AETHER SYSTEM-ON-CHIP
USER GUIDE

AETHER
SOC

VERSION: v1.0.0

TABLE OF CONTENTS

1. SUMMARY

Aether SoC is an open-source RISC-V microcontroller. The design aims to provide a lightweight and easily extensible platform for embedded system development and applications.

The system is built around a mutli-cycle RV32I processor, implementing the base RISC-V integer instruction set. It follows a Harvard memory architecture, meaning instruction and data memory are separated. The peripheral bus is a custom memory-mapped I/O (MMIO) Bus where all the peripheral connections along with the data memory (RAM) are integrated on the same bus, separated by address regions.

To support interaction with the external world, Aether SoC includes several configurable peripheral modules, including:

- 8 kB of RAM
- A UART module with 4 selectable baud rates
- An SPI module, configurable as either master or slave, master mode supporting up to 4 slaves
- 12 General-Purpose I/O (GPIO) pins, all individually configurable as input or output.
- 4 built-in LEDs and switches.
- A PWM-controlled output channels.
- An independent timer, designed primarily for delay generation and basic timing tasks.

The system operates at 50 MHz and has been fully implemented and tested on Digilent Zybo Z7 development board (Xilinx Zynq-7020 SoC). In addition, the design was successfully fabricated using the SkyWater 130 nm (Sky130 PDK) process within the OpenLane physical design flow.

Aether SoC has been developed in a vision with minimalist, modular and configurable design, specifically targeting students and beginners.

Please note that this project is not intended for industrial or safety-critical use. Rather, it serves as a beginner level engineering project, which will continue to evolve and improve through iterative development.

Yagiz Yagmur

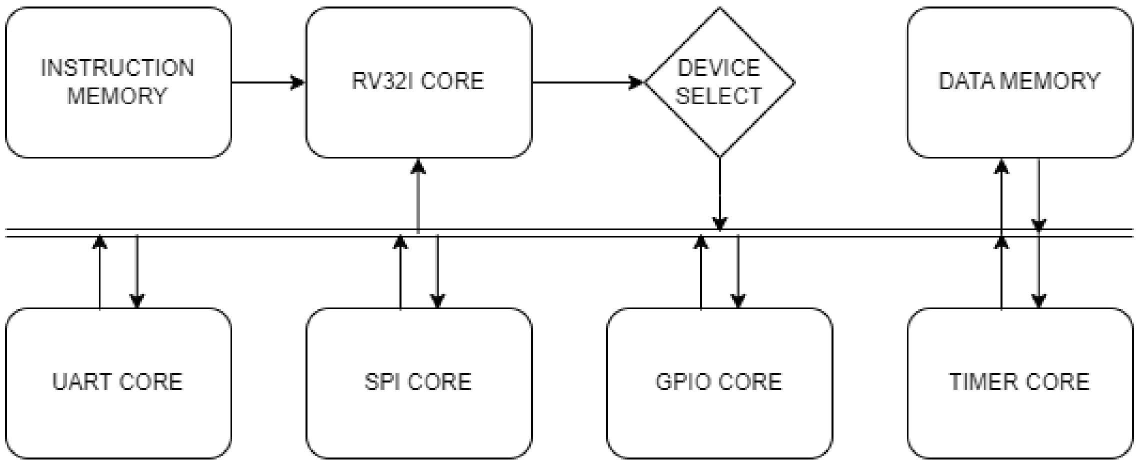
Electronics and Communication Engineer

MSc at Politecnico di Milano

Mail: yagiz.yagmur123@hotmail.com

Github: github.com/yagyag12

2. FEATURES



Instruction Set: RV32I ISA

Clock Frequency: 50 MHz

Memory Size: 4 kB Instruction Memory & 8 kB RAM

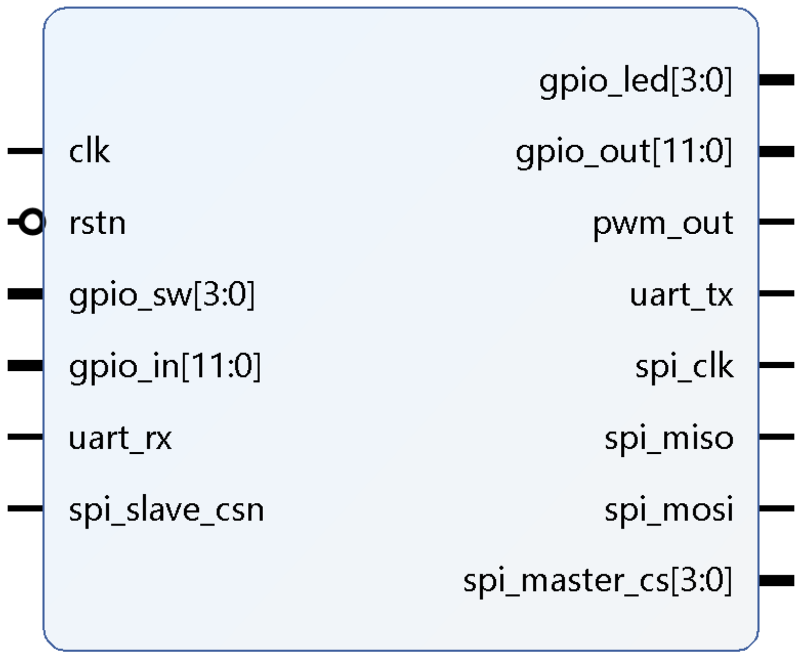
Data Bus: Custom MMIO

• mem_addr	: Memory Address	– 32 bit
• mem_rdata	: Memory Read Data	– 32 bit
• mem_wdata	: Memory Write Data	– 32 bit
• mem_ren	: Memory Read Enable	– 1 bit
• mem_wen	: Memory Write Enable	– 1 bit
• mem_wmask	: Memory Write Mask	– 4 bit

Peripherals:

- UART Transmitter, supporting 4800, 9600, 57600, 115200 baud rates
- UART Receiver, supporting 4800, 9600, 57600, 115200 baud rates
- SPI Master with 4 chip-select output
- SPI Slave
- 12 GPIO pins
- 4 built-in LED connections
- 4 built-in switch connections
- 1 Pulse Width Modulation-controlled output
- 1 Timer

3. PINOUT



Inputs:

- “clk”: 50 MHz Oscillator Clock
- “rstn”: Active – Low Reset Signal
- “gpio_sw[3:0]”: Four switch inputs
- “gpio_in[11:0]”: Twelve GPIO ports if selected as inputs
- “uart_rx”: UART Receiver Input
- “spi_slave_csn”: Active – Low SPI Slave Chip Enable

Outputs:

- “gpio_led[3:0]”: Four LED outputs
- “gpio_out[11:0]”: Twelve GPIO ports if selected as outputs
- “pwm_out”: PWM – Controlled Output
- “uart_tx”: UART Transmitter Output
- “spi_master_cs[3:0]”: SPI Master Chip Select

Inouts:

- “spi_clk”: SPI Clock
- “spi_miso”: SPI MISO
- “spi_mosi”: SPI MOSI

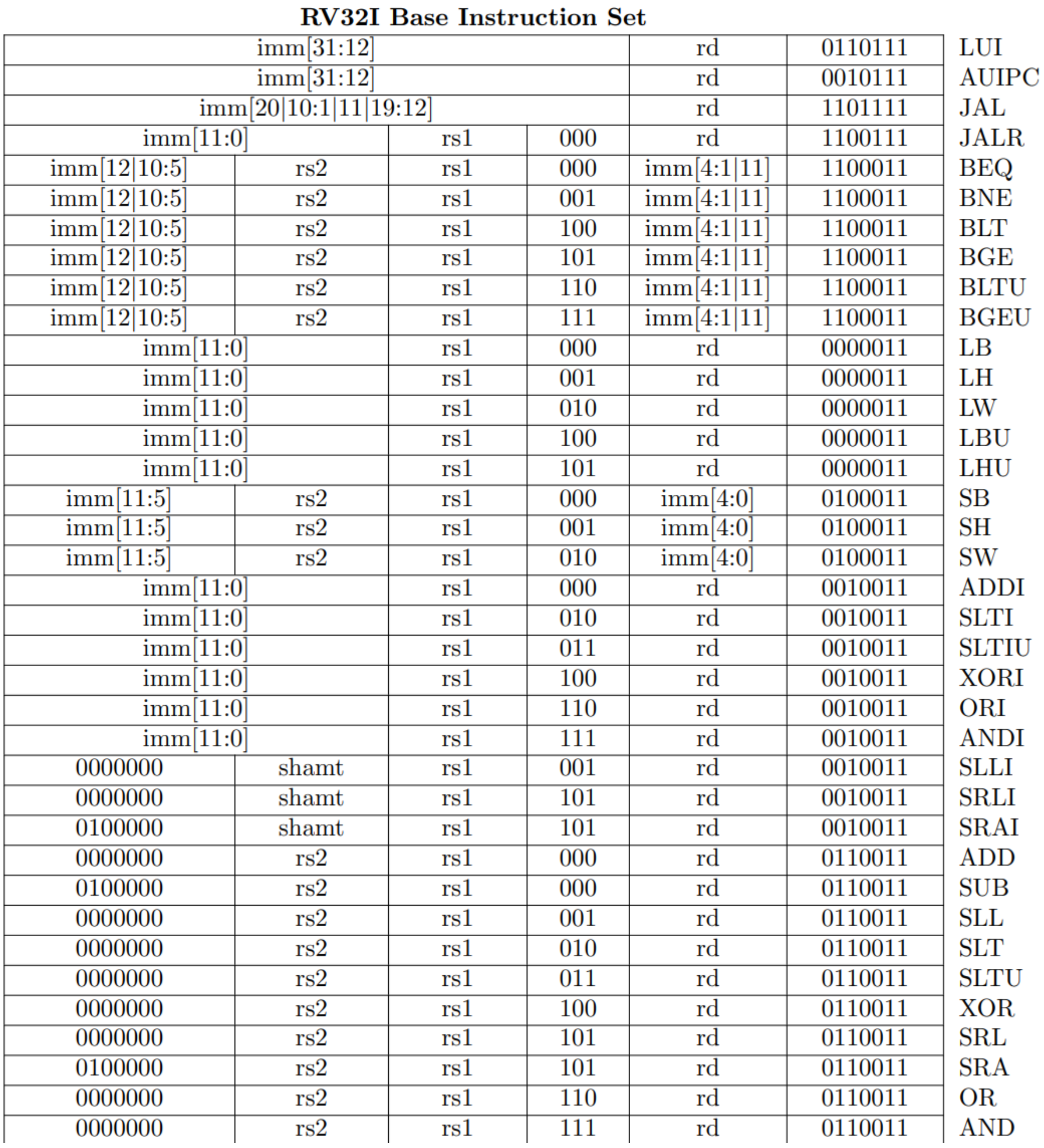
4. MEMORY MAP

Aether SoC has 4 GB Memory Map space with many reserved regions for future improvements and custom extensions.

Address Range	Size	Region	Mapped Peripheral	Status
`0x0000_0000 - 0x0FFF_FFFF`	256 MB	`0x0`	Data Memory (RAM)	Active
`0x1000_0000 - 0x1FFF_FFFF`	256 MB	`0x1`	—	Reserved
`0x2000_0000 - 0x2FFF_FFFF`	256 MB	`0x2`	—	Reserved
`0x3000_0000 - 0x3FFF_FFFF`	256 MB	`0x3`	—	Reserved
`0x4000_0000 - 0x4FFF_FFFF`	256 MB	`0x4`	GPIO Core	Active
`0x5000_0000 - 0x5FFF_FFFF`	256 MB	`0x5`	UART Core	Active
`0x6000_0000 - 0x6FFF_FFFF`	256 MB	`0x6`	Timer Core	Active
`0x7000_0000 - 0x7FFF_FFFF`	256 MB	`0x7`	—	Reserved
`0x8000_0000 - 0x8FFF_FFFF`	256 MB	`0x8`	SPI Core	Active
`0x9000_0000 - 0x9FFF_FFFF`	256 MB	`0x9`	—	Reserved
`0xA000_0000 - 0xAFFF_FFFF`	256 MB	`0xA`	—	Reserved
`0xB000_0000 - 0xBFFF_FFFF`	256 MB	`0xB`	—	Reserved
`0xC000_0000 - 0xCFFF_FFFF`	256 MB	`0xC`	—	Reserved
`0xD000_0000 - 0xDFFF_FFFF`	256 MB	`0xD`	—	Reserved
`0xE000_0000 - 0xEFFF_FFFF`	256 MB	`0xE`	—	Reserved
`0xF000_0000 - 0xFFFF_FFFF`	256 MB	`0xF`	—	Reserved

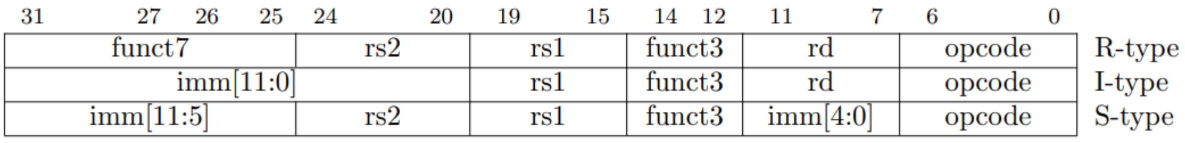
5. CORE PROPERTIES

The custom RISC-V core inside Aether SoC is using the RV32I Instruction Set Architecture given below.



RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND



31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

6. CHIP DESIGN SPECIFICATIONS

Processor, UART, SPI and Timer modules are hardened separately and then added as macros onto the final chip.

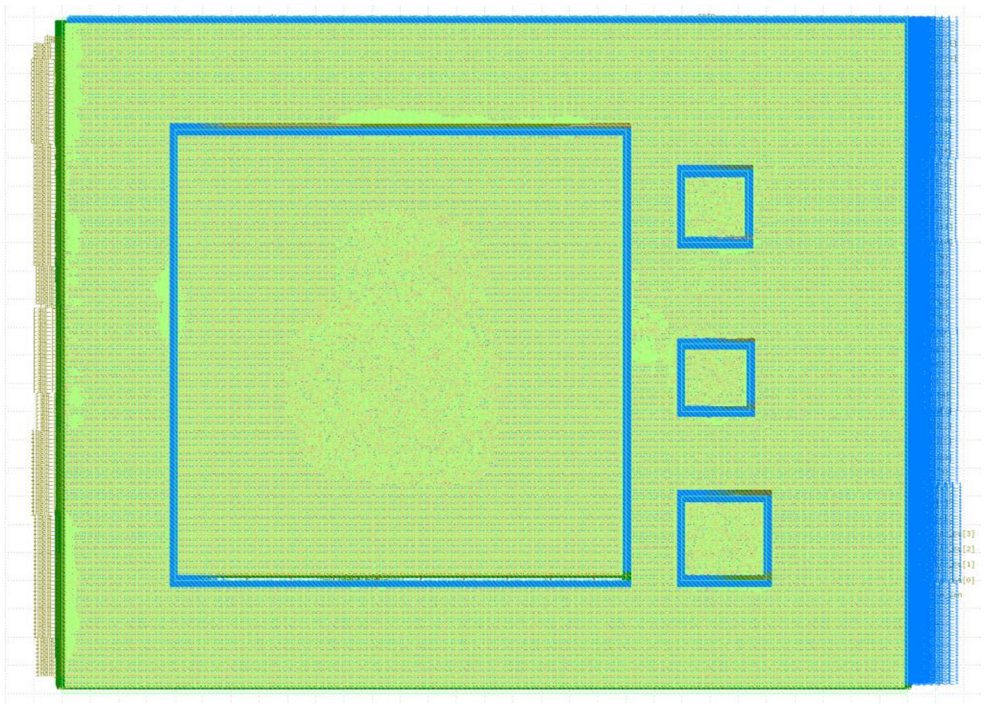
Note: The process is designed without the instruction (ROM) and data (RAM) memories and the connections are given as chip IOs.

Total die are is 1500x1200 microns and the core area is 5.52 x 10.88 x 1494.08 x 1188.64 microns.

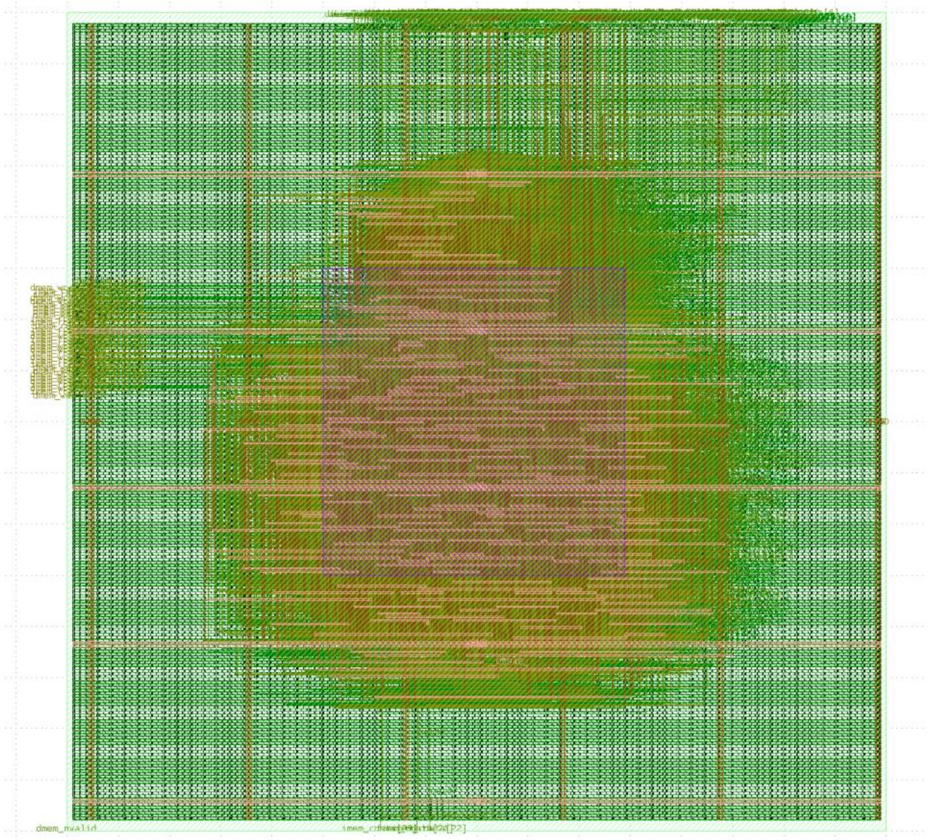
Total utilization is specified at the table below:

Module	Wire Bits	Public Wire Bits	Cells	Notable Instances
Top (`aether_soc`)	760	419	**466**	`rv32 core`, `uart_top`, `spi_top`, `timer_top`
`rv32_core`	8147	1502	**8080**	1024×DFXTP, 238×DFRTP, 1074×BUF
`spi_top`	582	235	**477**	93×DFXTP, 52×NOR2, 41×BUF
`timer_top`	1131	352	**1027**	68×DFXTP, 50×DFRTP, 63×BUF
`uart_top`	734	325	**629**	98×DFXTP, 56×BUF, 94×CONB

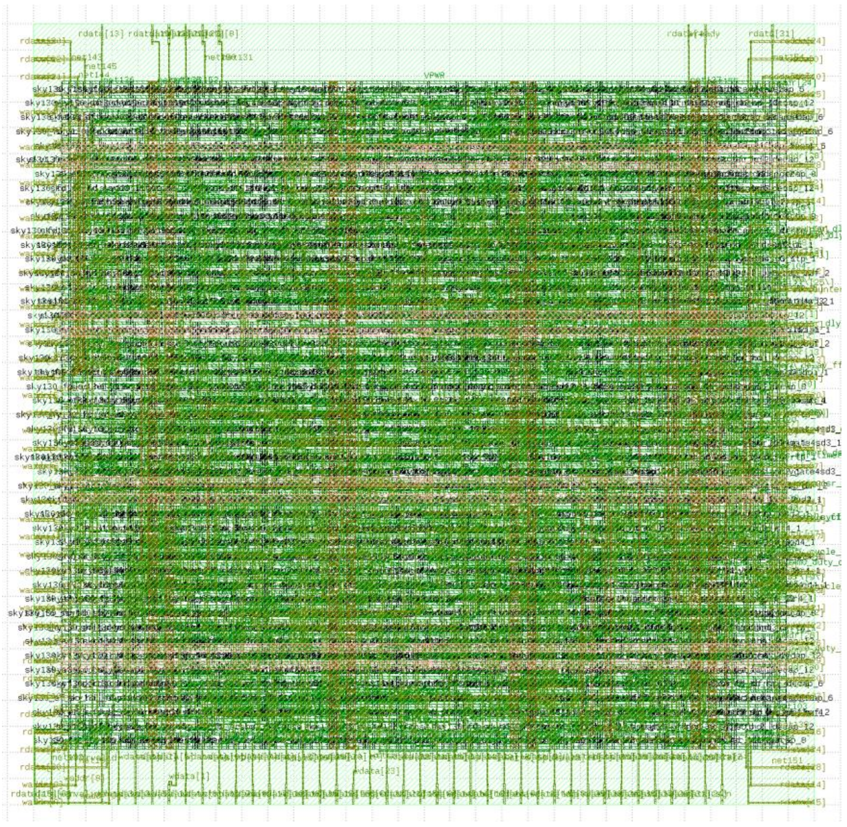
Aether SoC:



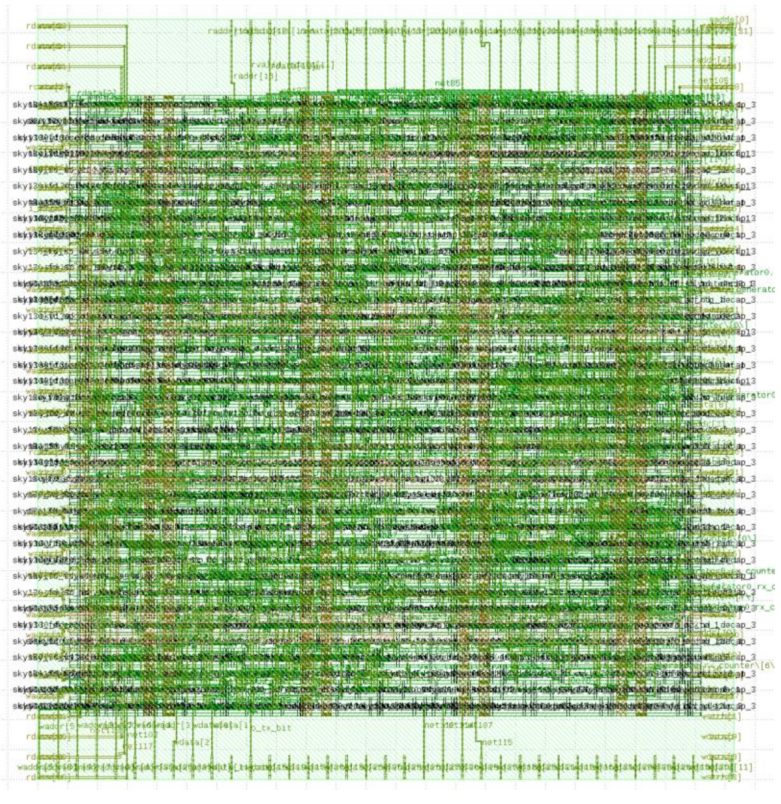
RV32 Core:



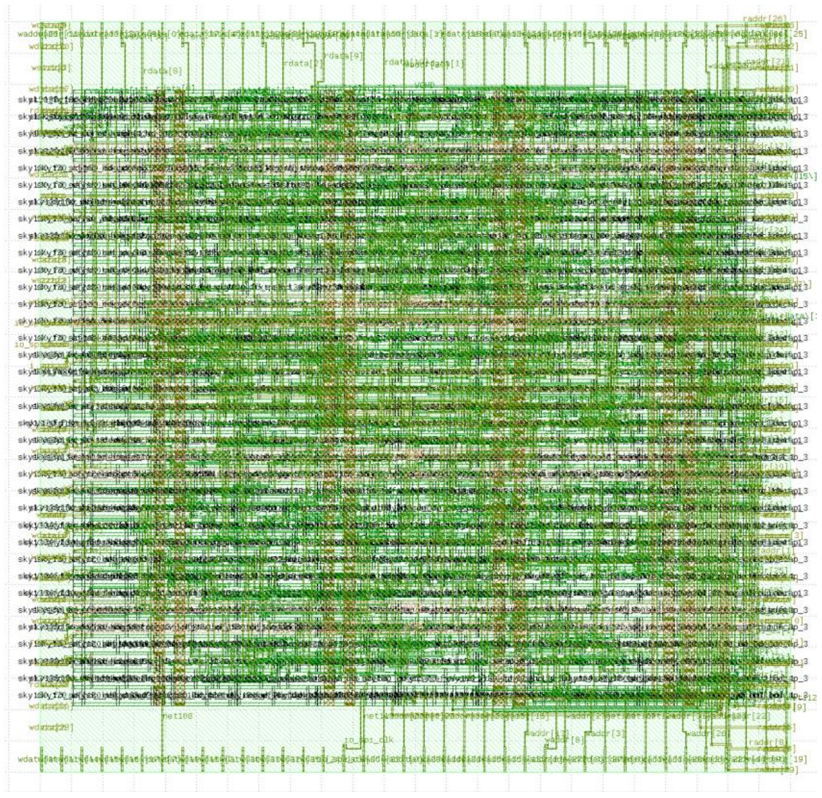
Timer:



UART:



SPI:

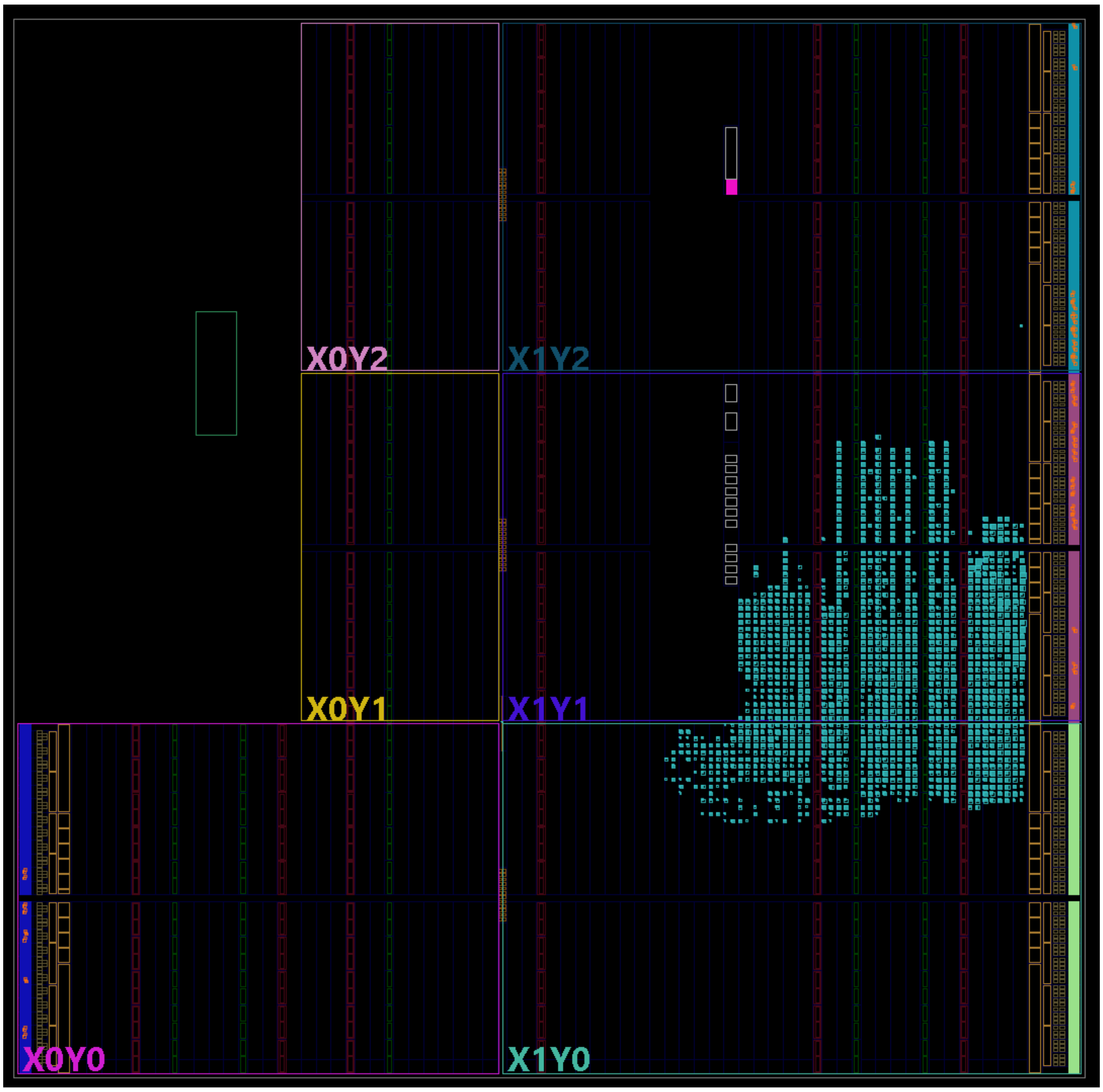


7. SOFTCORE VERSION FOR FPGA

A softcore version of the design is implemented for FPGA prototyping.

The design is synthesized and implemented in Xilinx Vivado and tested on the Zybo Z7 FPGA Development board.

Implementation on Zynq-7020:



Utilization:

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)
aether_soc	4707	1648	870	325	1382	3683	1024
├─data_mem	1092	0	530	256	302	68	1024
├─gpio0	4	44	0	0	26	4	0
├─rv32_core	3409	1257	339	69	1035	3409	0
│ ├─alu	21	0	0	0	32	21	0
│ ├─branch_unit	0	0	0	0	12	0	0
│ ├─instr_mem	544	0	63	1	170	544	0
│ └─regfile	2634	992	276	68	804	2634	0
├─spi0	54	93	1	0	28	54	0
│ ├─spi_master_i	40	43	1	0	18	40	0
│ ├─spi_regs_i	9	22	0	0	15	9	0
│ └─spi_slave_i	5	28	0	0	8	5	0
├─timer0	100	118	0	0	50	100	0
│ ├─timer_regs	64	67	0	0	33	64	0
│ └─timers0	37	51	0	0	43	37	0
└─uart_ip	51	96	0	0	29	51	0
├─ baud_rate_generator0	17	23	0	0	8	17	0
├─uart_regs	8	29	0	0	18	8	0
├─uart_rx0	10	27	0	0	8	10	0
└─uart_tx0	16	17	0	0	7	16	0

Timing Summary:

Time	Metric	Value
Setup	Worst Negative Slack (WNS)	5.793 ns
	Total Negative Slack (TNS)	0.000 ns
	Number of Failing Endpoints	0
	Total Number of Endpoints	14107
Hold	Worst Hold Slack (WHS)	0.052 ns
	Total Hold Slack (THS)	0.000 ns
	Number of Failing Endpoints	0
	Total Number of Endpoints	14107
Pulse Width	Worst Pulse Width Slack (WPWS)	8.750 ns
	Total Pulse Width Negative Slack (TPWS)	0.000 ns
	Number of Failing Endpoints	0
	Total Number of Endpoints	2673

Power Summary:

Metric	Value
Total On-Chip Power	**0.125 W**
Design Power Budget	Not Specified
Process	typical
Power Budget Margin	N/A
Junction Temperature	26.4 °C
Thermal Margin	58.6 °C (4.9 W)
Ambient Temperature	25.0 °C
Effective ΘJA	11.5 °C/W

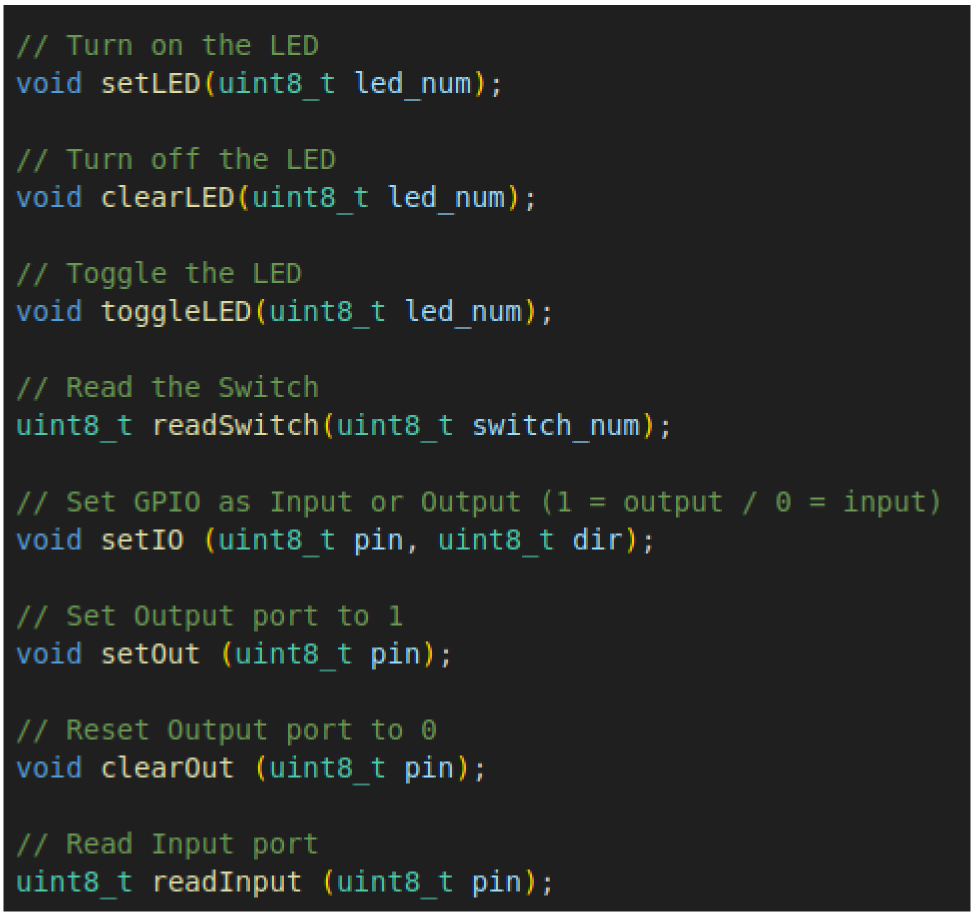
On-Chip Power Breakdown:

Category	Power	Percentage
Dynamic	0.019 W	15%
├─ Clocks	0.004 W	23%
├─ Signals	0.008 W	45%
├─ Logic	0.005 W	28%
└─ I/O	0.001 W	4%
Device Static	0.106 W	85%

8. FIRMWARE

For each peripheral extension, a C and its header file is created with firmware functions for controlling and manipulating the registers.

GPIO Functions:



```c
// Turn on the LED
void setLED(uint8_t led_num);

// Turn off the LED
void clearLED(uint8_t led_num);

// Toggle the LED
void toggleLED(uint8_t led_num);

// Read the Switch
uint8_t readSwitch(uint8_t switch_num);

// Set GPIO as Input or Output (1 = output / 0 = input)
void setIO (uint8_t pin, uint8_t dir);

// Set Output port to 1
void setOut (uint8_t pin);

// Reset Output port to 0
void clearOut (uint8_t pin);

// Read Input port
uint8_t readInput (uint8_t pin);
```

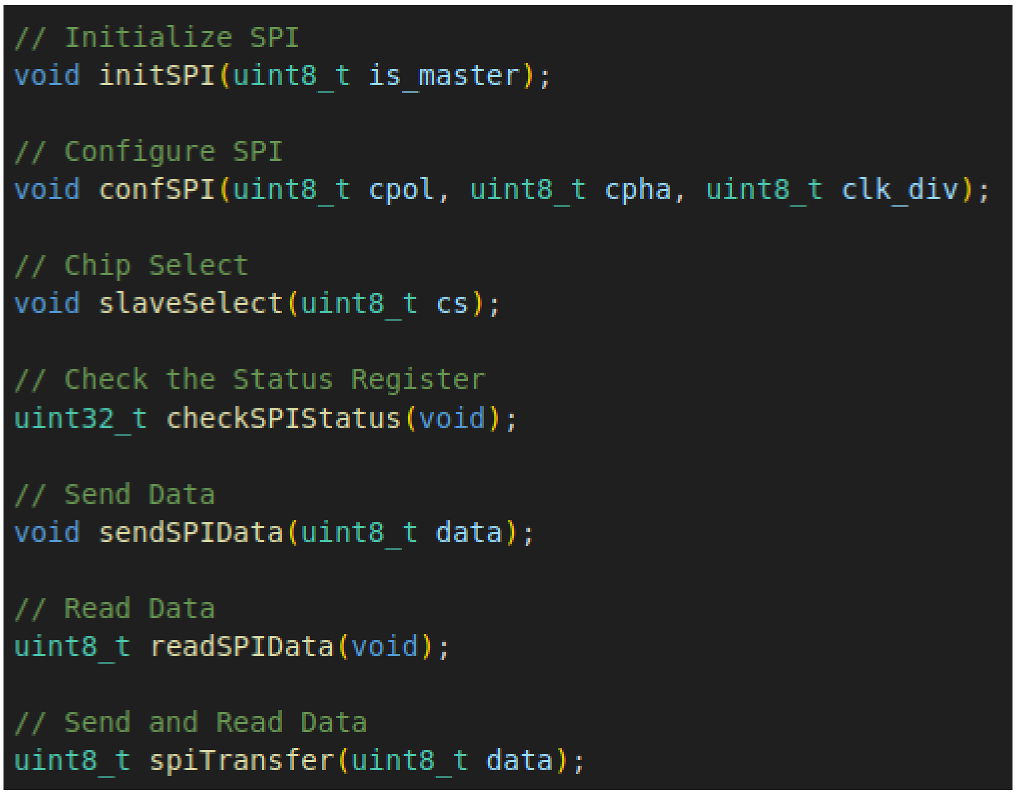
Timer Functions:

```c
// Set Delay in ms
void setDelayMili(uint32_t delay_ms);

// Set Delay in us
void setDelayMicro(uint32_t delay_us);

// Set PWM
void setPWM(uint16_t period_us, uint16_t duty_cycle);

// Stop PWM
void stopPWM();
```

UART Functions:

```c
// Initialize the UART and Set the Baud Rate
void startSerial(uint32_t baud);

// Disable the UART
void stopSerial(void);

// Send Data from UART TX
void sendChar(uint8_t data);

// Send String from UART TX
void sendString(const char *str);

// Read the UART RX
uint8_t readChar(void);

// Read the UART RX and Save it into the "buffer" with the size "max_length"
void readString(char *buffer, uint8_t max_length);

// Check TX Ready
uint8_t checkTX(void);

// Check RX Ready
uint8_t checkRX(void);
```

SPI Functions:

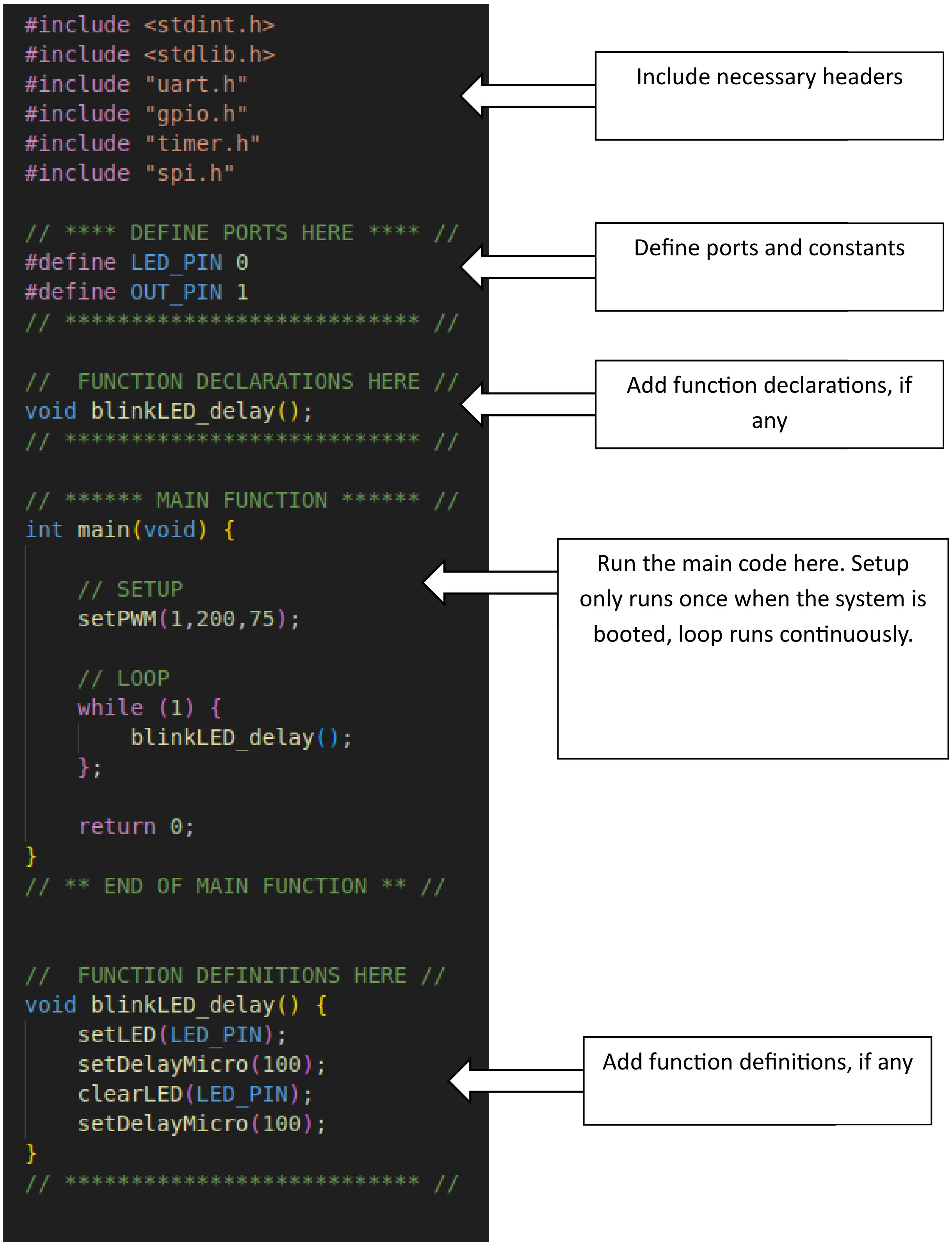


```c
// Initialize SPI
void initSPI(uint8_t is_master);

// Configure SPI
void confSPI(uint8_t cpol, uint8_t cpha, uint8_t clk_div);

// Chip Select
void slaveSelect(uint8_t cs);

// Check the Status Register
uint32_t checkSPIStatus(void);

// Send Data
void sendSPIData(uint8_t data);

// Read Data
uint8_t readSPIData(void);

// Send and Read Data
uint8_t spiTransfer(uint8_t data);
```

For the function details and example use projects, you can check the firmware file in the github repository.

9. PROGRAMMING AND CONFIGURING

General structure of a software code is given below.



```c
#include <stdint.h>
#include <stdlib.h>
#include "uart.h"
#include "gpio.h"
#include "timer.h"
#include "spi.h"

// **** DEFINE PORTS HERE **** //
#define LED_PIN 0
#define OUT_PIN 1
// *************************** //

//  FUNCTION DECLARATIONS HERE //
void blinkLED_delay();
// *************************** //

// ****** MAIN FUNCTION ****** //
int main(void) {

    // SETUP
    setPWM(1,200,75);

    // LOOP
    while (1) {
        blinkLED_delay();
    };

    return 0;
}
// ** END OF MAIN FUNCTION ** //


//  FUNCTION DEFINITIONS HERE //
void blinkLED_delay() {
    setLED(LED_PIN);
    setDelayMicro(100);
    clearLED(LED_PIN);
    setDelayMicro(100);
}
// *************************** //
```

Include necessary headers

Define ports and constants

Add function declarations, if any

Run the main code here. Setup only runs once when the system is booted, loop runs continuously.

Add function definitions, if any

For hardware extensions, the following steps are followed.

- Create new IP folder under 'core_generation' folder. It is recommended to copy any of the present IP folders and renaming for the structure.
- Create or add the HDL code of your peripheral core design.
- Create and configure 'regs.yaml' and 'csrconfig' file for your registers. These registers will be responsible for the connection between your hardware design and the processor.
- Execute $corsair and generate automated files for your registers. Make sure to use reserved memory locations for your registers for ensuring memory alignment.
- Open the '/sw' folder inside and create your own custom firmware header and functions. These functions will allow software to control the created registers.
- Copy the hardware files to src and software files to firmware folders respectively.
- Add your module instantiation to top.v module and connect it to the local bus.
- Create a new connection in device_sel module with your register address.
- Call your header file in the 'main.c' code. Write the software however you like to use the system.
- Run the following commands for generating new hex files with the new software and test it via simulation. You might need extra tasks on testbench that simulates inputs.

```
$ cd firmware

$ make clean

$ make

$ cd ..

$ python3 test.py top_tb.sv src /*
```

10. REFERENCES

- Logo design: https://chatgpt.com/
- Processor architecture: https://riscv.org/
- Register map design: https://github.com/esynr3z/corsair
- Peripheral bus design: https://www.udemy.com/course/building-a-risc-v-soc-from-scratch
- Physical integrated circuit design: https://github.com/The-OpenROAD-Project/OpenLane
- Verilog basics and SPI design: https://nandland.com/
- FPGA board: https://digilent.com/shop/zybo-z7-zynq-7000-arm-fpga-soc-development-board/