_reg_s    <= 1'b0;
            step_s      <= 1'b0;
            conv_s      <= 1'b0;
            operation_s <= 1'b0;
        end
        default :
        begin
            counter_s   <= 6'd0;
            out_rot     <= 1'b0;
            out_vect    <= 1'b0;
            sample      <= 1'b0;
            done        <= 1'b0;
            ////////////////
            en_reg_s    <= 1'b0;
```

```
        step_s      <= 1'b0;
        conv_s      <= 1'b0;
        operation_s <= 1'b0;
end
endcase
end

always_ff @(posedge CLK or negedge RSTn)
begin
if (RSTn == 1'b0)
        state <= IDLE;
else
        state <= next_state;
end

//output assignment
assign operation_o = operation_s;
assign en_reg = en_reg_s;
assign step = step_s;
assign conv = conv_s;
assign counter = counter_s;
```

## A.3 C code

### A.3.1 March C

```
#include "apb_driver.h"
#include <stdlib.h>
#include <stdbool.h>

int mem[16] = {0xFFFFFFFF,
               0x00000000,
               0xEEEEEEEE,
               0x11111111,
               0xDDDDDDDD,
               0x22222222,
               0xCCCCCCCC,
               0x33333333,
               0xBBBBBBBB,
               0x44444444,
               0xAAAAAAAA,
               0x55555555,
               0x99999999,
               0x66666666,
               0x88888888,
               0x77777777};

int memory[3053];

int zero;
int one;

int start_addr = SOC_PERIPHERALS_BASE_ADDR;

bool stop = false;

int
main(void)
{
    int address = start_addr;
    for (int i = 0; i < 8 && !stop; i++){
        zero = mem[2*i];
        one = mem[2*i+1];

        test_running = 1;
        test_outcome = 0;
      //<> w0
      for (int j = 0; j < 3053 && !stop; j++){
        memory[j] = zero;
      }
      //>r0,w1
      for (int t = 0; t < 3053 && !stop; t++){
        if(memory[t] != zero){
                stop = true;
                writeAPB(address,0x0000000A);
                writeAPB(address+4,t);
        }
        memory[t] = one;
      }
      //>r1,w0
```

```
        for (int f = 0; f < 3053 && !stop; f++){
          if(memory[f] != one){
                  stop = true;
              writeAPB(address,0x0000000B);
              writeAPB(address+4,f);
          }
          memory[f] = zero;
        }
        //<>r0
        for (int y = 0; y < 3053 && !stop; y++){
           if(memory[y] != zero){
              stop = true;
              writeAPB(address,0x0000000C);
              writeAPB(address+4,y);
           }
        }
        //<r0,w1
        for (int r = 3052; r >= 0 && !stop; r--){
             if(memory[r] != zero){
                    stop = true;
                    writeAPB(address,0x0000000D);
                    writeAPB(address+4,r);
             }
             memory[r] = one;
        }
        //<r1,w0
        for (int h = 3052; h >= 0 && !stop; h--){
             if(memory[h] != one){
                stop = true;
                writeAPB(address,0x0000000E);
                writeAPB(address+4,h);
             }
             memory[h] = zero;
        }
        //<>r0
        for (int n = 0; n < 3053 && !stop; n++){
             if(memory[n] != zero){
                stop = true;
                writeAPB(address,0x0000000F);
                writeAPB(address+4,n);
             }
        }

    }

    test_running = 0;
    if(stop == false){
        test_outcome = 1;
    }

    while(1){}
}
```

### A.3.2 APB random test

```
#include "apb_driver.h"
#include <stdlib.h>
#include <stdbool.h>

int start_addr = SOC_PERIPHERALS_BASE_ADDR;
int end_addr = SOC_PERIPHERALS_END_TEST_ADDR;

int value_vect[10];
int address_vect[10];

bool stop = false;

int
main(void)
{
  int num_iteration = 10;
  int num_extern_for = 10;
  int address = start_addr;
  int add;
  int temp;

  test_running = 1;
  test_outcome = 0;

  for (int t = 0; t < num_extern_for && !stop; t++) {
                for (int j = 0; j < num_iteration && !stop; j++){
                  temp = rand()%4294967295;
                  value_vect[j] = temp;
                  address_vect[j] = address;

                  writeAPB(address,temp);

                  add = rand()%(end_addr - start_addr);
                  add = add >> 2;
                  add = add << 2;
                  address = start_addr + add;
                }


                for (int i = 0; i < num_iteration && !stop; i++){
                  if (readAPB(address_vect[i]) != value_vect[i]){
                          stop = true;
                  }
                }
        }

  test_running = 0;

  if(stop == false) {
          test_outcome = 1;
  }

  while(1){}
}
```

### A.3.3 CORDIC precision test

```
#include <math.h>
#include "pulpino.h"
#include "cordic.h"

int incorrect = 0;

int main(void) {

float sin_math;
float cos_math;
float sin_cordic;
float cos_cordic;
float angle_float = 0.0;
float angle_radians;
float precision = 0.001;
WORD angle_word;

int data_sqrt = 2;
float cordic_sqrt;
float fw_sqrt;
float precision_sqrt;

test_check = 0x01;

for(int i = 0; i < 3600; i++){
        //sin/cos
        angle_radians = degrees_to_radians(angle_float);
        sin_math = sin(angle_radians);
        cos_math = cos(angle_radians);

        angle_word = cordic_32_angle(angle_float);
        sin_cordic = cordic_16_sin(angle_word);
        cos_cordic = cordic_cos_half_word_read();
        cos_cordic = cos_cordic / cordic_16_div;

        if((sin_cordic >  sin_math + precision) ||
        (sin_cordic < sin_math - precision)){
                incorrect = 1;
                test_check = 0x00;
        }

        if((cos_cordic >  cos_math + precision) ||
        (cos_cordic < cos_math - precision)){
                incorrect = 1;
                test_check = 0x00;
        }

        angle_float += 0.1;

        //sqrt
        cordic_sqrt = cordic_sqrt_half_word(data_sqrt);
        fw_sqrt = sqrt(data_sqrt);

        precision_sqrt = fw_sqrt/300;

        if((cordic_sqrt >  fw_sqrt + precision_sqrt) ||
```

```
        (cordic_sqrt < fw_sqrt - precision_sqrt)){
                incorrect = 1;
                test_check = 0x00;
        }
        data_sqrt += 1;
}

if(incorrect){
        test_check = 0x00;
} else{
    test_check = 0x02;
}
}
```

### A.3.4 I2C master write - master code

```
////////////////////////////////////////////////
/////////////main.c code//////////////////////
////////////////////////////////////////////////
test_check = 0x01;

i2c_enable();
i2c_master_mode_enable();
i2c_set_timeout(0x0400);
i2c_clock_low(0x0001);
i2c_clock_high(0x0001);
////////////////////////////////////////////////
/////////////interrupt_0.c code//////////////////
////////////////////////////////////////////////
BYTE operation;
lfsr = lfsr_16(lfsr, mask);
if(lfsr != 0xACE1){
        operation = lfsr & 0x0007;
        switch(operation)
        {
        case 1 : i2c_master_write_byte(0x01, 0x01, (lfsr & 0x00ff));
                 break;
        case 2 : i2c_master_write_half_word(0x01, 0x01, (lfsr & 0x00ff),
                 ((lfsr & 0x01fe) >> 1));
                 break;
        case 3 : i2c_master_write_three_bytes(0x01, 0x01, (lfsr & 0x00ff),
                 ((lfsr & 0x01fe) >> 1), ((lfsr & 0x03fc) >> 2));
                 break;
        case 4 : i2c_master_write_word(0x01, 0x01, (lfsr & 0x00ff),
                 ((lfsr & 0x01fe) >> 1), ((lfsr & 0x03fc) >> 2),
                 ((lfsr & 0x07f8) >> 3));
                 break;
        case 5 : i2c_master_write_five_bytes(0x01, 0x01, (lfsr & 0x00ff),
                 ((lfsr & 0x01fe) >> 1), ((lfsr & 0x03fc) >> 2),
                 ((lfsr & 0x07f8) >> 3), ((lfsr & 0x0ff0) >> 4));
                 break;
        case 6 : i2c_master_write_six_bytes(0x01, 0x01, (lfsr & 0x00ff),
                 ((lfsr & 0x01fe) >> 1), ((lfsr & 0x03fc) >> 2),
                 ((lfsr & 0x07f8) >> 3), ((lfsr & 0x0ff0) >> 4),
                 ((lfsr & 0x1fe0) >> 5));
                 break;
        default : i2c_master_write_six_bytes(0x01, 0x01, (lfsr & 0x00ff),
                 ((lfsr & 0x01fe) >> 1), ((lfsr & 0x03fc) >> 2),
                 ((lfsr & 0x07f8) >> 3), ((lfsr & 0x0ff0) >> 4),
                 ((lfsr & 0x1fe0) >> 5));
                 break;
        }
} else {
        test_check = 0x02;
}
////////////////////////////////////////////////
/////////////interrupt_1.c code//////////////////
////////////////////////////////////////////////
test_check = 0x00;
```

### A.3.5 I2C master write - slave code

```
////////////////////////////////////////////////
/////////////main.c code////////////////////////
////////////////////////////////////////////////
i2c_enable();
i2c_master_mode_disable();
i2c_slave0_address(0x01);
i2c_slave0_address_enabled();
i2c_clock_low(0x0001);
i2c_clock_high(0x0001);

i2c_stop_irq_enabled();

for(int i = 0; i < 1000;  i++){}

test_check ^= 0x01;
test_check ^= 0x01;
////////////////////////////////////////////////
/////////////interrupt_i2c.c code/////////////////
////////////////////////////////////////////////
BYTE test0;
BYTE operation;

i2c_clear_irq_stop();

lfsr = lfsr_16(lfsr, mask);

operation = i2c_rx_fifo_bytes_count();

for(int i = 0; i < operation; i++){
        test0 = i2c_read_data();
        switch(i)
        {
                case 0 : if(test0 != 0x01){
                        test_check ^= 0x02;
                }
                break;
                case 1 : if(test0 != (lfsr & 0x00ff)){
                        test_check ^= 0x02;
                }
                break;
                case 2 : if(test0 != ((lfsr & 0x01fe) >> 1)){
                        test_check ^= 0x02;
                }
                break;
                case 3 : if(test0 != ((lfsr & 0x03fc) >> 2)){
                        test_check ^= 0x02;
                }
                break;
                case 4 : if(test0 != ((lfsr & 0x07f8) >> 3)){
                        test_check ^= 0x02;
                }
                break;
                case 5 : if(test0 != ((lfsr & 0x0ff0) >> 4)){
                        test_check ^= 0x02;
                }
                break;
```

```
                case 6 : if(test0 != ((lfsr & 0x1fe0) >> 5)){
                        test_check ^= 0x02;
                }
                break;
        }
}

i2c_rx_fifo_flush();
test_check ^= 0x01;
test_check ^= 0x01;

IER = 0xFFFFFFFF;
//__enable_irq();
}
```

### A.3.6 I2C master read - master code

```
////////////////////////////////////////////////
/////////////main.c code//////////////////////
////////////////////////////////////////////////
i2c_enable();
i2c_master_mode_enable();
i2c_set_timeout(0x0400);
i2c_clock_low(0x0001);
i2c_clock_high(0x0001);

i2c_set_rx_bytes_count(0x01);
i2c_stop_irq_enabled();

test_check = 0x01;
////////////////////////////////////////////////
/////////////interrupt_i2c.c code/////////////////
////////////////////////////////////////////////

BYTE test0;
BYTE test1;
BYTE test2;
BYTE test3;
BYTE test4;
BYTE test5;
BYTE test6;
BYTE test7;

i2c_clear_irq_stop();

switch(op)
{
        case 0 : test0 = i2c_read_data();
        if(test0 != (lfsr & 0x00ff)){
                test_check = 0x00;
        }
        break;
        case 1 : test0 = i2c_read_data();
        test1 = i2c_read_data();
        if(test0 != (lfsr & 0x00ff) | test1 != ((lfsr & 0x01fe) >> 1)){
                test_check = 0x00;
        }
        break;
        case 2 : test0 = i2c_read_data();
        test1 = i2c_read_data();
        test2 = i2c_read_data();
        if(test0 != (lfsr & 0x00ff) | test1 != ((lfsr & 0x01fe) >> 1) |
        test2 != ((lfsr & 0x03fc) >> 2)){
                test_check = 0x00;
        }
        break;
        case 3 : test0 = i2c_read_data();
        test1 = i2c_read_data();
        test2 = i2c_read_data();
        test3 = i2c_read_data();
        if(test0 != (lfsr & 0x00ff) | test1 != ((lfsr & 0x01fe) >> 1) |
        test2 != ((lfsr & 0x03fc) >> 2) | test3 != ((lfsr & 0x07f8) >> 3)){
                test_check = 0x00;
```

```
}
break;
case 4 : test0 = i2c_read_data();
test1 = i2c_read_data();
test2 = i2c_read_data();
test3 = i2c_read_data();
test4 = i2c_read_data();
if(test0 != (lfsr & 0x00ff) | test1 != ((lfsr & 0x01fe) >> 1) |
test2 != ((lfsr & 0x03fc) >> 2) | test3 != ((lfsr & 0x07f8) >> 3) |
test4 != ((lfsr & 0x0ff0) >> 4)){
        test_check = 0x00;
}
break;
case 5 : test0 = i2c_read_data();
test1 = i2c_read_data();
test2 = i2c_read_data();
test3 = i2c_read_data();
test4 = i2c_read_data();
test5 = i2c_read_data();
if(test0 != (lfsr & 0x00ff) | test1 != ((lfsr & 0x01fe) >> 1) |
test2 != ((lfsr & 0x03fc) >> 2) | test3 != ((lfsr & 0x07f8) >> 3) |
test4 != ((lfsr & 0x0ff0) >> 4) | test5 != ((lfsr & 0x1fe0) >> 5)){
        test_check = 0x00;
}
break;
case 6 : test0 = i2c_read_data();
test1 = i2c_read_data();
test2 = i2c_read_data();
test3 = i2c_read_data();
test4 = i2c_read_data();
test5 = i2c_read_data();
test6 = i2c_read_data();
if(test0 != (lfsr & 0x00ff) | test1 != ((lfsr & 0x01fe) >> 1) |
test2 != ((lfsr & 0x03fc) >> 2) | test3 != ((lfsr & 0x07f8) >> 3) |
test4 != ((lfsr & 0x0ff0) >> 4) | test5 != ((lfsr & 0x1fe0) >> 5) |
test6 != ((lfsr & 0x3fc0) >> 6)){
        test_check = 0x00;
}
break;
case 7 : test0 = i2c_read_data();
test1 = i2c_read_data();
test2 = i2c_read_data();
test3 = i2c_read_data();
test4 = i2c_read_data();
test5 = i2c_read_data();
test6 = i2c_read_data();
test7 = i2c_read_data();
if(test0 != (lfsr & 0x00ff) | test1 != ((lfsr & 0x01fe) >> 1) |
test2 != ((lfsr & 0x03fc) >> 2) | test3 != ((lfsr & 0x07f8) >> 3) |
test4 != ((lfsr & 0x0ff0) >> 4) | test5 != ((lfsr & 0x1fe0) >> 5) |
test6 != ((lfsr & 0x3fc0) >> 6) | test7 != ((lfsr & 0x7f80) >> 7)){
        test_check = 0x00;
}
break;
default : test0 = i2c_read_data();
test1 = i2c_read_data();
test2 = i2c_read_data();
test3 = i2c_read_data();
```

```
        if(test0 != (lfsr & 0x00ff) | test1 != ((lfsr & 0x01fe) >> 1) |
        test2 != ((lfsr & 0x03fc) >> 2) | test3 != ((lfsr & 0x07f8) >> 3)){
                test_check = 0x00;
        }
        break;
}


switch(next_op)
{
        case 0 : i2c_set_rx_bytes_count(0x01);
        break;
        case 1 : i2c_set_rx_bytes_count(0x02);
        break;
        case 2 : i2c_set_rx_bytes_count(0x03);
        break;
        case 3 : i2c_set_rx_bytes_count(0x04);
        break;
        case 4 : i2c_set_rx_bytes_count(0x05);
        break;
        case 5 : i2c_set_rx_bytes_count(0x06);
        break;
        case 6 : i2c_set_rx_bytes_count(0x07);
        break;
        case 7 : i2c_set_rx_bytes_count(0x08);
        break;
        default : i2c_set_rx_bytes_count(0x03);
        break;
}

test_check ^= 0x04;
test_check ^= 0x04;

////////////////////////////////////////////////
////////////interrupt_0.c code//////////////////
////////////////////////////////////////////////
i2c_master_read(0x01, 0x02);
lfsr = lfsr_16(lfsr,mask);
lfsr_next = lfsr_16(lfsr_next,mask);
op = (lfsr & 0x0007);
next_op = (lfsr_next & 0x0007);

//stop condition
if(lfsr == 0xACE1){
        test_check = 0x02;
}
```

### A.3.7 I2C master read - slave code

```
////////////////////////////////////////////////
/////////////main.c code//////////////////////
////////////////////////////////////////////////
i2c_enable();
i2c_master_mode_disable();
i2c_slave0_address(0x01);
i2c_slave0_address_enabled();
i2c_clock_low(0x0001);
i2c_clock_high(0x0001);

i2c_slave_read_address_match_irq_enabled();

for(int i = 0; i < 300; i++){}
test_check ^= 0x01;
test_check ^= 0x01;
////////////////////////////////////////////////
/////////////interrupt_i2c.c code/////////////
////////////////////////////////////////////////

BYTE op;

i2c_clear_irq_slave_read_address_match();

lfsr = lfsr_16(lfsr,mask);
op = (lfsr & 0x0007);

switch(op)
{
        case 0 : i2c_write_data((lfsr & 0x00ff));
        break;
        case 1 : i2c_write_data((lfsr & 0x00ff));
        i2c_write_data((lfsr & 0x01fe) >> 1);
        break;
        case 2 : i2c_write_data((lfsr & 0x00ff));
        i2c_write_data((lfsr & 0x01fe) >> 1);
        i2c_write_data((lfsr & 0x03fc) >> 2);
        break;
        case 3 : i2c_write_data((lfsr & 0x00ff));
        i2c_write_data((lfsr & 0x01fe) >> 1);
        i2c_write_data((lfsr & 0x03fc) >> 2);
        i2c_write_data((lfsr & 0x07f8) >> 3);
        break;
        case 4 : i2c_write_data((lfsr & 0x00ff));
        i2c_write_data((lfsr & 0x01fe) >> 1);
        i2c_write_data((lfsr & 0x03fc) >> 2);
        i2c_write_data((lfsr & 0x07f8) >> 3);
        i2c_write_data((lfsr & 0x0ff0) >> 4);
        break;
        case 5 : i2c_write_data((lfsr & 0x00ff));
        i2c_write_data((lfsr & 0x01fe) >> 1);
        i2c_write_data((lfsr & 0x03fc) >> 2);
        i2c_write_data((lfsr & 0x07f8) >> 3);
        i2c_write_data((lfsr & 0x0ff0) >> 4);
        i2c_write_data((lfsr & 0x1fe0) >> 5);
        break;
        case 6 : i2c_write_data((lfsr & 0x00ff));
```

```
        i2c_write_data((lfsr & 0x01fe) >> 1);
        i2c_write_data((lfsr & 0x03fc) >> 2);
        i2c_write_data((lfsr & 0x07f8) >> 3);
        i2c_write_data((lfsr & 0x0ff0) >> 4);
        i2c_write_data((lfsr & 0x1fe0) >> 5);
        i2c_write_data((lfsr & 0x3fc0) >> 6);
        break;
        case 7 : i2c_write_data((lfsr & 0x00ff));
        i2c_write_data((lfsr & 0x01fe) >> 1);
        i2c_write_data((lfsr & 0x03fc) >> 2);
        i2c_write_data((lfsr & 0x07f8) >> 3);
        i2c_write_data((lfsr & 0x0ff0) >> 4);
        i2c_write_data((lfsr & 0x1fe0) >> 5);
        i2c_write_data((lfsr & 0x3fc0) >> 6);
        i2c_write_data((lfsr & 0x7f80) >> 7);
        break;
        default : i2c_write_data((lfsr & 0x00ff));
        i2c_write_data((lfsr & 0x01fe) >> 1);
        i2c_write_data((lfsr & 0x03fc) >> 2);
        i2c_write_data((lfsr & 0x07f8) >> 3);
        break;
}

////////////////////////////////////////////////
/////////////interrupt_0.c code/////////////////
////////////////////////////////////////////////
test_check ^= 0x01;
test_check ^= 0x01;
```

### A.3.8 UART - master code

```
////////////////////////////////////////////////
/////////////main.c code//////////////////////
////////////////////////////////////////////////
test_check = 0x01;

set_baudrate_divisor(UART_DIV);
osr_zero_set();
osr_one_clear();
osr_two_set();
uart_parity_config(1,0,1);
uart_char_size(3);
uart_stop_bit(0);
uart_baud_clk_enable(1);
while(!uart_baud_clk_ready()){}
////////////////////////////////////////////////
/////////////interrupt_0.c code//////////////////
////////////////////////////////////////////////
int operation;

lfsr = lfsr_16(lfsr, mask);
operation = (lfsr & 0x0007);

switch(operation)
{
        case 0 : if(lfsr != 0xACE1){
                 send_uart(lfsr & 0x00ff);
        } else {
                 test_check = 0x02;
        }
        break;
        case 1 : if(lfsr != 0xACE1){
                 send_uart(lfsr & 0x00ff);
                 send_uart((lfsr & 0x01fe) >> 1);
        } else {
                 test_check = 0x02;
        }
        break;
        case 2 : if(lfsr != 0xACE1){
                 send_uart(lfsr & 0x00ff);
                 send_uart((lfsr & 0x01fe) >> 1);
                 send_uart((lfsr & 0x03fc) >> 2);
        } else {
                 test_check = 0x02;
        }
        break;
        case 3 : if(lfsr != 0xACE1){
                 send_uart(lfsr & 0x00ff);
                 send_uart((lfsr & 0x01fe) >> 1);
                 send_uart((lfsr & 0x03fc) >> 2);
                 send_uart((lfsr & 0x07f8) >> 3);
        } else {
                 test_check = 0x02;
        }
        break;
        case 4 : if(lfsr != 0xACE1){
                 send_uart(lfsr & 0x00ff);
```

```
                send_uart((lfsr & 0x01fe) >> 1);
                send_uart((lfsr & 0x03fc) >> 2);
                send_uart((lfsr & 0x07f8) >> 3);
                send_uart((lfsr & 0x0ff0) >> 4);
        } else {
                test_check = 0x02;
        }
        break;
        case 5 : if(lfsr != 0xACE1){
                send_uart(lfsr & 0x00ff);
                send_uart((lfsr & 0x01fe) >> 1);
                send_uart((lfsr & 0x03fc) >> 2);
                send_uart((lfsr & 0x07f8) >> 3);
                send_uart((lfsr & 0x0ff0) >> 4);
                send_uart((lfsr & 0x1fe0) >> 5);
        } else {
                test_check = 0x02;
        }
        break;
        case 6 : if(lfsr != 0xACE1){
                send_uart(lfsr & 0x00ff);
                send_uart((lfsr & 0x01fe) >> 1);
                send_uart((lfsr & 0x03fc) >> 2);
                send_uart((lfsr & 0x07f8) >> 3);
                send_uart((lfsr & 0x0ff0) >> 4);
                send_uart((lfsr & 0x1fe0) >> 5);
                send_uart((lfsr & 0x3fc0) >> 6);
        } else {
                test_check = 0x02;
        }
        break;
        case 7 : if(lfsr != 0xACE1){
                send_uart(lfsr & 0x00ff);
                send_uart((lfsr & 0x01fe) >> 1);
                send_uart((lfsr & 0x03fc) >> 2);
                send_uart((lfsr & 0x07f8) >> 3);
                send_uart((lfsr & 0x0ff0) >> 4);
                send_uart((lfsr & 0x1fe0) >> 5);
                send_uart((lfsr & 0x3fc0) >> 6);
                send_uart((lfsr & 0x7f80) >> 7);
        } else {
                test_check = 0x02;
        }
        break;
        default : if(lfsr != 0xACE1){
                send_uart(lfsr & 0x00ff);
                send_uart((lfsr & 0x01fe) >> 1);
                send_uart((lfsr & 0x03fc) >> 2);
                send_uart((lfsr & 0x07f8) >> 3);
        } else {
                test_check = 0x02;
        }
        break;
}

while(!tx_fifo_empty()){}

while(tx_fifo_busy()){}
```

```
test_check ^= 0x04;
test_check ^= 0x04;
////////////////////////////////////////////////
/////////////interrupt_1.c code/////////////////
////////////////////////////////////////////////
test_check = 0x00;
```

### A.3.9 UART - slave code

```
////////////////////////////////////////////////
/////////////main.c code//////////////////////
////////////////////////////////////////////////
set_baudrate_divisor(UART_DIV);
osr_zero_set();
osr_one_clear();
osr_two_set();
uart_parity_config(1,0,1);
uart_char_size(3);
uart_stop_bit(0);
uart_baud_clk_enable(1);
while(!uart_baud_clk_ready()){}

for(int i = 0; i < 1000; i++){}
test_check ^= 0x01;
test_check ^= 0x01;
////////////////////////////////////////////////
/////////////interrupt_0.c code//////////////////
////////////////////////////////////////////////
BYTE data;

BYTE bytes;

lfsr = lfsr_16(lfsr, mask);

bytes = rx_fifo_bytes();

for(int i = 0; i < bytes; i++){
        data = recieve_uart();
        switch(i)
        {
        case 0 : if(data != (lfsr & 0x00ff)){
                                test_check ^= 0x02;
                        }
                        break;
        case 1 : if(data != ((lfsr & 0x01fe) >> 1)){
                                test_check ^= 0x02;
                        }
                        break;
        case 2 : if(data != ((lfsr & 0x03fc) >> 2)){
                                test_check ^= 0x02;
                        }
                        break;
        case 3 : if(data != ((lfsr & 0x07f8) >> 3)){
                                test_check ^= 0x02;
                        }
                        break;
        case 4 : if(data != ((lfsr & 0x0ff0) >> 4)){
                                test_check ^= 0x02;
                        }
                        break;
        case 5 : if(data != ((lfsr & 0x1fe0) >> 5))
                                test_check ^= 0x02;
                        }
                        break;
        case 6 : if(data != ((lfsr & 0x3fc0) >> 6)){
```

```
                                test_check ^= 0x02;
                        }
                        break;
        case 7 : if(data != ((lfsr & 0x7f80) >> 7)){
                                test_check ^= 0x02;
                        }
                        break;
        }
}

rx_flush();

test_check ^= 0x01;
test_check ^= 0x01;
```

### A.3.10 SPI word mode - master code

```
////////////////////////////////////////////////
/////////////main.c code//////////////////////
////////////////////////////////////////////////
test_check ^= 0x01;
spi_setup();
////////////////////////////////////////////////
/////////////interrupt_0.c code/////////////////
////////////////////////////////////////////////
int num_tx;

lfsr = lfsr_16(lfsr, mask);

if(lfsr == 0xACE1){
        test_check = 0x02;
}

WORD lfsr2, lfsr3;

lfsr2 = lfsr;
lfsr3 = lfsr;

lfsr2 = (lfsr2 << 16) | lfsr3;

num_tx = (lfsr & 0x0007);

switch(num_tx)
{
case 0 : spi_tx_num_char(4);
                spi_ss_sel(1);
                spi_push_word(lfsr2);
                spi_start_transaction();
                while(spi_status()){}
                spi_ss_desel(1);
                break;
case 1 : spi_tx_num_char(8);
                spi_ss_sel(1);
                spi_push_half_word(lfsr2);
                spi_push_half_word((lfsr2 >> 1));
                spi_start_transaction();
                while(spi_status()){}
                spi_ss_desel(1);
                break;
case 2 : spi_tx_num_char(12);
                spi_ss_sel(1);
                spi_push_half_word(lfsr2);
                spi_push_half_word((lfsr2 >> 1));
                spi_push_half_word((lfsr2 >> 2));
                spi_start_transaction();
                while(spi_status()){}
                spi_ss_desel(1);
                break;
case 3 : spi_tx_num_char(16);
                spi_ss_sel(1);
                spi_push_half_word(lfsr2);
                spi_push_half_word((lfsr2 >> 1));
                spi_push_half_word((lfsr2 >> 2));
```

```
                spi_push_half_word((lfsr2 >> 3));
                spi_start_transaction();
                while(spi_status()){}
                spi_ss_desel(1);
                break;
case 4 : spi_tx_num_char(20);
                spi_ss_sel(1);
                spi_push_half_word(lfsr2);
                spi_push_half_word((lfsr2 >> 1));
                spi_push_half_word((lfsr2 >> 2));
                spi_push_half_word((lfsr2 >> 3));
                spi_push_half_word((lfsr2 >> 4));
                spi_start_transaction();
                while(spi_status()){}
                spi_ss_desel(1);
                break;
case 5 : spi_tx_num_char(24);
                spi_ss_sel(1);
                spi_push_half_word(lfsr2);
                spi_push_half_word((lfsr2 >> 1));
                spi_push_half_word((lfsr2 >> 2));
                spi_push_half_word((lfsr2 >> 3));
                spi_push_half_word((lfsr2 >> 4));
                spi_push_half_word((lfsr2 >> 5));
                spi_start_transaction();
                while(spi_status()){}
                spi_ss_desel(1);
                break;
case 6 : spi_tx_num_char(28);
                spi_ss_sel(1);
                spi_push_half_word(lfsr2);
                spi_push_half_word((lfsr2 >> 1));
                spi_push_half_word((lfsr2 >> 2));
                spi_push_half_word((lfsr2 >> 3));
                spi_push_half_word((lfsr2 >> 4));
                spi_push_half_word((lfsr2 >> 5));
                spi_push_half_word((lfsr2 >> 6));
                spi_start_transaction();
                while(spi_status()){}
                spi_ss_desel(1);
                break;
case 7 : spi_tx_num_char(32);
                spi_ss_sel(1);
                spi_push_half_word(lfsr2);
                spi_push_half_word((lfsr2 >> 1));
                spi_push_half_word((lfsr2 >> 2));
                spi_push_half_word((lfsr2 >> 3));
                spi_push_half_word((lfsr2 >> 4));
                spi_push_half_word((lfsr2 >> 5));
                spi_push_half_word((lfsr2 >> 6));
                spi_push_half_word((lfsr2 >> 7));
                spi_start_transaction();
                while(spi_status()){}
                spi_ss_desel(1);
                break;
}
////////////////////////////////////////////////
/////////////interrupt_1.c code/////////////////
```

```
//////////////////////////////////////////////
test_check = 0x00;
```

### A.3.11 SPI word mode - slave code

```
////////////////////////////////////////////////
/////////////main.c code//////////////////////
////////////////////////////////////////////////
spi_setup();
spi_slave_mode();
spi_enable_rx_fifo();

spi_ss_deasserted_enable_irq();

for(int i = 0; i < 1000; i++){}
test_check ^= 0x01;
test_check ^= 0x01;
////////////////////////////////////////////////
/////////////interrupt_spi.c code/////////////
////////////////////////////////////////////////
WORD data;
BYTE bytes;

WORD lfsr2, lfsr3;

spi_clear_irq_ss_deasserted();
lfsr = lfsr_16(lfsr, mask);
bytes = spi_rx_fifo_cnt();

lfsr2 = lfsr;
lfsr3 = lfsr;

lfsr2 = (lfsr2 << 16) | lfsr3;


for(int i = 0; i < bytes/4; i++){
data = spi_pull_word();
switch(i)
{
case 0 :    if(data != lfsr2){
                                    test_check = 0x02;
                            }
                            break;
case 1 :   if(data != (lfsr2 >> 1)){
                                    test_check = 0x02;
                            }
                            break;
case 2 :   if(data != (lfsr2 >> 2)){
                                    test_check = 0x02;
                            }
                            break;
case 3 :         if(data != (lfsr2 >> 3)){
                                    test_check = 0x02;
                            }
                            break;
case 4 :         if(data != (lfsr2 >> 4)){
                                    test_check = 0x02;
                            }
                            break;
case 5 :         if(data != (lfsr2 >> 5)){
                                    test_check = 0x02;
```

```
                        }
                        break;
case 6 :        if(data != (lfsr2 >> 6)){
                                test_check = 0x02;
                        }
                        break;
case 7 :        if(data != (lfsr2 >> 7)){
                                test_check = 0x02;
                        }
                        break;

spi_clear_rx_fifo();

test_check ^= 0x01;
test_check ^= 0x01;
```

# Bibliography


[1] URL: https://en.wikipedia.org/wiki/RISC-V.

[2] URL: https://riscv.org/membership/members/.

[3] URL: https://www.maximintegrated.com/en.html.

[4] URL: https://pulp-platform.org/.

[5] URL: https://www.eclipse.org/.

[6] URL: https://gnu-mcu-eclipse.github.io/downloads/.

[7] URL: https://reference.digilentinc.com/reference/programmable-logic/arty-a7/start.

[8] URL: https://en.wikipedia.org/wiki/JTAG.

[9] URL: https://en.wikipedia.org/wiki/Advanced_Microcontroller_Bus_Architecture.

[10] URL: https://developer.arm.com/documentation/ihi0024/c/.

[11] URL: https://pulp-platform.org/docs/pulpino_datasheet.pdf.

[12] URL: https://en.wikipedia.org/wiki/Integrated_development_environment.

[13] URL: https://gnu-mcu-eclipse.github.io/toolchain/riscv/.

[14] URL: https://en.wikipedia.org/wiki/Intel_HEX.

[15] URL: https://en.wikipedia.org/wiki/Dhrystone.

[16] URL: https://github.com/embench/embench-iot.

[17] URL: https://ibex-core.readthedocs.io/en/latest/index.html.

[18] URL: http://www.vlsiip.com/soc/soc_0003.html.

[19] URL: https://www.iar.com/iar-embedded-workbench/#!?architecture=RISC-V.

[20] URL: https://forums.xilinx.com/t5/Versal-and-UltraScale/Comparing-ASIC-gate-equivalent-with-XU-LUTs/m-p/893795?advanced=false&collapse_discussion=true&search_type=thread.

[21] URL: https://ttssh2.osdn.jp/index.html.en.

[22] URL: https://reference.digilentinc.com/reference/instrumentation/analog-discovery-2/start.

[23] URL: https://en.wikipedia.org/wiki/Linear-feedback_shift_register.

[24] URL: https://it.mathworks.com/help/fixedpoint/ug/compute-square-root-using-cordic.html;jsessionid=cd29f8c21792e49ef58535c8b3fe.

[25] URL: https://en.wikipedia.org/wiki/Binary_scaling.

[26] URL: https://www.realintent.com/rtl-linting-ascent-lint/.