VHDL Project:
Simple RISC-V CPU Design
+
CORDIC Module

Project and report made by:

Matteo Cornacchia

Version: 1.1
Last review: February 11, 2026

Contents

Chapter 1

Introduction

This project has been entirely developed in VHDL for educational and learning purposes. The work is structured around three main components:

- Design and implementation of a basic RISC-V CPU architecture;
- Implementation of a CORDIC algorithm for sine and cosine computation;
- Hardware–software interaction for the CORDIC module.

Each part of the project is addressed in a dedicated section, where the adopted design choices and implementation details are discussed in greater depth.

The main tools and platforms used in this project are:

- Xilinx Vivado as the development environment for VHDL programming;
- Nexys 4 DDR board for testing and validating part of the system on real hardware.

This report primarily focuses on the logical design and architectural aspects of the developed components. For a complete view of the implementation details, the reader is referred to the project repository available on GitHub. The entire codebase is extensively commented in order to improve readability for first-time readers and to facilitate future maintenance and revisions.

To clarify the structure of the project and to facilitate navigation across its different components, the following project tree provides an overview of all the files included in the repository.

```
src/                              # MAIN STAGES
|-- cpu_top.vhd                   # Datapath (top level for CPU)
|   |-- if_stage.vhd              # Instruction Fetch (IF)
|   |   `-- instruction_memory.vhd
|   |-- id_stage.vhd              # Instruction Decode (ID)
|   |   |-- register_file.vhd
|   |   `-- decoder.vhd
|   |-- ex_stage.vhd              # Instruction Execute (EX)
|   |   |-- alu.vhd
|   |   `-- comparator.vhd
|   |-- mem_stage.vhd             # Data Memory (MEM)
|   `-- wb_stage.vhd              # Write Back (WB)
|       `-- data_mem.vhd
`-- top_nexys.vhd                 # Top level for Nexys 4 DDR (CPU +
   CORDIC)
    |-- cordic_wrapper.vhd        # CORDIC Wrapper
    |   `-- cordic_core.vhd       # CORDIC Algorithm
    |-- debounce_pulse.vhd        # Debounce + Pulse Generator for
       pushbuttons
    `-- seven_seg_driver.vhd      # 7-segment display driver


tb/                               # TESTBENCHES
|-- tb_if_stage.vhd               # Testbench for IF stage
|-- tb_id_stage.vhd               # Testbench for ID stage
|-- tb_ex_stage.vhd               # Testbench for EX stage
|-- tb_mem_wb_stage.vhd           # Testbench for MEM + WB stages
|-- tb_cpu_top.vhd                # Testbench for CPU datapath
|-- tb_cordic_core.vhd            # Testbench for CORDIC core
`-- tb_cordic_wrapper.vhd         # Testbench for CORDIC wrapper


constr/                           # CONSTRAINTS
`-- nexys4ddr.xdc                 # Constraints file for Nexys 4 DDR
```

Listing 1.1: Project tree of the repository.

Chapter 2

RISC-V CPU

This section provides a description of the CPU architecture and the functionality of its main modules. The RISC-V CPU design follows a classical five-stage organization:

- Instruction Fetch (IF);
- Instruction Decode (ID);
- Execute (EX);
- Data Memory access (MEM);
- Write Back (WB).

It is important to note that the implemented design corresponds to a basic, non-pipelined architecture: each instruction traverses all stages sequentially, and the complete datapath is executed once per instruction. A more advanced implementation would include a pipelined architecture, which would require the introduction of pipeline registers between stages as well as the management of hazards (e.g., data hazards and control hazards).

This functionality has not been implemented in the current version; however, due to the modular structure of the project, the introduction of pipelining and hazard handling is feasible and can be considered as a possible future extension of the design.

The following sections provide a general overview of the role of the main components of the CPU. This document does not aim to cover the full set of technical details and specifications of the RISC-V architecture. For a complete and authoritative reference, the reader is referred to the official RISC-V documentation available on the RISC-V website.

2.1 Instruction Fetch

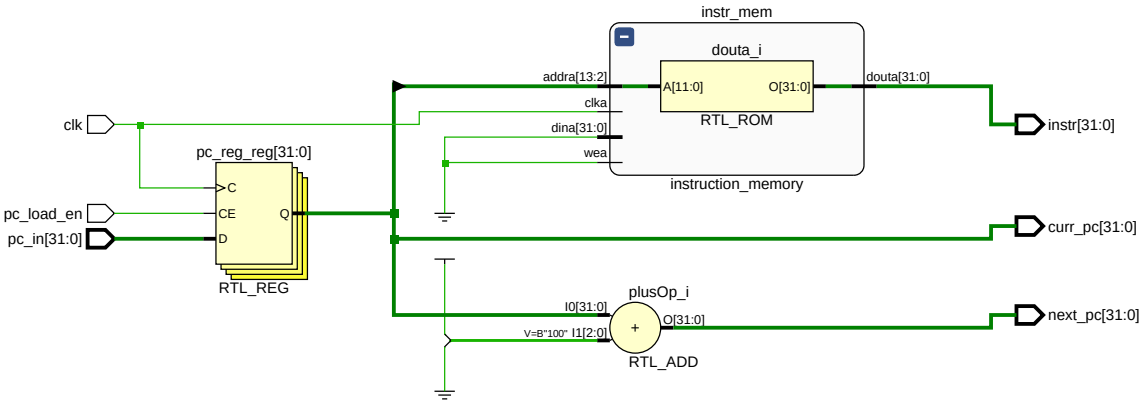

Figure 2.1: High-level overview of the Instruction Fetch stage as rendered by Vivado.

The Instruction Fetch (IF) stage manages the Program Counter, retrieves the current instruction from the instruction memory, and computes the address of the next sequential instruction. In the adopted single-cycle architecture, this stage enables a straightforward sequential execution model and provides the basis for future support of control-flow instructions such as branches and jumps.

The Program Counter is updated synchronously with the clock and only when the enable signal is asserted. The next instruction address is computed by adding four bytes to the current Program Counter value, while an externally provided target address can be loaded when required. The current Program Counter value and the fetched instruction are exposed to the subsequent stages of the processor.

For clarity, Listing 2.1 reports the interface of the Instruction Fetch stage, highlighting the main input and output signals.

```
1  entity if_stage is
2      port (
3          clk        : in std_logic;
4          pc_load_en : in std_logic;                     -- enables PC loading
5          pc_in      : in std_logic_vector(31 downto 0); -- value to load into PC
6
7          next_pc    : out std_logic_vector(31 downto 0); -- PC + 4
8          curr_pc    : out std_logic_vector(31 downto 0); -- current PC
9          instr      : out std_logic_vector(31 downto 0) -- fetched instruction
10     );
11 end if_stage;
```

Listing 2.1: Entity declaration of the Instruction Fetch stage.

The Instruction Fetch stage has been validated using a dedicated testbench. During simulation, the fetched instructions are provided by the instruction memory module, which is initialized with a small test program. This allows the verification of the correct instruction fetch sequence and the sequential progression of the Program Counter.

The instruction memory used in the testbench contains the following test instructions, selected to exercise basic arithmetic and memory access operations.

```
signal rom : rom_t := (
    0 => x"00500093", -- addi x1, x0, 5
    1 => x"00A00113", -- addi x2, x0, 10
    2 => x"002081B3", -- add x3, x1, x2
    3 => x"00302023", -- sw x3, 0(x0)
    4 => x"00002203", -- lw x4, 0(x0)
    others => (others => '0')
);
```

Listing 2.2: Initialization of the instruction memory used for simulation.

The testbench verifies the correct update of the Program Counter when enabled and the proper increment of the instruction address during sequential execution. The corresponding simulation results are reported in Figure 2.2.

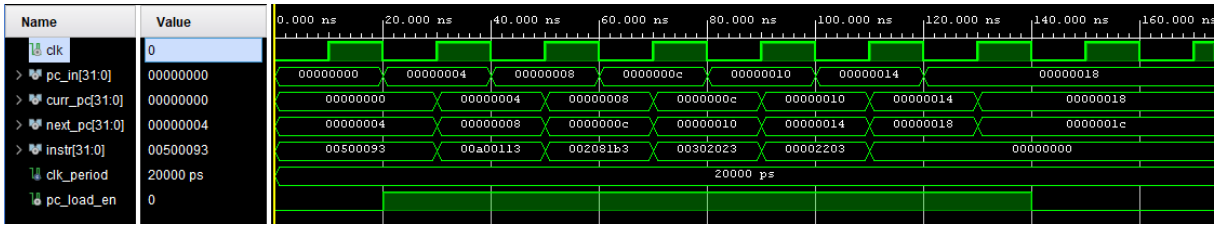

Figure 2.2: Simulation results of the Instruction Fetch testbench as rendered by Vivado.

2.2 Instruction Decode

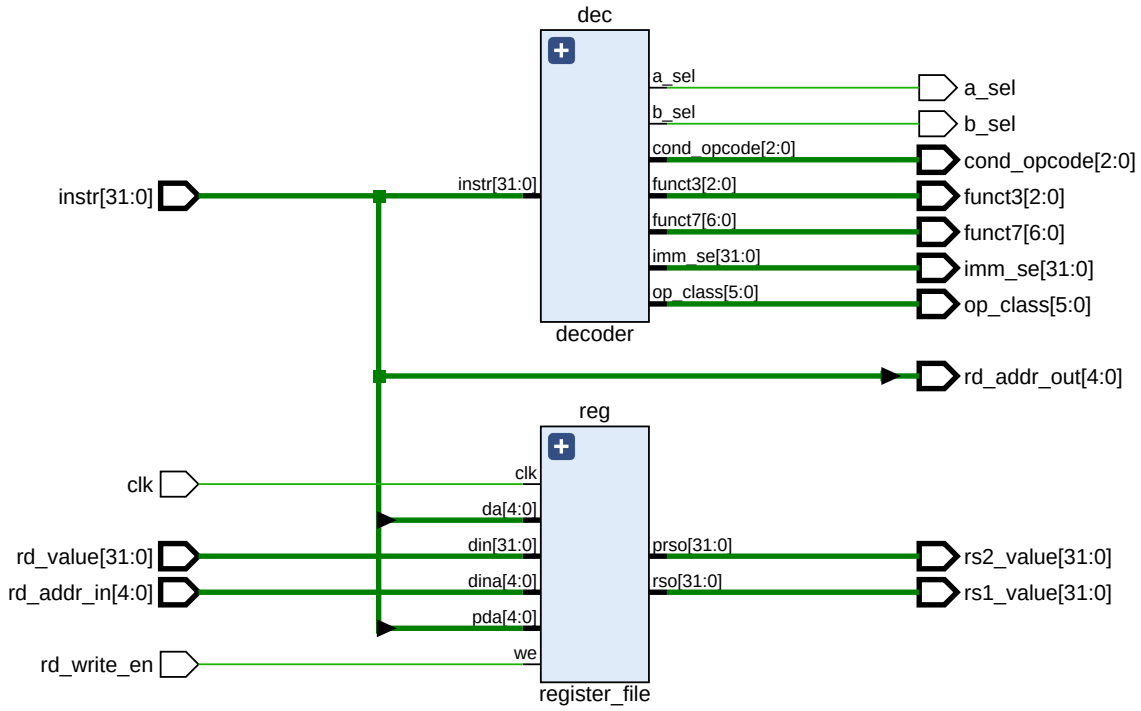

Figure 2.3: High-level overview of the Instruction Decode stage as rendered by Vivado.

The Instruction Decode (ID) stage is responsible for decoding the fetched instruction and for preparing all the information required by the subsequent execution stage. In particular, this stage extracts the opcode and function fields, selects the appropriate control signals, reads the source operands from the register file, and generates the sign-extended immediate value when required. The ID stage also handles the write-back interface, allowing register updates from the MEM/WB stage.

A high-level view of the Instruction Decode stage as synthesized and rendered by Vivado is shown in Figure 2.3. The diagram highlights the main submodules involved in instruction decoding, register file access, and immediate generation.

The interface of the Instruction Decode stage is reported in Listing 2.3. The entity exposes the fetched instruction as input, the write-back interface from the MEM/WB stage, and a set of decoded control signals and operands to be forwarded to the execution stage.

```
1  entity id_stage is
2      port (
3          clk        : in std_logic;
4          instr      : in std_logic_vector(31 downto 0);
5
6          -- Writeback interface (from MEM/WB stage)
7          rd_write_en : in std_logic;
8          rd_value    : in std_logic_vector(31 downto 0);
9          rd_addr_in : in std_logic_vector(4 downto 0);
10
```

```
        -- Decoded instruction information
        op_class   : out std_logic_vector(5 downto 0);
        funct3     : out std_logic_vector(2 downto 0);
        funct7     : out std_logic_vector(6 downto 0);
        a_sel      : out std_logic;
        b_sel      : out std_logic;
        cond_opcode : out std_logic_vector(2 downto 0);
        rd_addr_out : out std_logic_vector(4 downto 0);

        -- Operand values and immediate
        rs1_value  : out std_logic_vector(31 downto 0);
        rs2_value  : out std_logic_vector(31 downto 0);
        imm_se     : out std_logic_vector(31 downto 0)
    );
end id_stage;
```

Listing 2.3: Entity declaration of the Instruction Decode stage.

The Instruction Decode stage has been verified through a dedicated testbench. The testbench initializes a register value through the write-back interface and then applies a sequence of instructions in order to validate correct operand reading, immediate generation, and instruction decoding. Listing 2.4 reports an excerpt of the stimulus process used for verification.

```
-- Stimulus
stim_proc : process
begin
    -- ====================================
    -- Write x1 = 0x00000010
    -- ====================================
    rd_write_en <= '1';
    rd_addr_in <= "00001";
    rd_value   <= x"00000010";
    wait for clk_period;
    rd_write_en <= '0';

    -- ====================================
    -- ADDI x2, x1, 4
    -- ====================================
    instr <= x"00408113"; -- addi x2, x1, 4
    wait for clk_period;

    -- ====================================
    -- ADD x3, x1, x2
    -- ====================================
    instr <= x"002081B3"; -- add x3, x1, x2
    wait for clk_period;

    wait;
end process;
```

Listing 2.4: Excerpt of the stimulus process used in the Instruction Decode testbench.

The simulation results obtained from the testbench are shown in Figure 2.4. The waveforms confirm the correct decoding of the instruction fields, the proper reading of the source registers, and the correct generation of the immediate values for immediate-type instructions.

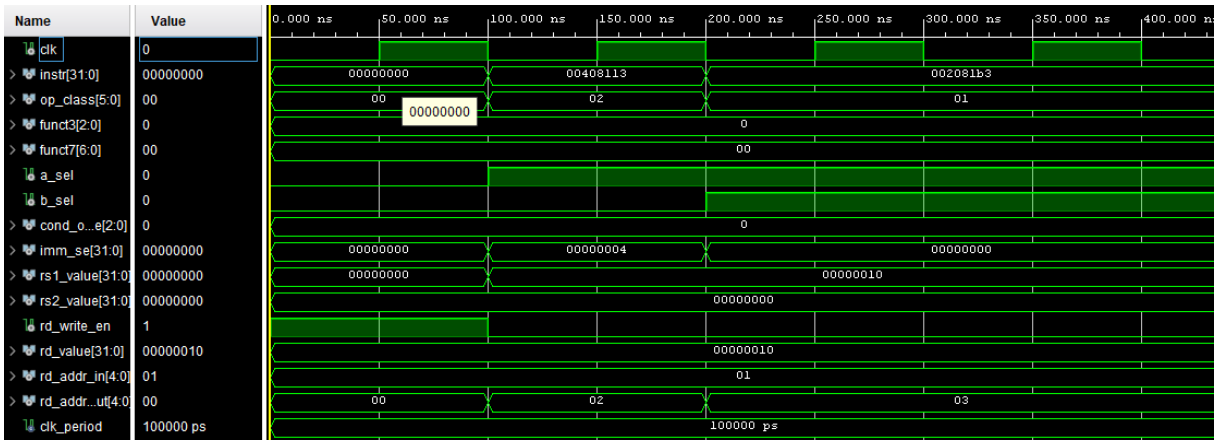

Figure 2.4: Simulation results of the Instruction Decode testbench as rendered by Vivado.

2.3 Instruction Execute

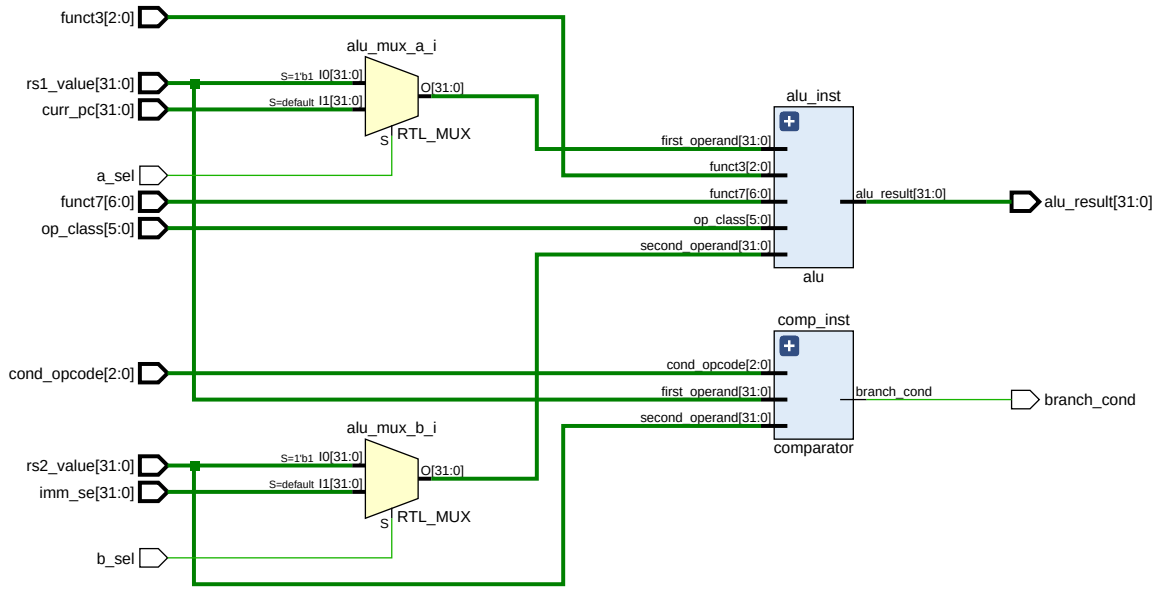

Figure 2.5: High-level overview of the Instruction Execute stage as rendered by Vivado.

The Instruction Execute (EX) stage is responsible for performing arithmetic and logical operations, computing branch conditions, and generating the intermediate results required by the subsequent stages of the datapath. This stage represents the computational core of the processor.

The EX stage includes two main functional units: an Arithmetic Logic Unit (ALU) and a comparator. The ALU performs arithmetic and logical operations such as addition, subtraction, and immediate-based computations. The comparator evaluates conditional expressions used by branch instructions, such as equality and signed or unsigned comparisons, and generates the corresponding branch condition signal. Based on the decoded instruction and control signals, the EX stage selects the appropriate operands and computes the final execution result.

A high-level view of the Instruction Execute stage as synthesized and rendered by Vivado is shown in Figure 2.5. The diagram highlights the integration of the ALU, the comparator, and the operand selection logic.

The interface of the Instruction Execute stage is reported in Listing 2.5. The entity exposes control signals for operand selection, the decoded instruction fields, and the data operands required by the ALU and comparator. The outputs include the computed ALU result and the evaluated branch condition.

```
1  entity ex_stage is
2      port (
3          -- Control signals
4          a_sel          : in std_logic; -- 0 = PC, 1 = rs1
5          b_sel          : in std_logic; -- 0 = imm, 1 = rs2
6
7          -- Data inputs
```

```
        rs1_value      : in std_logic_vector(31 downto 0);
        rs2_value      : in std_logic_vector(31 downto 0);
        imm_se         : in std_logic_vector(31 downto 0);
        curr_pc        : in std_logic_vector(31 downto 0);

        -- Decode information
        cond_opcode    : in std_logic_vector(2 downto 0);
        funct3         : in std_logic_vector(2 downto 0);
        funct7         : in std_logic_vector(6 downto 0);
        op_class       : in std_logic_vector(5 downto 0);

        -- Outputs
        branch_cond    : out std_logic;
        alu_result     : out std_logic_vector(31 downto 0)
    );
end ex_stage;
```

Listing 2.5: Entity declaration of the Instruction Execute stage.

The Instruction Execute stage has been validated using a dedicated testbench that applies a set of representative instructions and input operands. The testbench verifies both arithmetic operations and branch condition evaluation, including register-based operations, immediate-based operations, and PC-relative computations. Listing 2.6 reports an excerpt of the stimulus process used for verification.

```
-- Stimulus process
stim_proc : process
begin
    -- ====================================
    -- ADD: x1 + x2
    -- ====================================
    rs1_value <= x"00000005";
    rs2_value <= x"00000003";
    a_sel    <= '1';
    b_sel    <= '1';
    op_class <= "000001"; -- OP
    funct3   <= "000";
    funct7   <= "0000000";
    wait for 20 ns;

    -- ====================================
    -- SUB: x1 - x2
    -- ====================================
    funct7 <= "0100000";
    wait for 20 ns;

    -- ====================================
    -- ADDI: x1 + imm
    -- ====================================
    b_sel   <= '0';
    imm_se  <= x"00000004";
    op_class <= "000010"; -- OP-IMM
    funct7  <= "0000000";
```

```
    wait for 20 ns;

    -- ====================================
    -- AUIPC: PC + imm
    -- ====================================
    a_sel   <= '0';
    curr_pc <= x"00001000";
    imm_se  <= x"00000010";
    op_class <= "010010";
    wait for 20 ns;

    -- ====================================
    -- BEQ (true)
    -- ====================================
    rs1_value  <= x"0000000A";
    rs2_value  <= x"0000000A";
    cond_opcode <= "000"; -- BEQ
    op_class   <= "100000"; -- BRANCH
    wait for 20 ns;

    -- ====================================
    -- BLT (signed, false)
    -- ====================================
    rs1_value  <= x"00000005";
    rs2_value  <= x"00000002";
    cond_opcode <= "100"; -- BLT
    wait for 20 ns;

    wait;
end process;
```

Listing 2.6: Excerpt of the stimulus process used in the Instruction Execute testbench.

The simulation results obtained from the testbench are shown in Figure 2.6. The waveforms confirm the correct computation of ALU results and the correct evaluation of branch conditions for the tested instruction set.

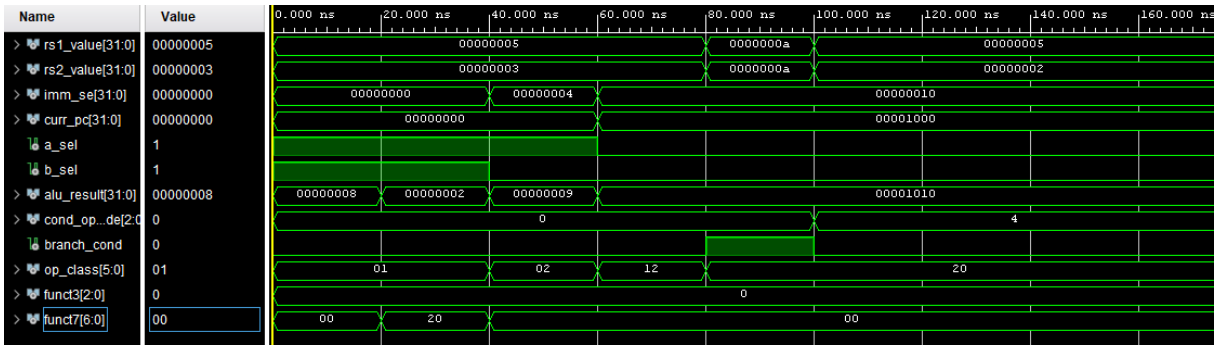

Figure 2.6: Simulation results of the Instruction Execute testbench as rendered by Vivado.

2.4 Data Memory

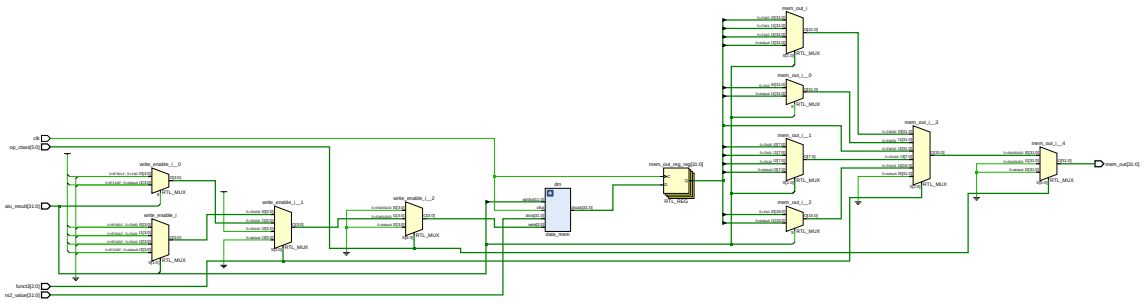

Figure 2.7: High-level overview of the Data Memory stage as rendered by Vivado.

The Data Memory (MEM) stage is responsible for handling memory access operations generated by load and store instructions. In this stage, the address computed by the execution stage is used to access the data memory, either to read a value from memory (load operations) or to write a value to memory (store operations). This stage therefore provides the interface between the CPU core and the data memory subsystem. The write-back of loaded values to the register file is handled in the subsequent stage.

In the current design, the Data Memory stage instantiates a dedicated memory module, referred to as data_mem, which implements the actual storage element. The MEM stage is responsible for decoding the operation type and for driving the appropriate control signals to the memory module.

A high-level view of the Data Memory stage as synthesized and rendered by Vivado is shown in Figure 2.7.

The interface of the Data Memory stage is reported in Listing 2.7. The entity receives the operation class and function fields required to identify the memory operation, the data to be written in case of store instructions, and the address computed by the execution stage. The output provides the value read from memory in case of load instructions.

```
entity mem_stage is
    port (
        clk        : in std_logic;
        op_class   : in std_logic_vector(5 downto 0);
        funct3     : in std_logic_vector(2 downto 0);
        rs2_value  : in std_logic_vector(31 downto 0);
        alu_result : in std_logic_vector(31 downto 0);

        mem_out    : out std_logic_vector(31 downto 0)
    );
end mem_stage;
```

Listing 2.7: Entity declaration of the Data Memory stage.

The actual memory storage is implemented by the data_mem module, whose interface is reported in Listing 2.8. This module provides a simple synchronous write and asynchronous read interface and is accessed using the address generated by the execution stage.

```
entity data_mem is
    port(
        clka  : in std_logic;
        wea   : in std_logic_vector(3 downto 0);
        dina  : in std_logic_vector(31 downto 0);
        addra : in std_logic_vector(11 downto 0);

        douta : out std_logic_vector(31 downto 0)
    );
end data_mem;
```

Listing 2.8: Entity declaration of the data memory module.

2.5 Write Back

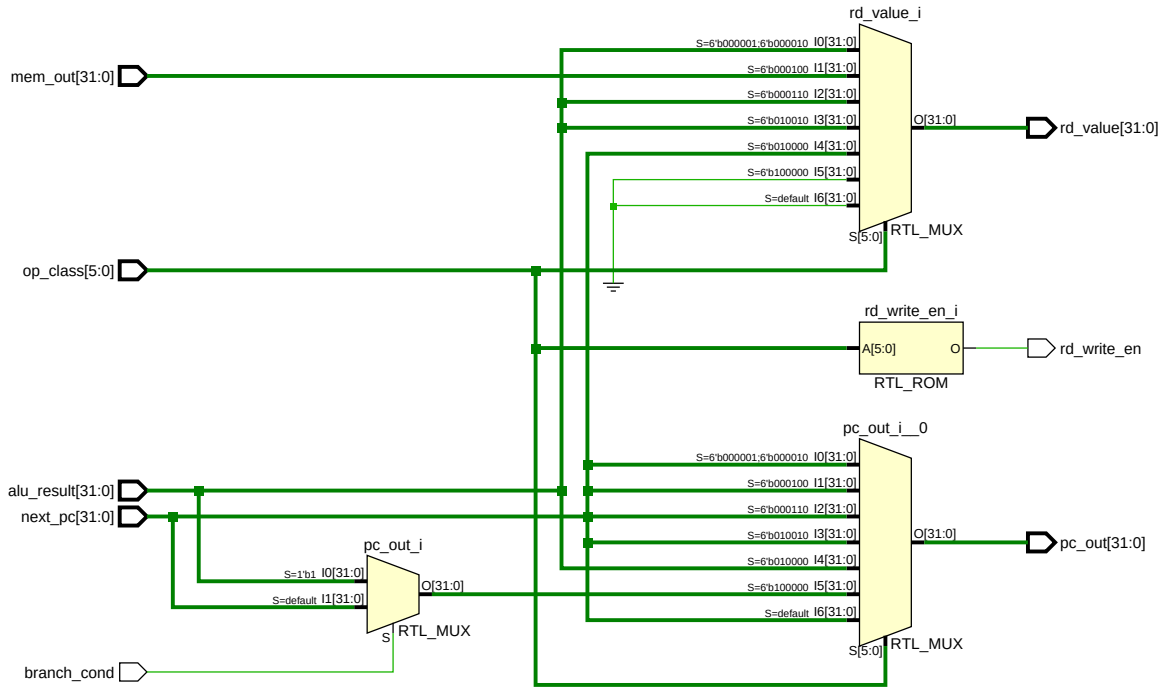

Figure 2.8: High-level overview of the Write Back stage as rendered by Vivado.

The Write Back (WB) stage is responsible for completing the execution of an instruction by writing the final result back to the register file and by updating the Program Counter when required. This stage selects the correct result source depending on the instruction type, such as the ALU output for arithmetic and logical operations or the memory output for load instructions. In addition, the WB stage handles control-flow updates by providing the next Program Counter value in case of branch instructions.

A high-level view of the Write Back stage as synthesized and rendered by Vivado is shown in Figure 2.8. The diagram highlights the selection logic used to route the correct result to the register file and to determine the next value of the Program Counter.

The interface of the Write Back stage is reported in Listing 2.9. The entity receives the execution results and control information from the previous stages and produces the control signals required to update the register file and the Program Counter.

```
1  entity wb_stage is
2      port (
3          branch_cond : in std_logic;
4          op_class    : in std_logic_vector(5 downto 0);
5          mem_out     : in std_logic_vector(31 downto 0);
6          next_pc     : in std_logic_vector(31 downto 0);
7          alu_result  : in std_logic_vector(31 downto 0);
8
9          rd_write_en : out std_logic;
10         pc_out      : out std_logic_vector(31 downto 0);
11         rd_value    : out std_logic_vector(31 downto 0)
```

```
    );
end wb_stage;
```

Listing 2.9: Entity declaration of the Write Back stage.

The combined behavior of the Data Memory and Write Back stages has been verified through a dedicated testbench. The testbench applies a sequence of store and load instructions in order to validate correct memory accesses and proper forwarding of the loaded values to the write-back stage. Listing 2.10 reports an excerpt of the stimulus process used for verification.

```
-- Stimulus
stim_proc : process
begin
    -- ================================
    -- SW: store word
    -- ================================
    op_class  <= "001000"; -- STORE
    funct3    <= "010";   -- SW
    alu_result <= x"00000000";
    rs2_value <= x"DEADBEEF";
    wait for clk_period;

    -- ================================
    -- LW: load word
    -- ================================
    op_class  <= "000100"; -- LOAD
    funct3    <= "010";   -- LW
    alu_result <= x"00000000";
    wait for clk_period;

    -- ================================
    -- LB (signed)
    -- ================================
    funct3    <= "000";
    alu_result <= x"00000000";
    wait for clk_period;

    -- ================================
    -- LBU (unsigned)
    -- ================================
    funct3    <= "100";
    wait for clk_period;

    wait;
end process;
```

Listing 2.10: Excerpt of the stimulus process used in the MEM/WB testbench.

The simulation results obtained from the MEM/WB testbench are shown in Figure 2.9. The waveforms confirm the correct handling of store and load operations, as well as the proper forwarding of memory data to the write-back stage.

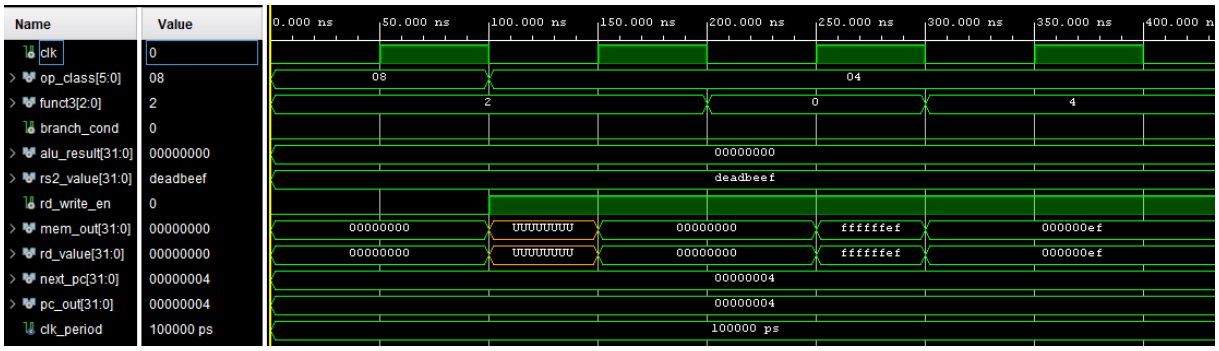

Figure 2.9: Simulation results of the Data Memory and Write Back testbench as rendered by Vivado.

2.6 Datapath

This section describes the datapath module, which represents the top-level integration of the CPU core. The datapath is implemented as a dedicated VHDL file whose purpose is to interconnect all the previously described stages, namely Instruction Fetch, Instruction Decode, Execute, Data Memory, and Write Back.

The datapath module defines the complete flow of data and control signals across the processor, connecting the output of each stage to the input of the subsequent one. In this way, it establishes the full execution path from instruction fetch to write back, enabling the coordinated operation of all stages within the single-cycle CPU architecture.

The datapath has been verified using a dedicated testbench that executes a small test program stored in the instruction memory. The sequence of instructions used during simulation is written in Listing 2.2.

These instructions allow the validation of immediate operations, register-to-register arithmetic, memory store operations, and memory load operations within the complete datapath.

The simulation results of the datapath testbench are shown in Figure 2.10.

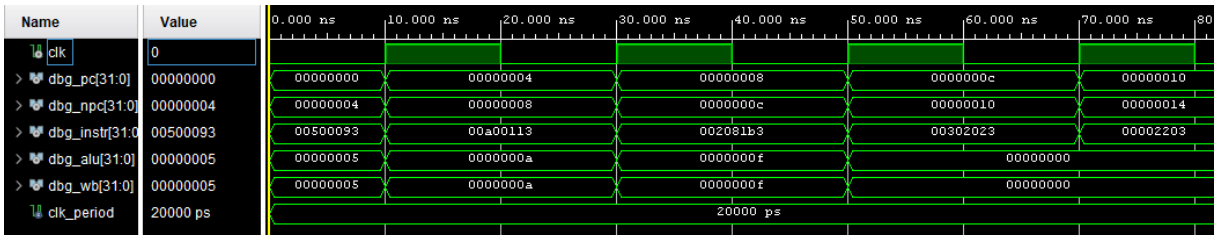

Figure 2.10: Simulation results of the datapath testbench as rendered by Vivado.

During simulation, the values observed at the output of the ALU and at the Write Back stage are reported below:

Value	Decimal	Source
5	5	addi x1, x0, 5
A	10	addi x2, x0, 10
F	15	add x3, x1, x2 = 5 + 10
0	0	address computation for sw and lw (x0 + 0)

The ALU and Write Back output signals exhibit the same values during execution because the implemented architecture follows a single-cycle model. In each clock cycle, the result produced by the ALU is immediately propagated to the Write Back stage without intermediate pipeline registers. As a consequence, the values observed at the ALU output and at the Write Back interface coincide for the executed instructions.

The observed values (5, 10, 15, and 0) correspond respectively to the two immediate additions, the register-to-register addition, and the address computation used by the store and load instructions. This behavior confirms the correct propagation of data through the datapath and validates the functional correctness of the integrated CPU architecture.

Chapter 3

CORDIC

The CORDIC algorithm is an iterative method used to compute trigonometric functions such as sine and cosine, as well as other elementary functions, using only shift and add operations. Due to its low hardware complexity, CORDIC is particularly well suited for digital hardware implementations, including FPGA-based designs.

In this report, the internal mathematical details of the CORDIC algorithm are not discussed in depth. For an intuitive explanation and a visual overview of the underlying principles, the reader is referred to this video by Bobater on YouTube.

Within this project, the CORDIC functionality is implemented through two main modules. The first module corresponds to the core CORDIC unit, which performs the iterative angle computation. The second module acts as a wrapper, and is responsible for handling sign corrections and quadrant mapping in order to extend the computation to the full trigonometric domain.

In the current implementation, the CORDIC module operates as a standalone component and is not directly integrated into the CPU datapath. The module can be controlled and tested independently from the processor core. However, due to the modular structure of the overall design, the CORDIC unit can be integrated as part of the CPU in future extensions, for example as a dedicated functional unit or coprocessor.

3.1 CORDIC Core

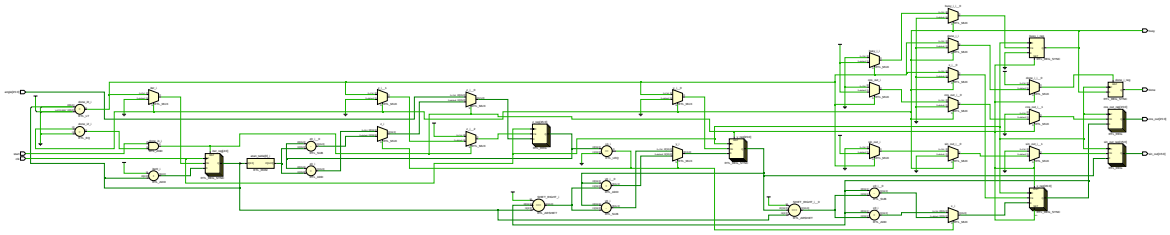

Figure 3.1: High-level overview of the CORDIC core as rendered by Vivado.

The CORDIC core implements the core computational engine used to evaluate trigonometric functions. Given an input angle within the range $[-\pi/2, +\pi/2]$, the module computes the corresponding sine and cosine values using an iterative shift-and-add algorithm. This approach is well suited for hardware implementations, as it avoids the use of multipliers and relies only on additions, subtractions, and arithmetic shifts.

The module operates in rotation mode and uses fixed-point arithmetic to represent both the input angle and the output values. The input angle is represented using a signed fixed-point format, while the sine and cosine outputs are provided in a normalized fixed-point representation. The number of iterations and the numerical precision are configurable through generic parameters, allowing a trade-off between accuracy and latency.

The interface of the CORDIC core is reported in Listing 3.1. The entity exposes the control signals required to start the computation, the input angle, and the output signals carrying the computed sine and cosine values, together with status flags indicating the progress and completion of the operation.

```
entity cordic_core is
    generic (
        WIDTH     : integer := 16;
        ITERATIONS : integer := 16
    );
    port (
        clk     : in  std_logic;
        start   : in  std_logic;
        angle   : in  signed(WIDTH-1 downto 0); -- Q2.(WIDTH-2)

        busy    : out std_logic;
        done    : out std_logic;

        cos_out : out signed(WIDTH-1 downto 0); -- Q1.(WIDTH-1)
        sin_out : out signed(WIDTH-1 downto 0) -- Q1.(WIDTH-1)
    );
end cordic_core;
```

Listing 3.1: Entity declaration of the CORDIC core.

Internally, the core maintains three state variables representing the current vector components and the residual angle. At each iteration, the algorithm updates these values

according to the sign of the residual angle, progressively rotating the vector towards the target angle. A constant scaling factor is applied to compensate for the intrinsic gain introduced by the CORDIC iterations. The arctangent values required by the algorithm are stored in a precomputed lookup table.

The computation is initiated by asserting a start signal. While the computation is in progress, the core signals a busy condition. Once all iterations have been completed, the resulting sine and cosine values are produced at the outputs and a done signal is asserted for one clock cycle. This handshake-based interface simplifies the integration of the CORDIC core within larger systems, such as the wrapper and the board-level top module.

The correct functionality of the CORDIC core has been verified through a dedicated testbench. In order to improve the readability of the results, the fixed-point sine and cosine outputs are converted to decimal values during simulation.

Figure 3.2 shows the simulation results obtained for a set of representative input angles, namely 15°, 30°, 45°, and 60°.

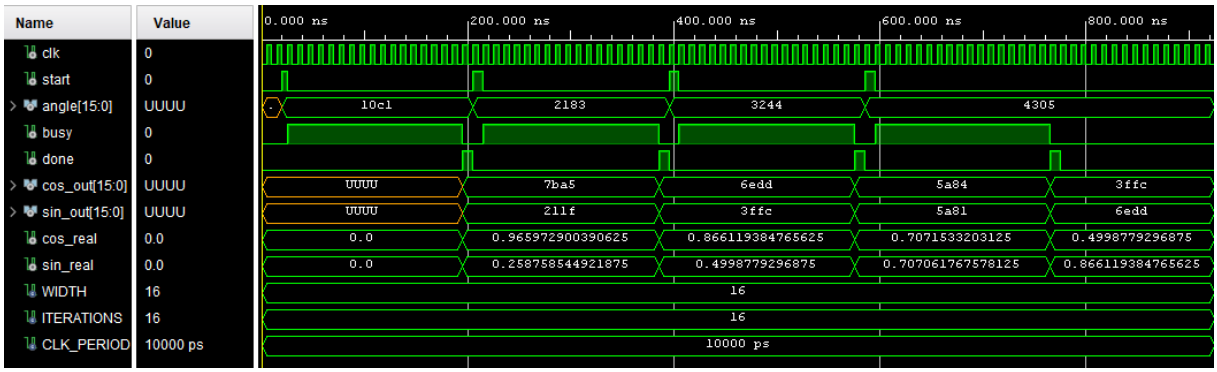

Figure 3.2: Simulation results of the CORDIC core testbench as rendered by Vivado.

The obtained results match the expected values with good accuracy. However, a limitation of the current implementation must be highlighted. The CORDIC algorithm exhibits incorrect sign behavior for specific edge cases, in particular for angles equal to zero or exact multiples of $\pi/2$. While the computed magnitude of sine and cosine is correct, the sign of one or both outputs may be incorrect in these boundary conditions.

This behavior is related to the iterative nature of the CORDIC rotation mode. Even for angles that theoretically require no rotation (such as zero), the algorithm still performs a sequence of micro-rotations. Due to finite precision and the sign-based update rule, these iterations can converge to a value with the correct magnitude but an incorrect sign in specific corner cases.

A possible workaround would consist in explicitly detecting these critical angles and overriding the output values with predefined results. However, such an approach would partially compromise the purely iterative nature of the CORDIC algorithm and introduce special-case handling logic. For this reason, this limitation is documented but not addressed in the current implementation.

3.2 CORDIC Wrapper

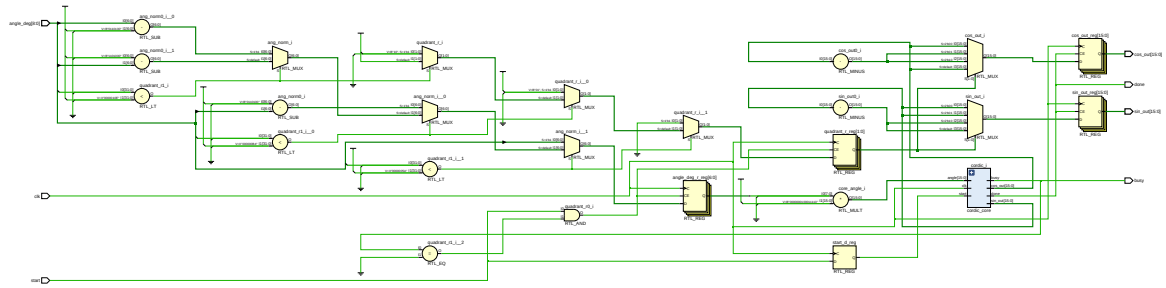

Figure 3.3: High-level overview of the CORDIC wrapper as rendered by Vivado.

The CORDIC wrapper extends the functionality of the CORDIC core in order to support the full trigonometric domain. While the CORDIC core operates correctly only for input angles within the range $[-\pi/2, +\pi/2]$, the wrapper enables the computation of sine and cosine for angles spanning the full circle. This is achieved by mapping the input angle to the corresponding equivalent angle within the valid range of the core and by applying the appropriate sign corrections based on the quadrant of the original angle.

The wrapper therefore performs angle normalization, quadrant identification, and output sign adjustment, while delegating the actual trigonometric computation to the underlying CORDIC core. This approach preserves the modularity of the design and allows the core algorithm to remain unchanged.

The interface of the CORDIC wrapper is reported in Listing 3.2. The entity receives the input angle expressed in degrees in the range from 0 to 359 and provides the corresponding sine and cosine values in fixed-point format, together with handshake signals indicating the computation status.

```
entity cordic_wrapper is
    generic (
        WIDTH     : integer := 16;
        ITERATIONS : integer := 16
    );
    port (
        clk       : in  std_logic;
        start     : in  std_logic;
        angle_deg : in unsigned(8 downto 0); -- 0..359

        busy      : out std_logic;
        done      : out std_logic;

        cos_out   : out signed(WIDTH-1 downto 0);
        sin_out   : out signed(WIDTH-1 downto 0)
    );
end cordic_wrapper;
```

Listing 3.2: Entity declaration of the CORDIC wrapper.

The wrapper has been validated through a dedicated testbench in which the sine and cosine outputs are converted to decimal values for ease of interpretation. In order to verify the correct handling of angle mapping and quadrant-dependent sign assignment, the following angles have been tested: 45°, 135°, 225°, and 315°.

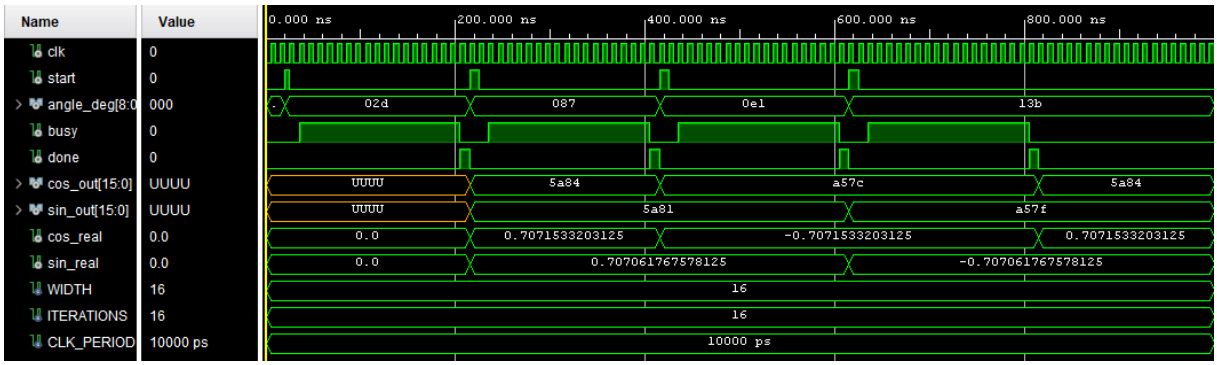

Figure 3.4: Simulation results of the CORDIC wrapper testbench as rendered by Vivado.

As expected, the magnitude of the sine and cosine values remains consistent across the tested angles, while the signs change according to the corresponding quadrant. This behavior confirms the correct implementation of the quadrant mapping and sign correction logic within the wrapper module.

Chapter 4

Nexys 4 DDR

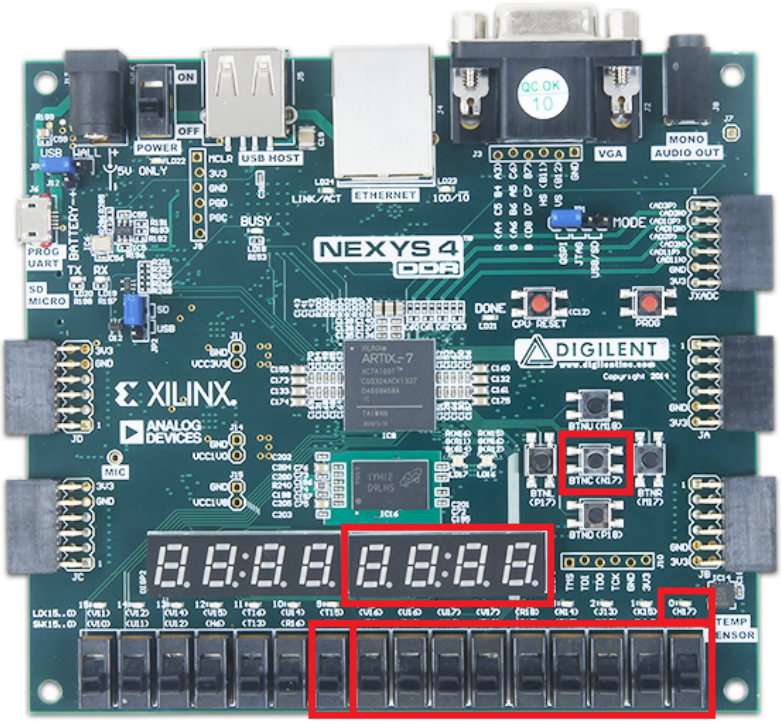

Figure 4.1: Image of the Nexis 4 DDR board with the elements used in the project highlighted.

One of the objectives of this project is to provide an interaction between the Nexys 4 DDR board and the previously developed hardware modules, in particular the CORDIC unit. The board is used as a simple hardware calculator for computing the sine and cosine of a given angle. The user selects the input angle in binary format through the on-board switches and chooses whether to compute the sine or the cosine. Once the input has been configured, the computation is triggered by pressing a push button. The CORDIC module then performs the calculation, and the resulting value is displayed on the seven-segment display available on the board.

This chapter describes all the hardware and control elements required to enable this interaction, including the input interface, the control logic, and the output visualization on the Nexys 4 DDR platform.

4.1 Constraints File

In order to ensure proper communication between the hardware design and the physical Nexys 4 DDR board, it is necessary to define a constraints file. The constraints file (.xdc) specifies how the logical signals of the design are mapped to the physical pins of the FPGA device, enabling the interaction between the implemented hardware modules and the external peripherals available on the board.

A reference template of the constraints file for the Nexys 4 DDR board is provided by Digilent on GitHub. This template can be used as a starting point and adapted to the specific requirements of the project.

For this project, only a subset of the available I/O resources is required. In particular, the signals used to interface with push buttons, LEDs, switches, and the seven-segment display have been constrained and mapped to the corresponding physical pins of the board.

4.2 Debouncer and Pulse Generator

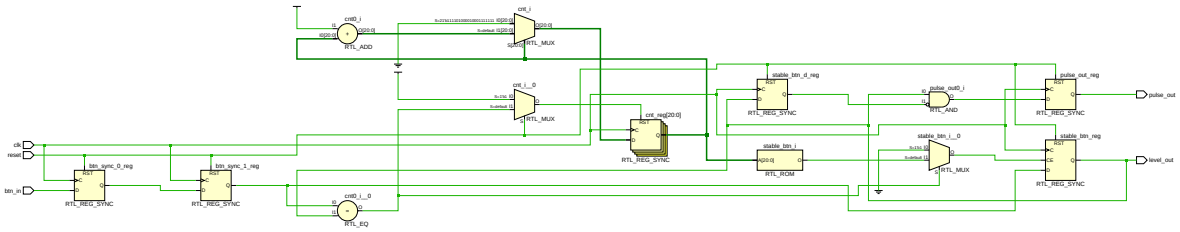

Figure 4.2: High-level overview of the Debouncer and Pulse Generator module as rendered by Vivado.

Mechanical pushbuttons produce noisy signals due to contact bouncing and are not synchronized with the system clock. For this reason, their raw output cannot be used directly to control synchronous hardware modules. The Debouncer and Pulse Generator module provides a clean and reliable interface between the pushbutton inputs of the Nexys 4 DDR board and the digital logic of the system.

The module first synchronizes the asynchronous button signal to the system clock using a two-stage flip-flop synchronizer, reducing the risk of metastability. The synchronized signal is then filtered by a debounce mechanism based on a counter. A new button state is accepted only if the input remains stable for a configurable number of clock cycles, defined by the debounce window parameter. This ensures that short glitches caused by mechanical bouncing are ignored.

Once a stable button level has been detected, a rising-edge detector generates a single-clock-wide pulse whenever the button is pressed. This pulse is suitable for triggering control actions, such as starting a CORDIC computation, without the risk of multiple activations caused by a prolonged button press. In addition to the pulse signal, the module also provides the debounced button level, which can be used for monitoring or control purposes if required.

This module plays a key role in ensuring reliable user interaction with the system, as it guarantees that each physical button press results in a single, well-defined control event.

4.3 7-Segment Display Driver

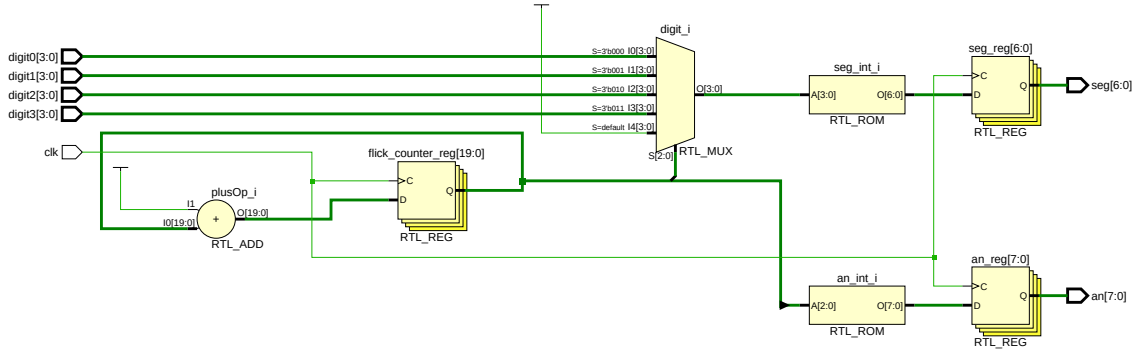

Figure 4.3: High-level overview of the 7-Sefmante Display driver as rendered by Vivado.

In order to correctly visualize numerical values on the seven-segment display of the Nexys 4 DDR board, the output signals generated by the hardware modules must be properly encoded. The 7-segment display driver is responsible for translating a binary digit into the corresponding pattern of segments to be activated on the display.

Each digit to be shown is mapped to a specific combination of the seven LED segments. This mapping is defined through a combinational encoding table, where each possible input value is associated with the corresponding segment activation pattern. The driver therefore acts as an interface between the binary representation of a number and its human-readable visualization on the display.

Listing 4.1 shows an example of the encoding logic used to map a 4-bit digit to the corresponding seven-segment pattern. Invalid or unsupported input values are mapped to a blank display configuration.

```
1  with digit select
2        seg_int <=
3            "1000000" when "0000", -- 0
4            "1111001" when "0001", -- 1
5            "0100100" when "0010", -- 2
6            "0110000" when "0011", -- 3
7            "0011001" when "0100", -- 4
8            "0010010" when "0101", -- 5
9            "0000010" when "0110", -- 6
10           "1111000" when "0111", -- 7
11           "0000000" when "1000", -- 8
12           "0010000" when "1001", -- 9
13           "1111111" when others; -- blank / invalid
```

Listing 4.1: Seven-segment display encoding table for decimal digits.

Figure 4.4: Example of the seven-segment display encoding for decimal digits.

This encoding approach allows the display driver to present the computed results of the CORDIC module in a clear and readable format, enabling direct visual feedback to the user through the on-board display.

4.4 Top Level for Nexys 4 DDR

The top-level module represents the integration layer between the board-specific interface logic and the CORDIC computation modules. Its main purpose is to interconnect the input and output peripherals of the Nexys 4 DDR board with the internal hardware components responsible for the computation and control flow.

This module instantiates and connects the CORDIC core, the input handling logic (including debouncers and control signals derived from pushbuttons and switches), and the output interface modules used to drive the LEDs and the seven-segment display. As a result, it acts as the main coordination point of the system, ensuring that user inputs are correctly interpreted and that the computed results are properly visualized on the board.

Listing 4.2 reports the entity declaration of the top-level module used for the Nexys 4 DDR board. The interface exposes the board clock, the user input signals, and the output signals required to drive the on-board peripherals.

```
entity top_nexys is
    generic (
        WIDTH     : integer := 16;
        ITERATIONS : integer := 16
    );
    port (
        -- Clock from Nexys4 DDR (100 MHz)
        clk100mhz : in std_logic;

        -- User inputs
        sw  : in std_logic_vector(15 downto 0); -- switches
        btn : in std_logic_vector(4 downto 0); -- pushbuttons

        -- User outputs
        led : out std_logic_vector(15 downto 0); -- LEDs
        an  : out std_logic_vector(7 downto 0); -- 7-seg anodes (active low)
        seg : out std_logic_vector(6 downto 0) -- 7-seg segments (g..a, active
            low)
    );
end top_nexys;
```

Listing 4.2: Entity declaration of the Nexys 4 DDR top-level module.

This top-level organization enables a clear separation between board-specific interfaces and computational logic, simplifying both verification and future extensions of the system.

4.5 Interaction with Nexys 4 DDR

Once the board has been programmed, it is possible to interact with the system through the on-board peripherals. The following elements of the Nexys 4 DDR board are used:

- Switches from 0 to 8 are used to select the input angle in binary format. Since the selectable angles range from 0 to 359 degrees, the maximum value to be represented is 359 in decimal, corresponding to 101100111 in binary, which requires 9 bits.
- Switch 9 is used to select the trigonometric function: cosine when the switch is off and sine when the switch is on.
- The central pushbutton (BTNC, N17) is used to start the computation.
- The seven-segment display is used to show the computed result using four decimal digits. The displayed format is N.NNN, where the first digit represents the integer part and the remaining three digits represent the fractional part.

A schematic view of the board elements involved in the interaction and their corresponding roles is shown in Figure 4.5.

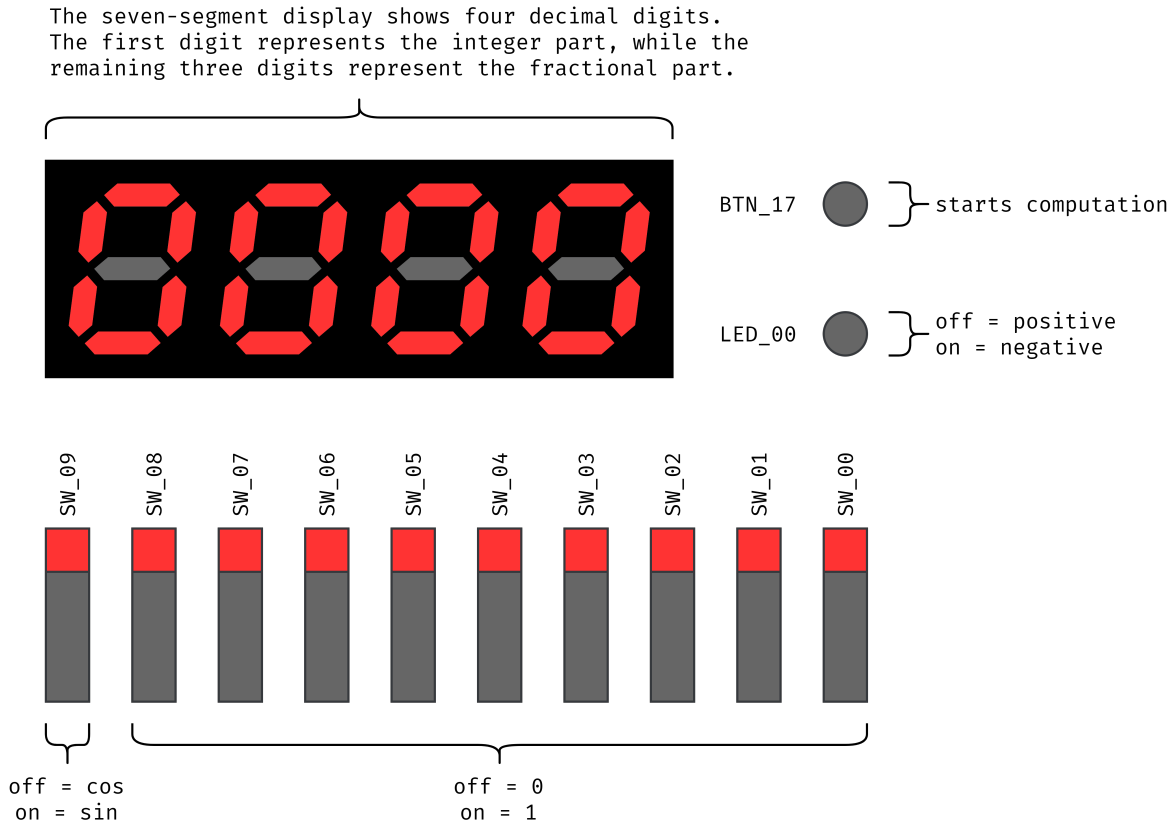

Figure 4.5: Overview of the interactive elements on the Nexys 4 DDR board.

Two experimental tests are reported below. In the first test, the sine of 35 degrees is evaluated. The value 35 is selected through the switches in binary form (100011), and switch 9 is set to select the sine function. The value shown on the display is 0.573, which is consistent with the expected result, since $\sin(35^\circ) = 0.57357644$. The sign LED remains off because the result is positive.

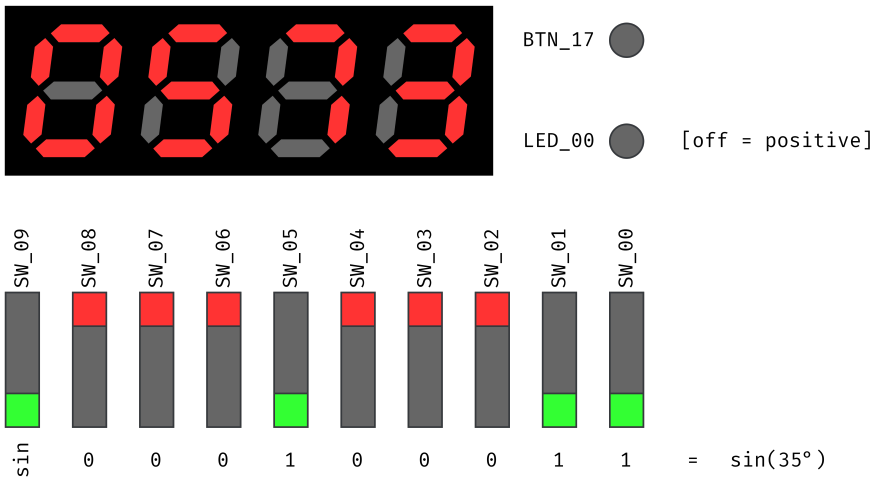

Figure 4.6: Expected reult for sin(35°) on the Nexys 4 DDR board.

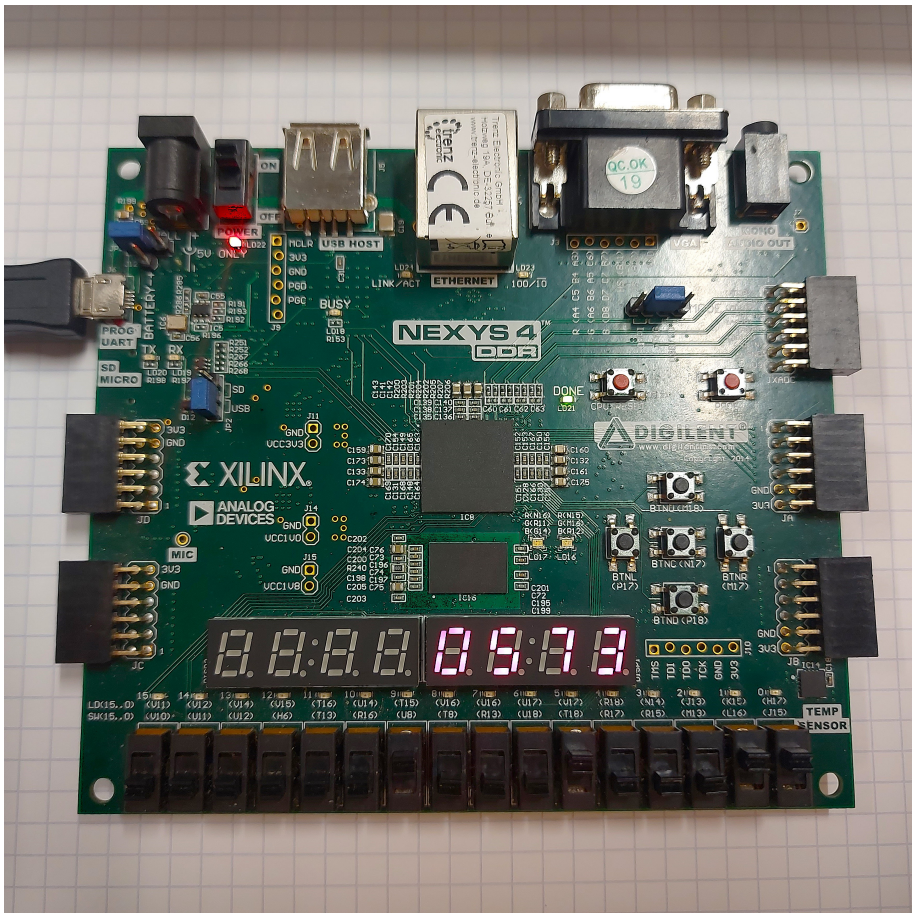

Figure 4.7: Hardware test of sin(35°) on the Nexys 4 DDR board.

In the second test, the cosine of 130 degrees is evaluated. The value 130 is selected through the switches in binary form (10000010), and switch 9 is set to select the cosine function. The value shown on the display is 0.642, which matches the expected magnitude of the result, since cos(130°) = -0.64278761. In this case, the sign LED is turned on to indicate that the computed value is negative.

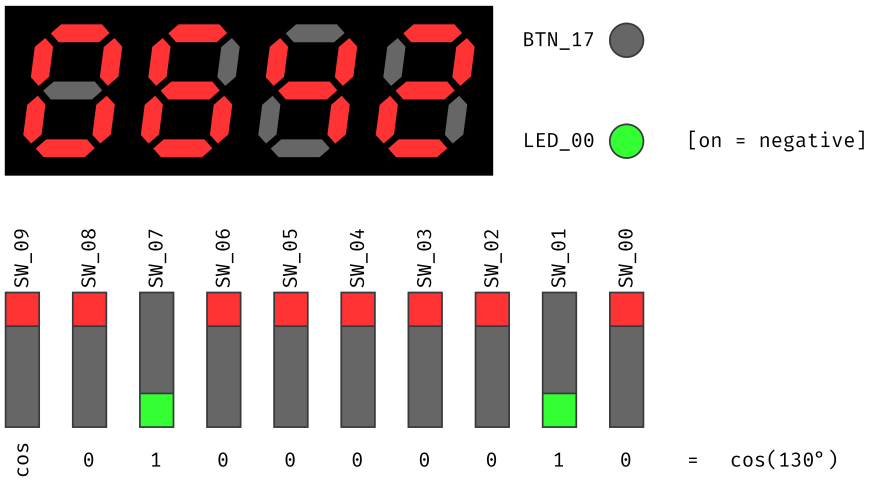

Figure 4.8: Expected result for cos(130°) on the Nexys 4 DDR board.

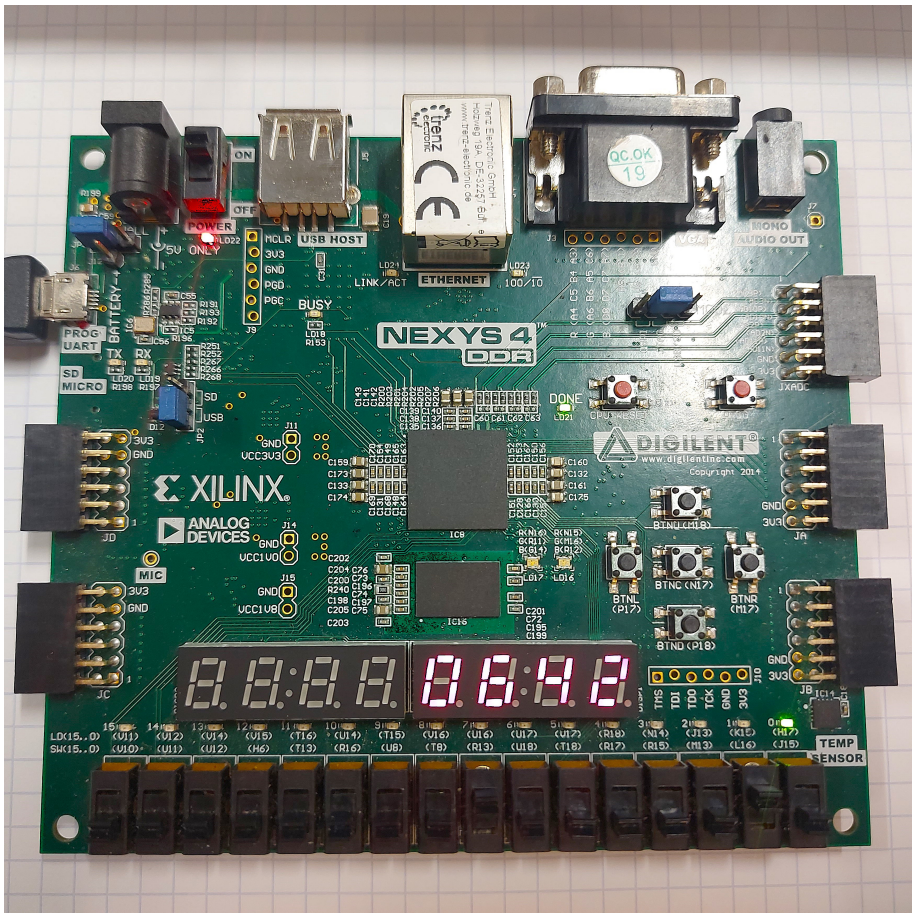

Figure 4.9: Hardware test of cos(130°) on the Nexys 4 DDR board.

Chapter 5

Conclusion

In conclusion, the project has been successfully completed and the developed system operates as expected. Each main component of the project, namely the RISC-V CPU, the CORDIC module, and the Nexys 4 DDR board interface, functions correctly when considered individually. Moreover, the interaction between the CORDIC module and the Nexys 4 DDR platform has been validated through hardware tests, confirming the correct communication between computation logic and user interface.

From an educational perspective, the project can be considered a successful outcome, as it covers the design, verification, and hardware integration of multiple digital systems. Nevertheless, several limitations and possible future extensions can be identified:

- The RISC-V CPU implements a basic single-cycle architecture without pipelining. Although the design is modular and can be extended, introducing a pipelined architecture would significantly improve performance.
- The CORDIC module currently operates as a standalone unit and is not integrated into the CPU datapath. A future extension could include the CORDIC as a dedicated functional unit or coprocessor within the CPU.
- The CORDIC algorithm exhibits limitations for specific edge cases, in particular for input angles equal to zero or exact multiples of $\pi/2$, where the sign of the computed sine and cosine may be incorrect.
- The interaction with the Nexys 4 DDR board is functional but not fully user-friendly, as the input angle must be provided in binary form using switches. Future improvements could focus on simplifying user input, for instance by introducing a more intuitive input method or a higher-level interface.

Overall, the project provides a solid foundation for further extensions and represents a meaningful learning experience in hardware design, digital architectures, and FPGA-based system integration.

List of Figures

Listings