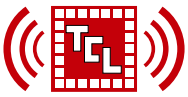
Telecommunications
Circuits Laboratory



Open-PIO

An Open-Source HDL Implementation of the RP2040 Programmable I/O

Ismael Tobias Frei

Master's Thesis at the Institute of Electrical and Micro Engineering
Spring 2025

Professor : Andreas Burg
Advisors : Ludovic Blanc
Christoph Müller

EPFL STI IEM TCL
ELD 131 (Bâtiment ELD)
Station 11

Contents

1 Introduction **1**
- 1.1 Definitions and Technical Context . . . 1
- 1.2 Motivation & Context . . . 1
- 1.3 Objective & Procedure . . . 2

2 Instruction Set & Architecture **3**
- 2.1 PIO Instruction Set . . . 3
  - 2.1.1 Side-set and Delay . . . 4
- 2.2 Architecture . . . 4
  - 2.2.1 Registers . . . 5
- 2.3 State Machine Design . . . 6
  - 2.3.1 Instruction Execution . . . 7
- 2.4 GPIO Priority Sorting . . . 9

3 Implementation **11**
- 3.1 Registers . . . 11
- 3.2 IRQ . . . 13
- 3.3 FIFOs . . . 13
- 3.4 Feature Matrix . . . 13

4 Verification with cocotb **15**
- 4.1 Testbench Organization & Makefile Flow . . . 15
- 4.2 Test Example: ISR Partial Right Shift . . . 15
- 4.3 Test Results . . . 16
- 4.4 Verification Limitations . . . 17

5 Hardware Implementation & Prototyping **19**
- 5.1 OpenPIO Wrapper & IPBus Interface . . . 19
- 5.2 Synthesis for FPGA . . . 20
  - 5.2.1 Timing Limitations . . . 20
- 5.3 Synthesis for 65nm ASIC . . . 21
- 5.4 Validation with programs . . . 21

6 Limitations & Future work **22**

7 Conclusion **23**

1 Introduction

1.1 Definitions and Technical Context

- **I/O:** Input/Output, refers to the communication between the microcontroller and the outside world.
- **FIFO:** First In First Out, a type of memory buffer that stores data in the order it was received. The first data written to the FIFO is the first data read from it.
- **Cycle accuracy:** The fidelity of a model in reproducing the timing behavior as the hardware system at the level of individual clock cycles.
- **Bit-banging:** A technique for serial communication where the CPU manipulates I/O pins manually to implement a protocol in software, often used when dedicated hardware peripherals are not available.
- **IMEM:** Instruction Memory, a memory block, that stores the instructions of a program.
- **GPIO:** General Purpose Input/Output, refers to the physical pin interface. Each pin can be accessed with 3 signals:
  - GPIO pindir: controls pin direction as output (1) or input (0)
  - GPIO out (or GPIO pin): This will be the valid logical output value of the GPIO pin, if the pin direction is set as output.
  - GPIO in: This will be the valid logical input value of the GPIO pin, if the pin direction is set as input.

1.2 Motivation & Context

CPUs in modern microcontrollers excel at complex algorithms. They can also bit-bang a single pin in tight assembly, but cannot sustain multiple parallel, cycle-true streams without dedicating essentially the entire execution pipeline to I/O. For example, driving address/data lines for custom bus protocols or generating VGA timing signals demands cycle-true parallel I/O, which a CPU alone can either not deliver, or it would be at the least very resource intensive.

On-chip peripherals like UART, SPI, VGA, and CAN implement a single I/O protocol in fixed logic. The advantages of separate peripherals instead of letting the CPU handle all I/O tasks are manifold:

- **Performance:** Dedicated hardware for I/O tasks allows the CPU to focus on computation, leading to better overall performance.
- **Parallel I/O:** Multiple I/O operations can be performed in parallel, which is not possible for a CPU that can only handle one task at a time.

- **Cycle Accuracy:** Hardware peripherals can perform I/O tasks with cycle accuracy, which the CPU is not always capable of (e.g. for parallel I/O).

The downside of this approach is that each hardware peripheral is specialized for one single communication protocol, and they are usually linked to a fixed set of GPIO pins. Thus, a microcontroller can only support the fixed set of I/O protocols for which it has a dedicated peripheral. The PIO (Programmable Input/Output), introduced in the Raspberry Pi RP2040 microcontroller, solves this problem. It is a programmable hardware block that can be configured to perform various I/O tasks. Architecturally it resembles a small 4-core CPU with its own very reduced instruction set, registers, and a program memory. Just a few examples of communication standards, that can be implemented with the PIO are, duplex SPI, UART, I2C, WS2812 LED control, and most importantly: custom communication protocols.

1.3 Objective & Procedure

The RP2040 PIO HDL is proprietary and a company secret. Therefore it is not known and cannot be implemented in open-source projects.

The objective of this project is to create an open-source PIO hardware block that is functionally equivalent to the RP2040 PIO, but can be adapted and integrated into other microcontroller designs, such as ASICs or SoCs.

The following procedure was followed to achieve this objective:

- **Study Specifications:** Analyze the RP2040 PIO datasheet [1] to understand the PIO's architecture, features, and instruction set. Run test programs on the original RP2040 to get hands-on understanding of its PIO blocks. During this phase a simplified mockup on python was created to understand the PIO's architecture. As the mockup is not functional or of use, it is not further discussed in this report.

- **Design:** With the specification as a manual, design the PIO hardware in Verilog, ensuring cycle accuracy and adherence to the datasheet.

- **Verification:** Develop a comprehensive test suite for each submodule to ensure functional correctness during simulation.

- **Prototyping:** Deploy the openPIO on an FPGA board to demonstrate its functionality.

- **Testing:** Write and run test programs to validate the openPIO's operation on the FPGA.

2 Instruction Set & Architecture

This chapter is taking a closer look at the instruction set and architecture of the PIO. Note, that the RP2040 datasheet [1] can provide additional information.

2.1 PIO Instruction Set

The instruction set of the PIO is encoded in a 16-bit operation code (op-code). It can be decoded in 9 different main instructions. All instructions are decoded in the bits 15 to 13. Bits 12 to 8 are used for the delay and side-set (see 2.1.1). Bits 7 to 0 are used for the instruction parameters, which can be immediate values, addresses, or conditions.

Table 1: Op-codes of the whole PIO Instruction Set. Replicated from the RP2040 Datasheet [1]

Bits / Name	15-13	12-8	7	6	5	4	3	2	1	0
`JMP`	000	Delay/side-set	Condition			Address				
`WAIT`	001		Pol	Source		Index				
`IN`	010		Source			Bit count				
`OUT`	011		Destination			Bit count				
`PUSH`	100		0	IfF	Blk	0	0	0	0	0
`PULL`	100		1	IfE	Blk	0	0	0	0	0
`MOV`	101		Destination			Op		Source		
`IRQ`	110		0	Clr	Wait	Index				
`SET`	111		Destination			Data				

Here is a brief overview of the instructions:

- `JMP`: Jump to an absolute address in the instruction memory. The jump can be conditional based on a few different conditions(jump if scratch X is zero).
- `WAIT`: Wait for a specific condition to be met before continuing. If the condition is not met, the state machine will stall until it is.
- `IN`: Read a data word from chosen source and shift N-bits of it into the ISR. Sources can be GPIO Pins, X, Y, ISR, OSR, or all zeros.
- `OUT`: Write a number if bits from the OSR to a chosen destination. Destinations can be GPIO Pins, X, Y, discarded, GPIO Pindirs, PC, ISR, or EXEC (execute 16 bits as next instruction).
- `PUSH`: Push full word of ISR into the RX-FIFO with some conditions (IfF: only if ISR full, Block: stall if RX-FIFO full).
- `PULL`: Pull full word from the TX-FIFO into the OSR with some conditions (IfE: only if OSR empty, Block: stall if TX-FIFO empty).

- `MOV`: Move full word from a source to a destination with an optional binary operation (None, bitwise inverse, reversed bit order).
- `IRQ`: Set or clear IRQ flags selected by Index argument or stall until an IRQ flag is cleared.
- `SET`: Set a destination (Pins, Pindirs, X, or Y) to a specific value. The value is encoded in the last 5 bits of the instruction and can be any value from 0x00 to 0x1F.

Apart from the `IRQ` instruction and its infrastructure, all instructions are implemented in openPIO.

2.1.1 Side-set and Delay

Parallel to every instruction, the state machine can add a delay and/or a side-set. The delay and side-set are encoded within the same 5 bits in the instruction op-code (bits 12-8). The control registers `SMn_EXECCTRL` and `SMn_PINCTRL` define their respective behavior and how the bits are separated between delay and side-set. Example: If the delay takes all 5 bits, the side-set is deactivated. If the side-set takes 3 bits, the delay takes 2 bits, and so on.

Delay: At the end of the instruction execution, the PIO waits for the encoded number of clock cycles before executing the next instruction. The maximum delay, encoded in 5 bits, is 31 clock cycles. With every added side-set bit, the maximum possible delay is reduced.

Side-set: The side-set can be used to set the state of up to 5 GPIO pins at the same time as the instruction execution. It can either take effect on the GPIO Output or the GPIO pin direction.

2.2 Architecture

The RP2040 has two PIO blocks. Architecture-wise each PIO block looks like a small 4-core processor with a very limited instruction set. In addition to the four state machines ("processor cores"), it contains an instruction memory, a TX- and RX-FIFO for each state machine, and a GPIO Mapper. The PIO blocks are themselves slave devices on a bus interface. The RP2040 PIO's master device is the CPU, which connects to the PIO via an AHB-Lite BUS. OpenPIO is BUS agnostic and can be wrapped to be a slave device to any bus interface.

The TX- and RX-FIFOs are the data link between the PIO and the master. The TX-FIFO is for outputting data. It is written to by the master device and read from by the state machine. The RX-FIFO does the inverse.

Note that all 4 state machines share the same instruction memory, which holds 32 instructions in total. Therefore, the program size is very limited.

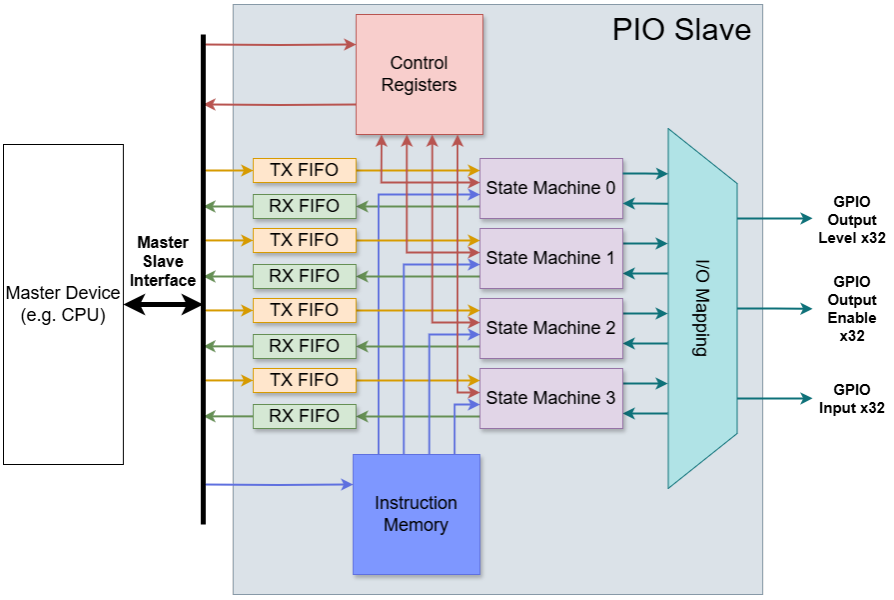


Figure 1: openPIO architecture. Inspired from *RP2040 Datasheet* [1]

2.2.1 Registers

The RP2040 PIO defines 81 x 32-bit registers—including global control, instruction memory, SM-specific control, and status/debug registers—each at its own bus address:

- Global control registers (e.g. `PIO_CTRL`, IRQ flags)
- The 32-word instruction memory (each word is 16 bits, but occupies a 32-bit address)
- State machine specific control registers (e.g. CLKDIV, EXECCTRL, SHIFTCTRL, PINCTRL)
- Additional status/debug registers.

Control Registers: Each 32-bit control register is comprised of individual bits or bit groups, controlling a specific feature or parameter of the PIO block or its state machines. They control parameters global parameters like IRQ flags or state machine specific parameters like the divided clock frequency, the wrap around address of a program, the GPIO Pin mapping, the behavior of the ISR and OSR, etc. Depending on the parameter they can either be read or written to by the master device, or both.

Instruction Memory: The instruction memory is a 32-word memory, where it stores all of the operation codes as 16-bit words. In the memory mapping, it behaves, however, like a 32-bit memory, by driving the 16 MSB as zeros.

2.3 State Machine Design

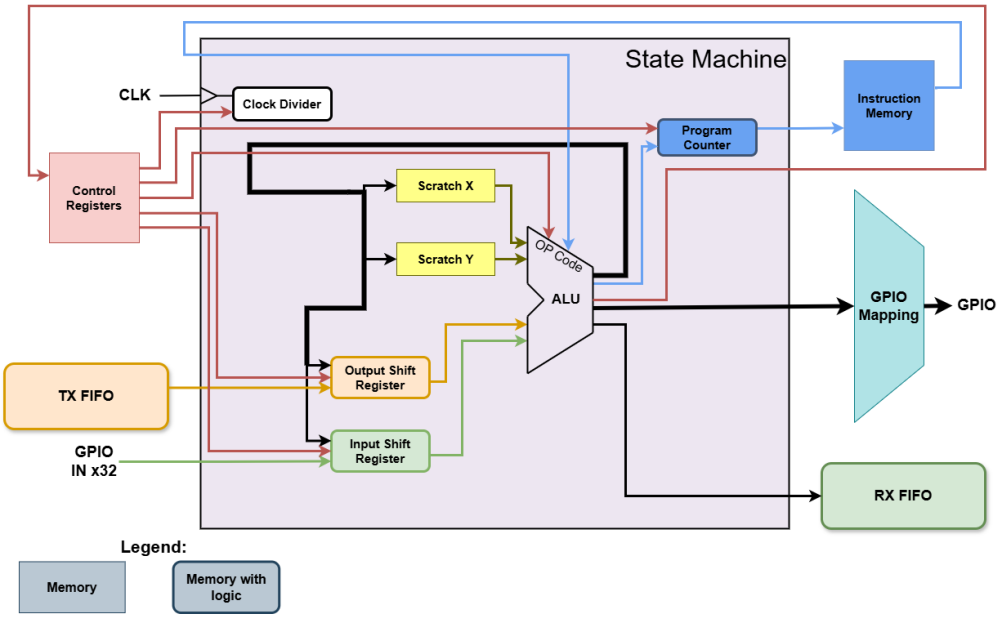


Figure 2: openPIO's State Machine block diagram, based on the RP2040 datasheet's behavioral description.

Clock Divider: The clock divider allows each state machine to run at a different clock frequency. Architecturally it is a first-order delta-sigma modulator, which allows for fractional clock division. With a 16-bit integer part and an 8-bit fractional part, coming from `SMn_CLKDIV`, it can divide the clock by a factor of 1 to 65536 in steps of 1/256. It provides the clock signal to the whole state machine. Every state machine can run at a different clock speed and at a different clock phase. In the RP2040's PIO, there is a mechanism in place to synchronize the clock phase of all state machines. That mechanism is controlled by the IRQ instruction, which is not yet implemented in the openPIO project.

Program Counter: is a 5-bit register that points to the current instruction in the instruction memory. By default it increments after every instruction and wraps around at the end of the program. The program end and wrap target addresses are both set in the control register `SMn_EXECCTRL` (where n is the state machine number, 0-3). This means that different programs can coexist in the same instruction memory. The JMP instruction can overwrite the program counter.

Side-Load Instruction: The EXEC register is a 16-bit register that can be loaded with an instruction to be executed next. It can be loaded by the `MOV` and `OUT` instruction. If an instruction is executed from the EXEC register, the program counter is not incremented.

Scratch Registers: The two scratch registers (X and Y) are 32-bit registers with no combinational logic. They are used to store temporary values during the execution of the program. They can also serve in basic operations, like countdowns.

GPIO Mapper: The GPIO Mapper is a combinational block that combines all the GPIO related signals to a few usable input/output signals. All output related signals will go from there to the priority sorter. Figure 3 shows the official block diagram of the GPIO Mapper. OpenPIO implements the same architecture.

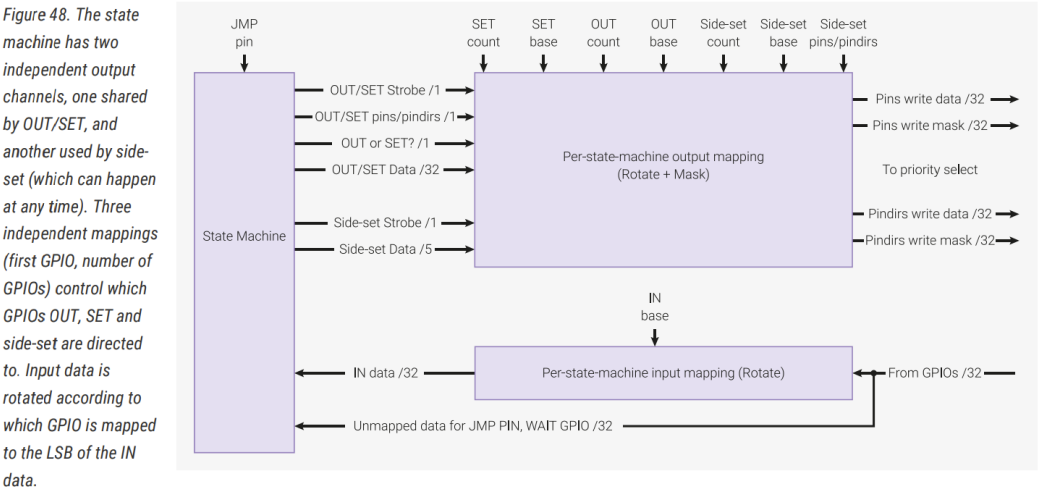


Figure 3: Block diagram of the PIO's GPIO Mapper, reproduced from the *RP2040 Datasheet*, Raspberry Pi Foundation (2021), p.341 [1].

Input Shift Register (ISR): The ISR can (left or right) shift in N bits from a selected source (e.g. GPIO pins or the X/Y scratch registers). On a PUSH instruction, it can push a whole 32-bit word into the RX-FIFO. It can also automatically push the word into the RX-FIFO, once a fill up threshold of the ISR is reached. This is called autopush and can be configured via the control registers. Note that for the ISR and OSR, the FIFO interactions are always word-wise, while the other interactions can be of any length (0 to 32 bits).

Output Shift Register (OSR): The OSR does about the same thing as the ISR but in reverse. It can shift out N bits to a selected destination. On a PULL instruction, it can pull a whole 32-bit word from the TX-FIFO. It also has autopull, which can automatically pull a word from the TX-FIFO, once the OSR passes an empty threshold.

2.3.1 Instruction Execution

The Execution Unit (analog to an ALU in a processor) is a purely combinational block that executes the current instruction and defines the next state of the whole state machine.

It defines the signals of the delay, side-set, the next values of scratch X/Y and EXEC register, and output signals. It controls ISR, OSR, and program counter blocks. All that by executing the current instruction.

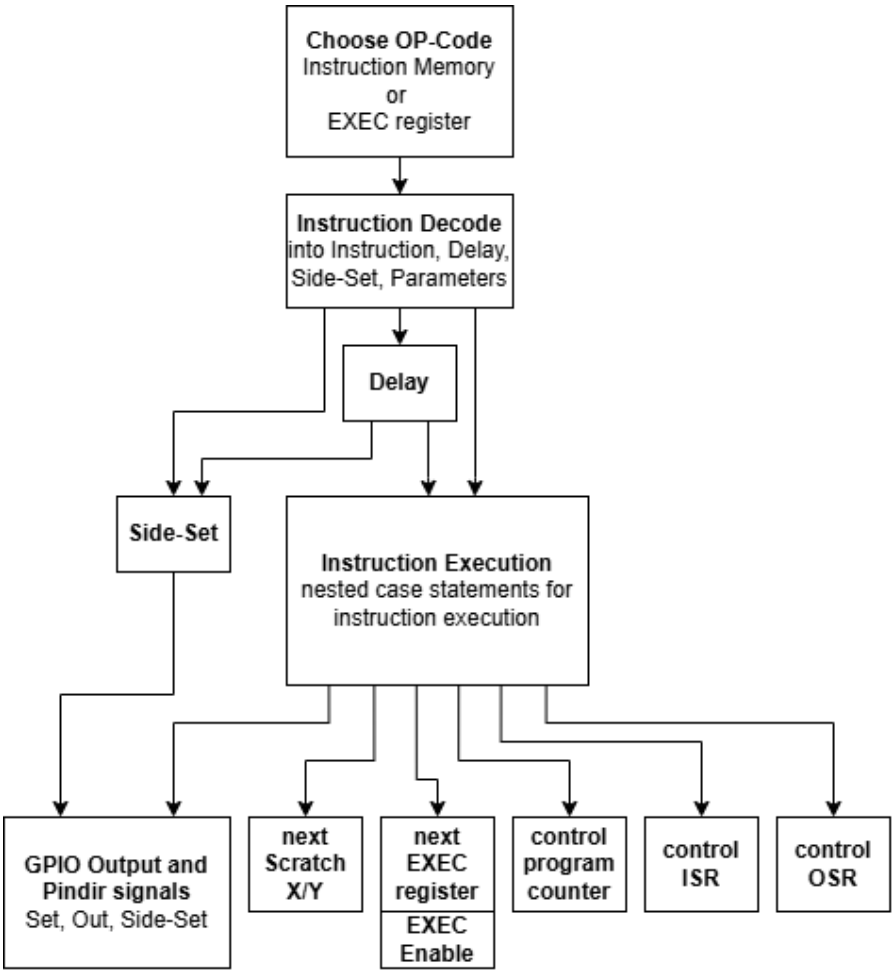


Figure 4: Signal path inside the state machine and its Execution Unit

Multiplexing between IMEM and EXEC: Because the state machine can side-load an instruction into the EXEC register, the ALU must first select, which 16-bit word to decode. If `reg_EXECEnable` is high, it uses the EXEC register (and clears `reg_EXECEnable`); otherwise it uses the instruction fetched from the instruction memory.

Instruction Decoding: The instruction decoder combinationally decodes the 16-bit op-code into instruction, side-set, delay, and instruction parameters, which are then used by the execution unit.

Delay: The delay counter defines if the next instruction should be executed. If it is not zero, the execution unit stalls and the delay counter is decremented by one. Else, the next instruction is executed.

Side-set: The side-set is part of the execution unit and drives the side-set output signals, if the instruction is not delayed.

Instruction Execution: The instruction execution consists of a big case statement for each instruction and nested case and if-else statements for their instruction conditions. This makes readability easy, which is perfect for an open-source project in an educational environment.

2.4 GPIO Priority Sorting

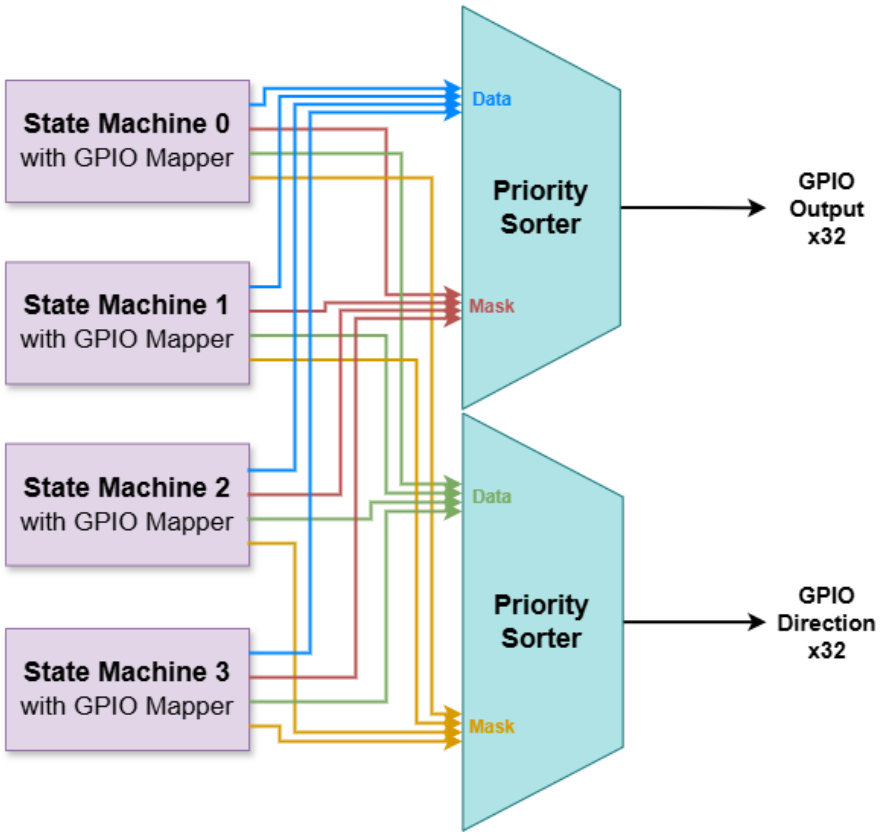


Figure 5: block diagram of the GPIO Priority Sorter.

As all of the four state machines interact with the same 32 GPIO pins, a priority sorter is needed to avoid collisions and determine what signals (`GPIO_out` and `GPIO_pindir`) are actually written to the GPIO interface. The priority sorter selects the active signals of

the state machine with the highest priority (SM3 > SM2 > SM1 > SM0). This applies to both the `GPIO_out` and `GPIO_pindir` signal. This behavior follows the RP2040 datasheet (§3.5.6.1. Output Priority). In Figure 5, each of the two identical sorter blocks receives a 32-bit mask and a 32-bit data signal from each state machine. Only the bits with a corresponding enable bit in the mask are considered for the output.

3 Implementation

The last chapter described the whole PIO architecture. However, there are some features that are not implemented in the openPIO project yet. This chapter will describe the current state of the openPIO implementation, including the implemented features and the missing ones.

3.1 Registers

Control Registers: In openPIO, we have prioritized only a subset of the control registers. And of those implemented, some control bits are not yet functional. In the current implementation, contrary to the datasheet, all registers are read-write for the master device and the PIO. Table 2 lists only the registers, that openPIO currently implements. All addresses and bit field positions are true to the datasheet to allow binary compatibility with the RP2040.

Instruction Memory: The openPIO's instruction memory has 5 read ports instead of 4, as per the original RP2040 PIO. This allows for it to be read and write for the master over the bus interface. In this implementation, the write port is synchronous (only writing to a given address, on a positive clock edge and with a writeEnable signal). All of the read ports are asynchronous (outputting new read data as soon as the read address changes). This read configuration allows the state machines to read and execute every instruction in the same clock cycle. Because of its non-standard number of write and read ports, the instruction memory is implemented as an array of 32x16-bit wide flip flops. Implementing a register array has negligible impact on area or performance at this scale.

Table 2: List of all implemented PIO registers in openPIO. The status column indicates the implemented bitfields. ✓indicates fully implemented, ⚠partially implemented, and ✗not implemented. See the RP2040 datasheet [1] for the exact function of each bitfield.

Address	Register Name	Status	Implemented bit fields
0x000	CTRL	⚠	✓[11:8]: SMn_CLKDIV_RESTART ✗[7:0]: SMn_RESTART ✓[3:0]: SMn_ENABLE
0x048 - 0x0C4	INSTR_MEM 0-31	✓	✓Instruction memory, 32 instructions, 16 bits each
0x0c8	SM0_CLKDIV	✓	✓[31:16]: INT ✓[15:8]: FRAC
0x0cc	SM0_EXECCTRL	⚠	✗[31]: EXEC_STALLED ✓[30:29]: SIDE_EN, SIDE_PINDIR ✓[28:24]: JMP_PIN ✗[23:18]: OUT_EN_SEL, INLINE_OUT_SEL ✓[17] OUT_STICKY ✓[16:7]: WRAP_TOP, WRAP_BOTTOM [6:4]: Reserved ✗[4:0]: STATUS_SEL, STATUS_N
0x0d0	SM0_SHIFTCTRL	⚠	✗[31:30]: FJOIN_RX, FJOIN_TX ✓[29:20]: PULL_THRESH, PUSH_THRESH ✓[19:18]: OUT_SHIFTDIR, IN_SHIFTDIR ✓[17:16]: AUTOPULL, AUTOPUSH [15:0]: Reserved
0x0dc	SM0_PINCTRL	✓	✓[31:26]: SIDESET_COUNT, SET_COUNT ✓[25:20]: OUT_COUNT ✓[19:10]: IN_BASE, SIDESET_BASE ✓[9:0]: SET_BASE, OUT_BASE
0x0e0	SM1_CLKDIV	✓	same as for corresponding SM0 register
0x0e4	SM1_EXECCTRL	⚠	same as for corresponding SM0 register
0x0e8	SM1_SHIFTCTRL	⚠	same as for corresponding SM0 register
0x0f4	SM1_PINCTRL	✓	same as for corresponding SM0 register
0x0f8	SM2_CLKDIV	✓	same as for corresponding SM0 register
0x0fc	SM2_EXECCTRL	⚠	same as for corresponding SM0 register
0x100	SM2_SHIFTCTRL	⚠	same as for corresponding SM0 register
0x10c	SM2_PINCTRL	✓	same as for corresponding SM0 register
0x110	SM3_CLKDIV	✓	same as for corresponding SM0 register
0x114	SM3_EXECCTRL	⚠	same as for corresponding SM0 register
0x118	SM3_SHIFTCTRL	⚠	same as for corresponding SM0 register
0x124	SM3_PINCTRL	✓	same as for corresponding SM0 register

3.2 IRQ

The whole `IRQ` instruction and its infrastructure is not yet implemented in openPIO. This includes the `IRQ`'s control registers and the `WAIT` instruction's IRQ condition.

3.3 FIFOs

The TX- and RX-FIFOs are missing in the openPIO implementation. This means that the `PUSH` and `PULL` instructions work from the state machine side, but are unconnected to the corresponding FIFO.

3.4 Feature Matrix

The following is a full feature matrix of the openPIO project, comparing it to the RP2040 PIO datasheet [1]:

Table 3: Feature Matrix of openPIO with corresponding datasheet reference. ✓indicates fully implemented, ⚠partially implemented, and ✗not implemented.

Feature	Datasheet Ref	Status	Notes
Instruction Memory	–	✓	32 instructions, 16-bit wide
OSR/ISR with shift counters	§3.2.3.1-2	✓	–
Scratch Registers	§3.2.3.4	✓	Two 32-bit registers, X and Y.
FIFOs	§3.2.3.5	✗	Not implemented yet.
Instruction Set	§3.4	⚠	IRQ is not yet implemented. The other 8 instructions are.
Side-set & Delay	§3.5.1	✓	
Program Wrapping	§3.5.2	✓	The program can be wrapped to loop between any address a and b.
FIFO Joining	§3.5.3	✗	Not implemented yet.
Autopush and autopull	§3.5.4	⚠	Implemented from state machine side, but not from the FIFO side.
Clock Divider	§3.5.5	✓	First-order delta-sigma for the fractional divider.
GPIO Mapping & Output Priority	§3.5.6	✓	SM3 has output priority over SM2, SM1 and SM0.
Forced and EXEC'd Instructions	§3.5.7	⚠	EXEC'd instructions are implemented, but not forced instructions.
Control Registers	–	⚠	Some control registers are implemented, but not all. See Table 2

Table 3 shows that, although openPIO aims to implement the full RP2040 PIO feature set, items such as IRQ/interrupt handling, multi-entry FIFOs, and forced instructions remain unimplemented or partially implemented. Other core features (JMP, WAIT, IN, OUT, MOV, SET, side-set/delay, program wrapping, GPIO mapping, and the clock divider) are cycle-accurate and match the datasheet.

4 Verification with cocotb

In this chapter we will discuss the verification toolchain to test the OpenPIO design in simulation.

4.1 Testbench Organization & Makefile Flow

With the available infrastructure at the Telecommunications Circuits Laboratory of EPFL, we use a toolflow based on cocotb (coroutine-based co-simulation testbench) [2] together with Modelsim [3]. A single Makefile target builds all HDL, opens ModelSim, and runs the Python tests for the specified module. A helper script collects all resulting XML reports and summarizes them in the terminal output. Table 4 is based on this output. Each relevant submodule (instructionDecoder, stateMachine, isr, osr, smGPIOMapper, instructionMemory, clockDivider, binaryOperations) has its own testbench. Furthermore, there is a top-level testbench that tests reading and writing to all memory-mapped registers of the PIO through IPBus.

4.2 Test Example: ISR Partial Right Shift

This is a short example with the test `test_isr_partial_shift_right`, which tests the ISR module's partial right shift functionality. Here is a diagram of the ISR module, our DUT (Device Under Test):

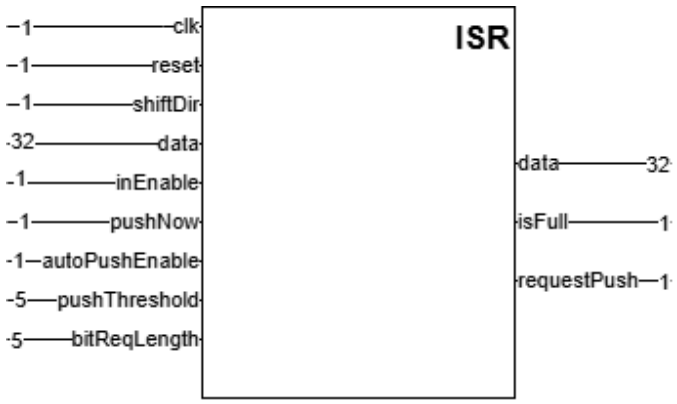


Figure 6: Black box ISR module block diagram with all of its I/O signals and their respective bitlengths

Here is the testbench code, simplified as pseudo-code:

```
function test_isr_partial_shift_right():
    TESTVALUE = 0b01001100011100001111000001111100

    reset ISR

    for n in 1 to 31 do
        // clear any leftover bits
        ISR.in_pushNow = 1
        wait one clock cycle
        ISR.in_pushNow = 0
        assert ISR.reg_data       == 0
        assert ISR.reg_shiftCount == 0

        // configure for an n-bit right shift
        ISR.in_shiftDirection = RIGHT
        ISR.in_bitReqLength   = n
        ISR.in_data           = TESTVALUE
        ISR.in_inEnable       = 1
        wait one clock cycle
        ISR.in_inEnable       = 0

        // compute what we expect
        expected_data  = (TESTVALUE & ((1 << n) - 1)) << (32 - n)
        expected_count = n

        // verify
        assert ISR.reg_data       == expected_data
        assert ISR.reg_shiftCount == expected_count

    end for
```

Listing 1: Testbench pseudo-code for the ISR partial right shift test.

The testbench resets the ISR module, then iterates through all possible shift counts from 1 to 31. In each iteration it empties the ISR by pushing its contents, then configures the ISR for a right shift of the test value by the current count. It waits for one clock cycle, then verifies that the ISR's data and shift count registers contain the expected values.

Thanks to cocotb, all of the testbenches are written in Python, which allows for a very high-level and readable testbench code.

4.3 Test Results

The test results are summarized in Table 4.

Table 4: List of all tests and results, performed in simulation

Module	Test	Status
clockDivider	clock frequency test with several integer and fractional clock divisions	**PASS**
instructionDecoder	Instruction Decoding	**PASS**
	Side-Set Count with SideSetEnable Off	**PASS**
	Side-Set Count with SideSetEnable On	**PASS**
binaryOperations	none, bitwise inversion, bit reverse order (for MOV instruction)	**PASS**
stateMachine	series of instructions to fill all registers (scratch X/Y, ISR/OSR) with test values (used for all following instructions tests)	**PASS**
	JMP	**PASS**
	WAIT	**PASS**
	IN	**PASS**
	OUT	**PASS**
	PUSH	**PASS**
	PULL	**PASS**
	MOV	**PASS**
	test the delay function with an instruction	**PASS**
isr	partial shift right (from 0 to 32)	**PASS**
	partial shift left (from 0 to 32)	**PASS**
	partial shift right (from 0 to 32) while pushing	**PASS**
osr	pull the full register	**PASS**
	shift output left and right while output is disabled	**PASS**
	shift output left and right while output is enabled	**PASS**
smGPIOMapper	Map Side-Set Signals	**PASS**
	Map SET and OUT signals	**PASS**
	Map SET/OUT and Side-Set Signals	**PASS**
	Input Mapping	**PASS**
instructionMemory	complete filling and reading with different test data	**PASS**
top	Send read and write packages to IPBus for all mapped PIO registers and verify response (see chapter 5.1)	**PASS**

As all of the tests were created to verify the simulated behavior of the OpenPIO design, all tests currently pass successfully. At every change in the design, the tests can be run to verify that the design still behaves as expected. The testbench code is available in the project repository [4]. All instructions were tested with all of their implemented parameter combinations.

4.4 Verification Limitations

There are some limitations to the verification process.

- **Edge cases:** The testbench is not exhaustive, and some edge cases are not tested.

- **Misinterpretation:** Our tests check DUT vs. the author's own test. Misinterpretations of the RP2040's specification would not be caught by the tests.
- **Timing:** The simulation cannot verify timing constraints, as it is not a real hardware test.
- **PIO-wide test:** While testbenches are written for all submodules, including the stateMachine as a whole, there is no testbench that tests the PIO block as a whole with several state machines and different clock divisions.

The timing, and PIO-wide test limitations can be addressed in the next chapter, where the hardware implementation of the OpenPIO design will be discussed.

5 Hardware Implementation & Prototyping

In this chapter we take a look at the physical implementation of the OpenPIO design on an FPGA.

The FPGA used for the implementation is the TUL PYNQ-Z2 board [5].

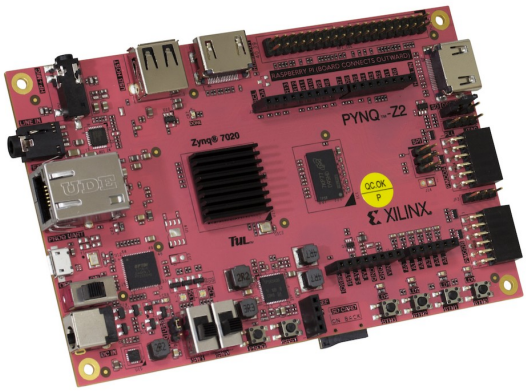


Figure 7: TUL PYNQ-Z2 board based on the Xilinx Zynq-7000 SoC.[5]

5.1 OpenPIO Wrapper & IPBus Interface

As seen in Figure 1 the PIO is a slave device on a bus interface. To control the PIO, a bus master device is needed to set the control registers and program the instruction memory. The openPIO prototype on FPGA wraps the PIO in an adaptation of the IPBus project [6] as the bus master device and contact point to the computer.

IPBus [6] is mostly open-source project developed by CERN. It creates a simple packet-based control protocol for reading and writing memory-mapped resources. However, there is one part that is not open-source:

IPBus uses a Xilinx proprietary block in its ethernet interface. However, the openPIO project aims for open-source and cross-platform compatibility. Therefore, Delphine Allimann's adaptation of IPBus was used [7], which combines IPBus with the open-source verilog-ethernet project [8]. This allows for a fully open-source and platform agnostic implementation of openPIO. Figure 8 shows a block diagram depicting Delphine Allimann's project, integrating verilog-ethernet and IPBus, connected to openPIO.

Topologically, the openPIO is wrapped as a slave device inside the IPBus project.

Delphine Allimann's repository also provides a Python library for simulation of the IPBus protocol, which was used to test writing and reading the PIO registers over IP in simulation.

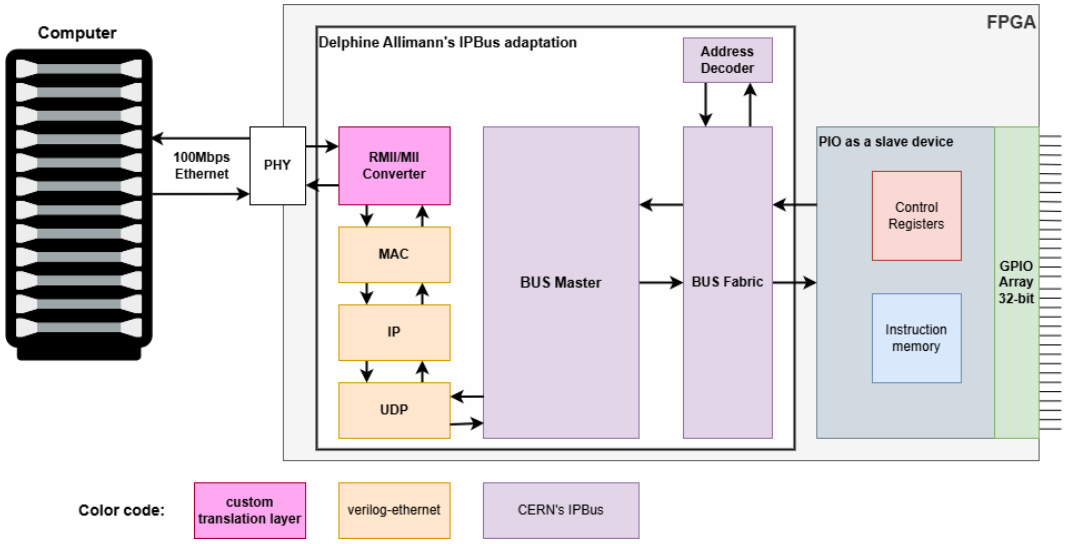


Figure 8: openPIO IPBus Wrapper, integrating verilog-ethernet and IPBus. Inspired from Delphine Allimann's adaptation of IPBus [7].

Clock Bridge: IPBus logic runs at 31.25 MHz, while the PIO state machines target 125 MHz. The IPBus adaptation includes a built-in clock-bridge to safely cross between these domains, ensuring reliable packet handling without metastability.

5.2 Synthesis for FPGA

As the whole project was already written in HDL and the integration into IPBus was already tested in simulation, the porting to FPGA was straightforward.

5.2.1 Timing Limitations

Although the RP2040 PIO is designed for a 125 MHz clock, our PYNQ-Z2 FPGA prototype only meets timing at half, 62.5 MHz. Vivado's post place & route timing report consistently flags paths whose total delay exceeds the 8 ns clock period. By inspecting the critical-path breakdown, we found that the programmable interconnect (wires plus switch-matrix hops) contributes far more delay than the pure LUT/combinational logic. This effect is intrinsic to FPGAs: each net must traverse a network of configurable switch blocks, inflating interconnect delay. In an ASIC implementation, which will be our ultimate target, the metal-wire interconnect is fixed and much shorter, and there is no large programmable switch fabric. As a result, meeting a 125 MHz clock (8 ns) should become straightforward given our moderate logic depth.

Using a more powerful FPGA would probably also have solved the timing issues, but as it was only for prototyping, these results were sufficient for the project.

5.3 Synthesis for 65nm ASIC

A synthesis in TSMC 65nm technology was performed to estimate the area and timing of the design. Timing constraints at 125MHz were met without issues, as expected. The area of one single PIO block without the IPBus infrastructure is $0.06mm^2(62'041\mu m^2)$. At this size per PIO block in 65 nm, the design is a compact, practical I/O accelerator for SoC integration. Currently, no post-sythesis simulation has been performed.

5.4 Validation with programs

To date there have been two programs written and tested on the openPIO on FPGA: a slow and a fast blinky program. The two programs confirm the correct operation of the openPIO implementation, wrapped by IPBus. It has GPIO interactions and uses The SET, MOV (Y to Y as NOP instruction), and JMP instructions as well as the delay functionality. Listing 2 shows all programmed registers on openPIO for the slow blinky program to run on state machine 0.

```
CTRL         = 1                        //Activate SM0
SM0_CLKDIV   = 65535 << 16              //clk_div=65535
SM0_EXECCTRL= (31 << 12) | (1<<17)  //WRAP_TOP=31, OUT_STICKY=1
SM0_PINCTRL = 5 << 26                   //SET_COUNT=5

iMEM0 = 0b1110000010000001  //SET 1 to PINDirs, 31 delay
iMEM1 = 0b1110000010000001  //SET 1 to PINs,    31 delay
iMEM2 = 0b1011111101000010  //NOP (MOV Y to Y), 31 delay
iMEM3 = 0b1011111101000010  //NOP (MOV Y to Y), 31 delay
iMEM4 = 0b1011111101000010  //NOP (MOV Y to Y), 31 delay
iMEM5 = 0b1011111101000010  //NOP (MOV Y to Y), 31 delay
iMEM6 = 0b1011111101000010  //NOP (MOV Y to Y), 31 delay
iMEM7 = 0b1111111000000000  //SET 0 to PINs,    30 delay
iMEM8 = 0b1011111101000010  //NOP (MOV Y to Y), 31 delay
iMEM9 = 0b1011111101000010  //NOP (MOV Y to Y), 31 delay
iMEM10= 0b1011111101000010  //NOP (MOV Y to Y), 31 delay
iMEM11= 0b1011111101000010  //NOP (MOV Y to Y), 31 delay
iMEM12= 0b1011111101000010  //NOP (MOV Y to Y), 31 delay
iMEM13= 0b1011111101000010  //JMP to address 0, 31 delay
```

Listing 2: Full blinky program running on state machine 0

6 Limitations & Future work

The openPIO project was made in a 4-month time frame as a master's thesis. As it started from scratch, it was not possible to implement all features of the RP2040 PIO in this time frame. Here is a list of features that are planned for future work, before an ASIC tape-out is theasable:

- **Write more test programs:** Apart from the existing blinky programs, it would be interesting so see more test programs. For example, a simple handshake protocol between the openPIO and the RP2040's PIO.
- **Interrupt Request (IRQ):** The design currently doesn't support the IRQ Instruction and its infrastructure. The corresponding control registers, a IRQ condition in the WAIT instruction, and similar things are part of the missing IRQ feature.
- **FIFOs:** The FIFOs, that are used for inbound and outbound data between CPU and PIO are missing. They, together with the feature to join them together.
- **Control Registers:** Some control registers are not yet implemented. For example `SMn_ADDR`, used to be read from the master device to know, what address any state machine is currently executing. Others are debug and status registers, that inform the master about the current state of the PIO.

From there on, an ASIC implementation could be made. EPFL's X-HEEP project [9] could be the perfect target for this implementation, as it is an open-source SoC project, that is currently being developed at EPFL.

While programming the openPIO, it was also noticed, that the PIO's instruction set is not fully utilized. There are still a few reserved op-code configurations, that could be used for additional instructions.

7 Conclusion

This thesis set out to design, implement, and verify *openPIO*, a cycle-accurate, open-source HDL re-implementation of the RP2040's PIO. We:

- Delivered a specifications-driven Verilog core faithful to the RP2040 datasheet.
- Built a comprehensive cocotb[2] + ModelSim[3] test suite.
- Demonstrated FPGA prototyping on a PYNQ-Z2 at 62.5 MHz, projecting 125 MHz in ASIC given shorter fixed interconnects.

Remaining work includes implementing the IRQ instruction, multi-entry FIFOs, and completing all control-registers. With these extensions, openPIO will provide a robust, low-cost programmable I/O engine for future microcontroller and SoC designs.

Acknowledgements

I would like to thank Delphine Allimann for their invaluable support in the wrapping of openPIO into the IPBus project.

Furthermore, I thank Ludovic Blanc for running a synthesis on 65nm TSMC of my project, to estimate the area and timing. Ludovic, together with Christoph Müller, also provided valuable support through the project, as my co-advisors.

References

[1] Raspberry Pi Foundation, *Rp2040 datasheet*, 2021. [Online]. Available: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf (visited on 05/30/2025).

[2] The cocotb developers, *cocotb: coroutine based cosimulation testbenches*, 2020. [Online]. Available: https://www.cocotb.org/ (visited on 06/03/2025).

[3] Siemens EDA, *ModelSim HDL Simulator*. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/modelsim/ (visited on 06/03/2025).

[4] Ismael Tobias Frei, *OpenPIO: an open-source Hardware implementation of the RP2040 PIO*. [Online]. Available: https://tclgit.epfl.ch/semester-projects/25s-frei-open_source_pio (visited on 06/09/2025).

[5] TUL, *Pynq-z2 fpga datasheet*, 2018. [Online]. Available: https://dpoauwgwqsy2x.cloudfront.net/Download/pynqz2_user_manual_v1_0.pdf (visited on 05/30/2025).

[6] CERN and the IPbus collaboration, *IPbus - A Flexible Ethernet-based Control System*. [Online]. Available: https://ipbus.web.cern.ch/ (visited on 06/03/2025).

[7] D. Allimann, *Ethernet ipbus project*, Accessed: 2025-06-18, 2025. [Online]. Available: https://tclgit.epfl.ch/semester-projects/24w-allimann-ethernet_ipbus.

[8] A. Forencich, *Verilog Ethernet Components*, 2025. [Online]. Available: https://github.com/alexforencich/verilog-ethernet (visited on 06/06/2025).

[9] EPFL X-HEEP Team, *X-HEEP: an open-hardware RISC-V system-on-chip*. [Online]. Available: https://x-heep.epfl.ch/ (visited on 06/09/2025).