Exercise 4

Programmable Architectures

Objectives

- Implementation of control logic for instruction and address decoding
- Assembling of Machine Code
- Functional evaluation targeting the HAW MODSYS board (XC7A100T CSG324-1)

General Description

This lab exercise is about installing and operating a simple programmable processing unit (PU), implemented as a register-load-store RISC architecture (see Fig. 1). An (almost complete) VHDL-based design description is provided to you in TEAMs, comprising a configuration package (`global.vhd`), the top-level (`DIRISC_top.vhd`) and its submodules (`control_unit.vhd`, `pmem.vhd`, `dmem.vhd`, `RF.vhd`, `ALU.vhd`, `IO.vhd`) as well as a small testbench (`DIRISC_tb.vhd`). A brief overview about the components of a register-load store architecture is provided in the next section as well as in the lecture slides (computer architectures).

Out of the instruction set architecture (ISA) in Fig. 2 and the Adress Map (AM) in Fig. 3, the processor-specific control functionality has to be set up. The correct behavior is validated by manually setting up assembler-related machine code instructions. The resulting evaluation is twofold: first, the core functionality of the PU is tested by behavioral simulation means. Second, more comprehensive IO-functionality will be tested by using the HAW MODSYS evaluation board with IOMOD extension boards (already known from exercise 2, also see Appendix 1).

After the session, each group must submit the final scoring card of exercise 4 via TEAMs: Deadline: 23rd of January 11.59 pm (earlier submissions will also be processed earlier).

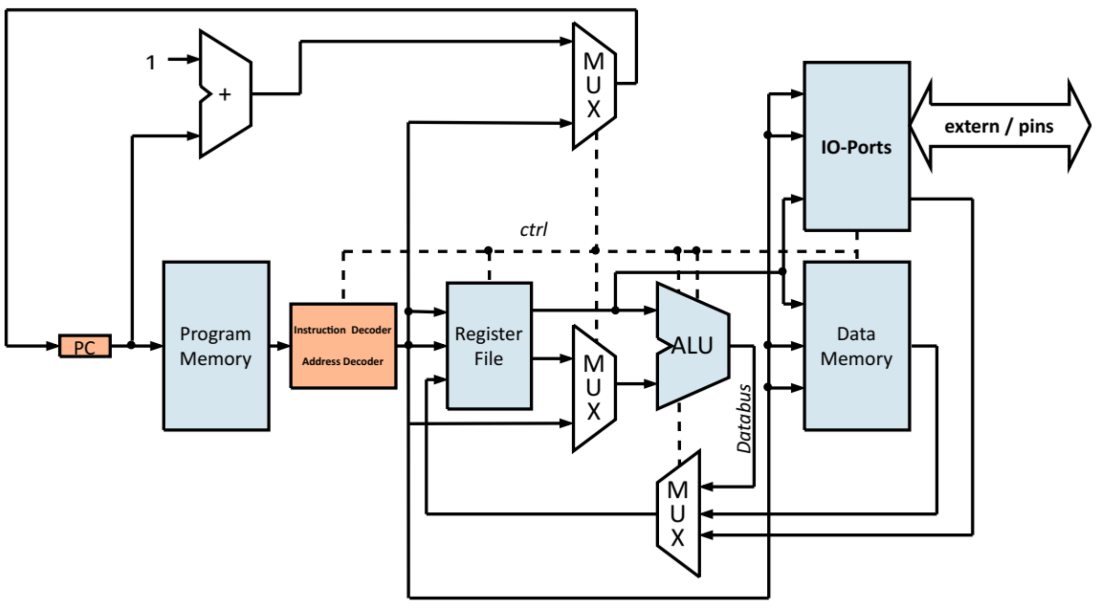


Fig. 1: *Overview about a register-load-store architecture. The orange boxes mark the control unit modules.*

RISC-Architecture with IO-Module – General Overview

A general description of the components of a register-load-store-based RISC-architecture is given in the following. More information can be found in the lecture slides.

1. Data memory (DIRISC_dmem.vhd) and Program memory (DIRISC_pmem.vhd)

Data memory (DMEM) and program memory (PMEM) are key components for storing (large amounts of) data and instructions, respectively. The PMEM stores the instructions that the PU executes. Hence, it possesses the machine code of a program. Normally, instructions are processed in a sequential manner, but also more comprehensive processing schemes are possible, e.g. branch instructions. PMEMs are typically read-only during execution and therefore implemented as optimized bram modules in FPGAs. The DMEM stores payload data, such as variables, operands, values, numbers, etc., that are used during program execution. The PU accesses this memory through explicit (load and store) instructions. A DMEM provides full read/write access during runtime, also implemented by bram modules on an FPGA.

Exemplarily, when computing an expression like `a = b + c` in a high-level language, such as C/C++, the values of `b` and c are loaded from the corresponding DMEM via their specific addresses into the PU (more precisely the register file). The addition takes places in the ALU and the result is stored back into the DMEM at the address of `a`.

Compared to CMOS-based memorizing elements (namely D-Flipflops and registers), DMEM and PMEM are slower, but more efficient regarding energy consumption and area requirements.

2. Register File (DIRISC_RF.vhd)

The register file (RF) is a high-speed (but comparatively small) storage module inside the processing unit that contains a fixed number of CMOS-based registers. These registers serve as the primary workspace of the PU because the architecture does not allow any arithmetic or logic operations to be carried out directly on memory entries. Hence, all computations are performed using values stored in registers. The RF acts as an interface between the PU and DMEM: values are loaded from data memory into registers using load instructions, and results are written back to data memory using store instructions.

3. Arithmetic, logic unit (DIRISC_ALU.vhd)

The arithmetic, logic unit (ALU) performs all arithmetic and logical operations, such as additions, subtractions, multiplications, but also AND, OR, XOR and logical shifts (see Tab. 2). As it cannot operate directly on data from the data memory, only registers of the register file can be considered as input and output operands.

4. Control unit (DIRISC_control_unit.vhd)

The control unit handles the generation of control signals that are required for the correct processing of all instructions and access to data/memories.

One key component is the **program counter register** (**PC**, externally shown in Fig. 1) that possesses the memory address of the next instruction in the PMEM to be processed. At runtime, the PC automatically increments each clock cycle, so that the following instruction is selected in the PMEM. If a branch occurs, the PC can be updated with an entirely new address, enabling the possibility to flexible modify the flow of instructions.

Another important control component is the **instruction decoder** that is capable of transforming machine code into digital control signals that direct other components, such as the ALU, the register file, and the memory modules. Typically, the instruction decoder determines the type of instruction (e.g., load, store, register-register, branch), identifies the source and destination registers, specifies any memory addresses involved, etc.

The **address decoder** is a control module used to distinctively select a correct memory location or device based on an address, e.g. provided by an instruction. Besides real memory modules, also other peripheral components, such as I/O devices, can be mapped to specific addresses in the same space as the memory. This so-called memory-mapped IO technique allows the processing unit to access peripheral components by the same instructions that are used for memory access (load/store instructions). To activate the desired component, the address decoder generates a specific chipselect signal (`cs`). Also, a write enable signal (`wren`) is needed to distinguish between read and write access.

The DI-RISC-Architecture

In this exercise a small dummy Register-Load-Store-based RISC architecture is investigated, called the **DI-RISC** architecture, possessing all the functionality mentioned before. Further device-specific parameters and properties are described in the following.

1. Architectural Overview

Data path width	16 Bit	Size of each data word is 16 bit
Instruction width	16 Bit	Size of each instruction is 16 bit
Data memory capacity	1024 (* 16 Bit)	Data memory provides 1024 memory slots *(the highest 16 are reserved for constants)*
Program memory capacity	512 (* 16 Bit)	Program memory can contain up to 512 instructions
Registers	8	Processing unit possesses 8 general purpose registers in the register file (R0-R7)

Tab. 1: *Global parameters of the DI-RISC architecture*

2. Instruction set architecture

The DI-RISC architecture distinguishes four different types of instructions (see Fig. 2).

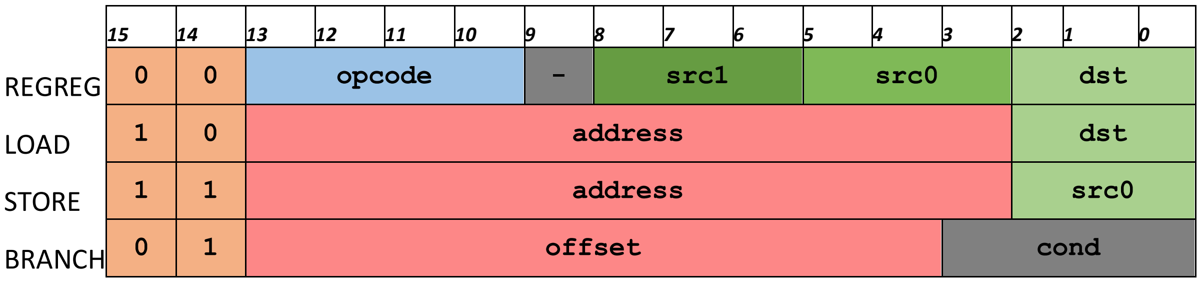


Fig. 2: *Instruction Set Architecture of the DI-RISC processing unit. The first row indicates the bit position within the instruction. gray boxes can be neglected or set to zero.*

Each of these so-called operand groups (**`opgroup`,** orange boxes) are described in the following:

REGREG: Up to two source registers (specified by the indices in **`src0`** and **`src1`**) from the RF are selected and forwarded to the ALU. There, an operation according to the **`opcode`** is carried out (see Tab. 1). The result is written back to the destination register (specified by the index in **`dst`**)

LOAD: The value at address **`address`** from the data memory is loaded into register **`dst`**.

STORE: The value in register **`src0`** is stored into the data memory at address **`address`**.

BRANCH: The PC is set to the address **`offset`** (Note: In this exercise only unconditional branches are considered. **`cond`** should be set to zero)

Instruction (command)	***Opgroup***	***Opcode***	***Operation***
Move/Copy (mov)	REGREG	0000	`<dst> := <src0>`
Addition (add)	REGREG	0001	`<dst> := <src0> + <src1>`
Subtraction (sub)	REGREG	0010	`<dst> := <src0> - <src1>`
Multiplication (mul)	REGREG	0011	`<dst> := <src0> * <src1>`
Logical AND (and)	REGREG	0110	`<dst> := <src0> ∧ <src1>`
Logical OR (or)	REGREG	0111	`<dst> := <src0> ∨ <src1>`
Logical XOR (xor)	REGREG	0100	`<dst> := <src0> ⊕ <src1>`
Logical NOT (not)	REGREG	0101	`<dst> := NOT <src0>`
Shift left (shl)	REGREG	1000	`<dst> := <src0> << 1`
Shift right (shr)	REGREG	1001	`<dst> := <src0> >> 1`
Load (ld)	LOAD	-	`<dst> := mem[addr]`
Store (st)	STORE	-	`mem[addr] := <src0>`
Branch always (b)	BRANCH	-	`PC := offset`

Tab. 2: *Overview about the full DI-RISC Instruction Set*

3. Address Map

The Address Map of the DI-RISC architecture comprises two components: the data memory and the IO module. Hence, any access to the DMEM requires the highest bit of the address (**`address`**) to be set to zero. Consequently, if **`address(10)`** is set to one, the IO-module can be accessed. To distinguish the access in the digital domain, a chipselect bit must be set. For the DI-RISC architecture, all chipselects are grouped in the `cs_s` signal. Moreover, whenever data is written to an address, this must be signalized by setting the `wren_s` signal to one (zero otherwise).

An overview about the address map of the DI-RISC architecture is given in Fig. 3. Please note, that the 16 highest numbers of the DMEM are reserved for loading constants values into the processing unit. Storing data to these addresses will have no effect.

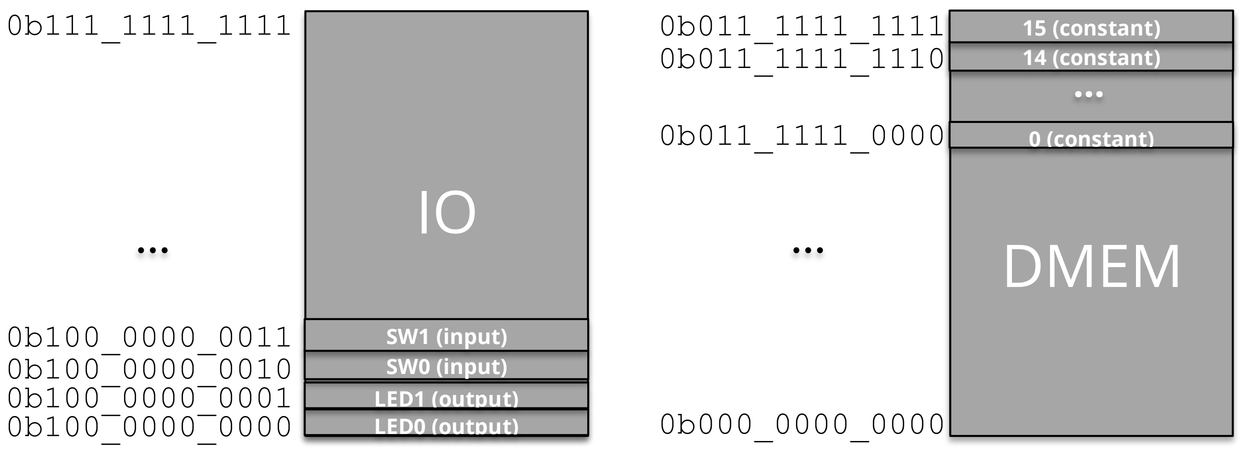


Fig. 3: *Overview about the address map of the DI-RISC architecture (with `0b` denoting a binary address). The desired resource can be accessed by load/store instructions.*

PART 1 – Core Unit

The first part of this exercise is about implementing the core functionalities of the DI-RISC architecture, mainly comprising REGREG instructions, and validating it by behavioral simulation in Vivado.

Note: Though there are multiple VHDL-files needed for this exercise, only the files **`DIRISC_global.vhd`**, **`DIRISC_control_unit.vhd`** and **`DIRISC_pmem.vhd`** need to be modified. All other files do not need to be changed.

Preparation

P1.1 – Opcodes (2pt)

- Download and familiarize yourself with the global package `DIRISC_global.vhd`.
- Fill in the opcode constants necessary for ALU-related REGREG operations (starting from line 31)

P1.2 – Instruction Decoder (6pt)

- Download and familiarize yourself with the `DIRISC_control_unit.vhd`.
- Implement the instruction decoder functionality according to the explanations in the DI-RISC architecture section on pages 3-4 (starting from line 93).
    - Note: each bit field in Fig. 2 must be assigned to its related VHDL signal (e.g. `opcode_s`).

P1.3 – Machine Code (4pt)

- Download and familiarize yourself with the `DIRISC_pmem.vhd`.
- Extend the machine code example starting from line 69 by adding at least 4 more different ALU-based (REGREG) instructions.

  Note:
    - The outcome must lead to altered values in the register file, so consecutive data manipulation is recommended.
    - You can use the values already present in `r0` (=56), `r1` (=1) and `r2` (=7) after the initial machine code instructions.

Task

T1.1 – Validation (4pt)

- Bring a copy of your prepared files and the remaining files of the DI-RISC architecture from TEAMs to the lab session, e.g. on an USB-Stick
- Create a new RTL project in Vivado (device XC7A100T CSG324-1) and add all files to the project.
- Select `DIRISC_top` as toplevel module. Select `DIRISC_tb` as simulation file.
- Perform a behavioral simulation to validate correct outcome of your architecture. Present the results in a Waveform.
    - It is recommended to use the waveform configuration DIRISC_behav.wcfg provided in TEAMs. Just after starting the simulation open the file via File-> Simulation Waveform -> Open configuration…

PART 2 – IO

In the second part Inputs are read from the IO extension module (switches), manipulated, and written back to the IO extension module (LEDs).

Task

T2.1 – Address Decoder (4pt)

- Extend the address decoder functionality of the corresponding process in the `DIRISC_control_unit.vhd` according to the explanations in the in the DI-RISC architecture section on page 4 (starting from line 101).

T2.2 – Machine Code (4pt)

- Set up a small machine code example, that loads data from the switches (`SW0`), performs a logical left shift and stores it to the LEDs (`LED0`).

T2.3 – Implementation LEDs (4pt)

- Apply the changes to the Vivado project.
- Perform the Implementation Design Flow. (no simulation, just generate Bitstream)
- Test the adder by applying your test vectors through the IOM-Boards (Connector 3, see Appendix)
    - Note: clock mode of the MODSYS board must be set to single

T2.4 – Timing Analysis (2pt)

- Open the results of the physical synthesis, (Open Implemented Design -> Report Timing Analysis)
- Inspect the timing and reflect the results in the final discussion.

Appendix 1 – MOSYS evaluation board and Extension boards

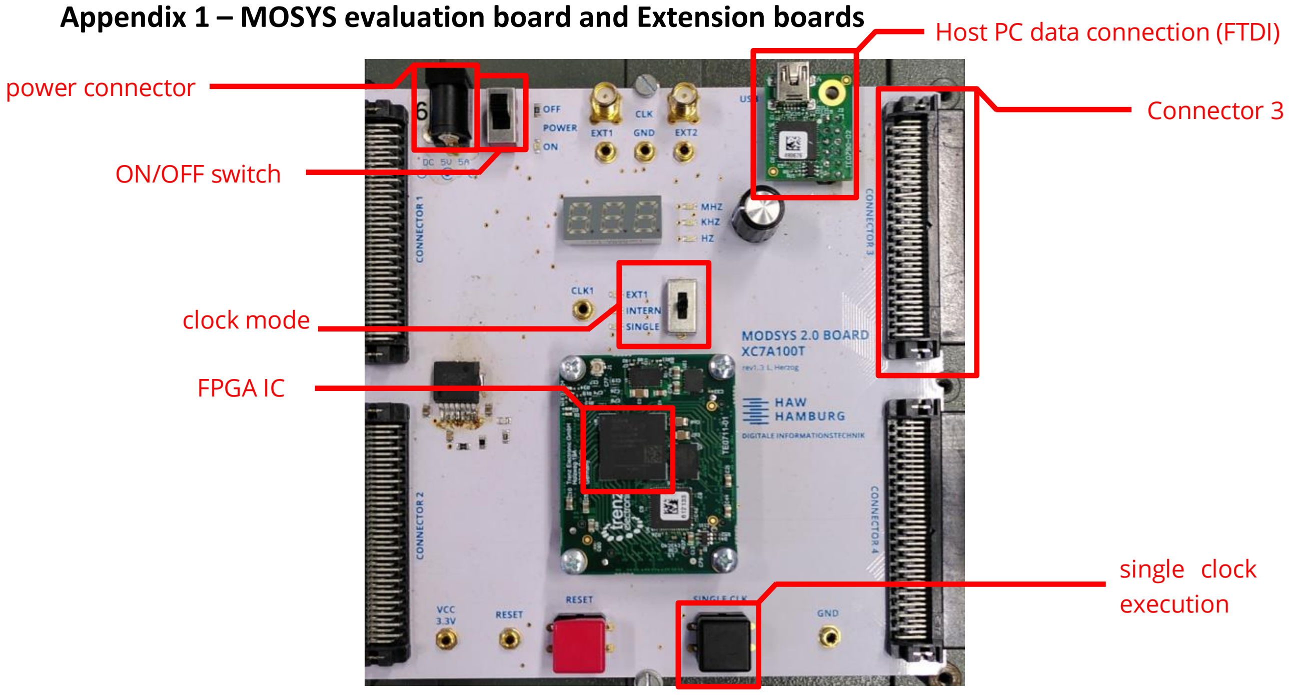


Fig. A1: MODSYS 2.0 Evaluation Board

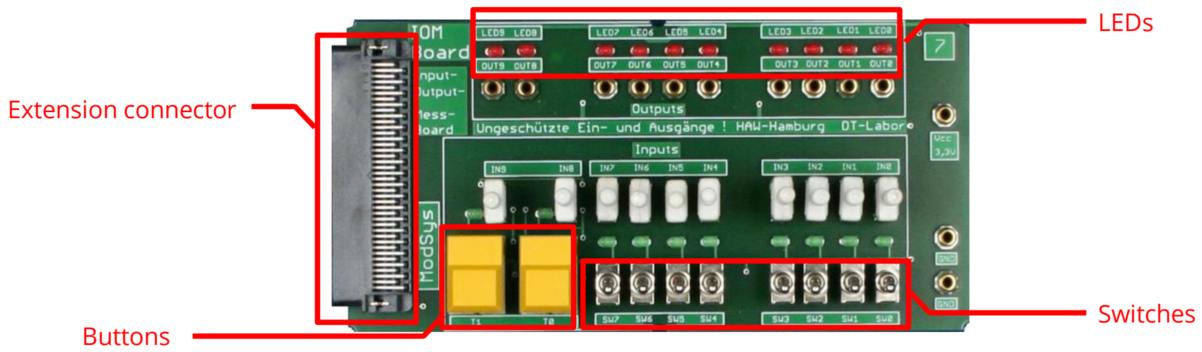


Fig. A2: IO-Module extension card

For the exercise, the IO extension cards is connected to the MODSYS 2.0 via Connector 3:

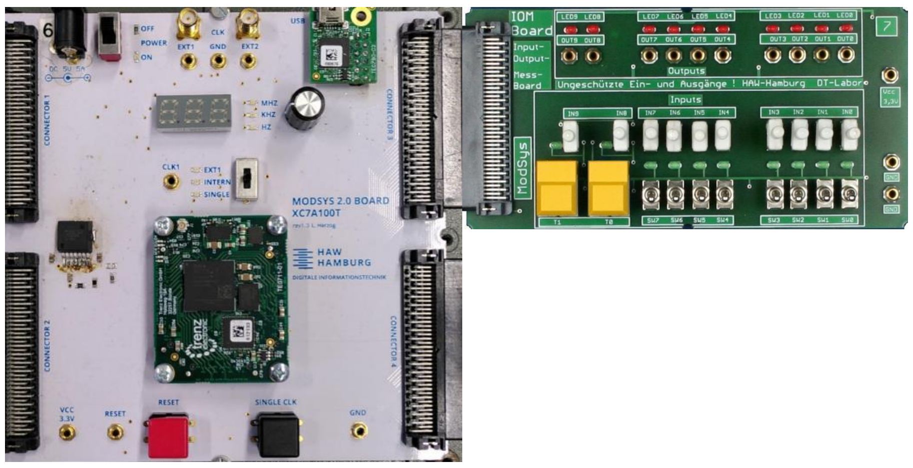
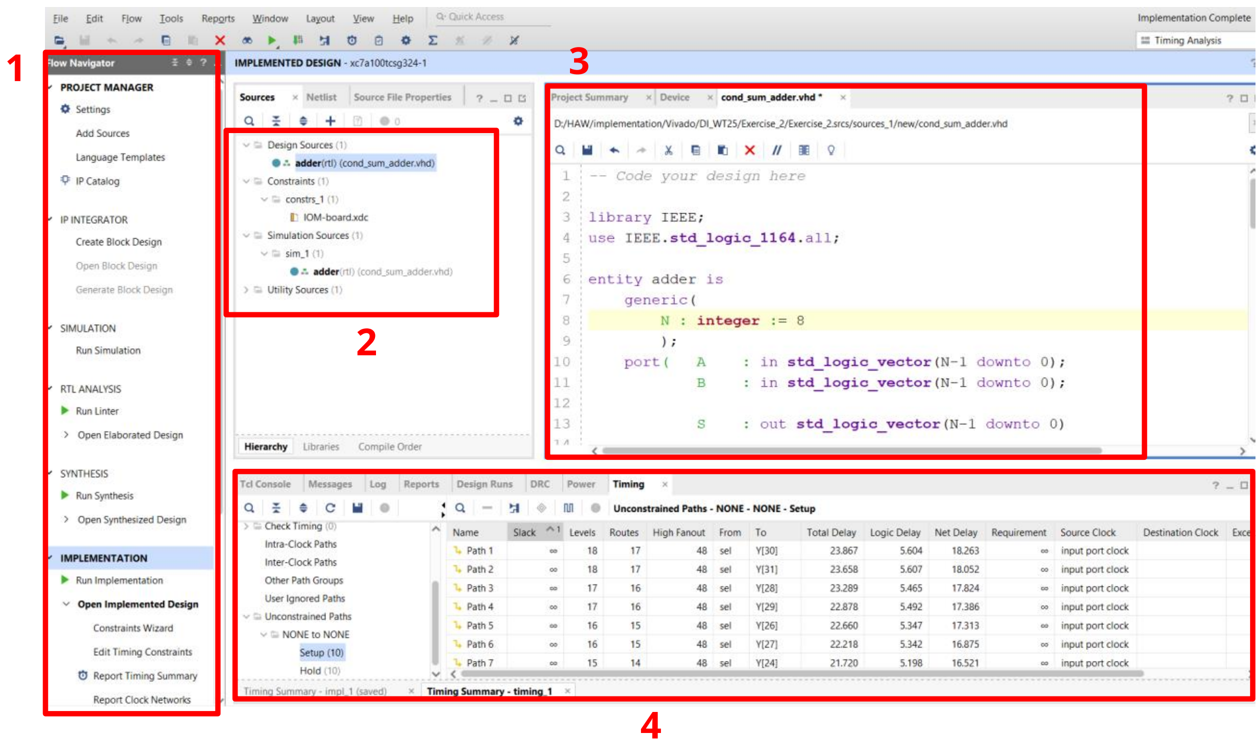


Fig. A3: Exercise Setup

Appendix 2 – Overview Vivado



```vhdl
-- Code your design here

library IEEE;
use IEEE.std_logic_1164.all;

entity adder is
    generic(
        N : integer := 8
        );
    port(   A    : in std_logic_vector(N-1 downto 0);
            B    : in std_logic_vector(N-1 downto 0);

            S    : out std_logic_vector(N-1 downto 0)
```

Unconstrained Paths - NONE - NONE - Setup

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock	Destination Clock	Exce
Path 1	∞	18	17	48	sel	Y[30]	23.867	5.604	18.263	∞	input port clock		
Path 2	∞	18	17	48	sel	Y[31]	23.658	5.607	18.052	∞	input port clock		
Path 3	∞	17	16	48	sel	Y[28]	23.289	5.465	17.824	∞	input port clock		
Path 4	∞	17	16	48	sel	Y[29]	22.878	5.492	17.386	∞	input port clock		
Path 5	∞	16	15	48	sel	Y[26]	22.660	5.347	17.313	∞	input port clock		
Path 6	∞	16	15	48	sel	Y[27]	22.218	5.342	16.875	∞	input port clock		
Path 7	∞	15	14	48	sel	Y[24]	21.720	5.198	16.521	∞	input port clock		

1. Flow navigator:
    a. For starting the synthesis steps, bitstream generation, etc.
    b. For opening designs and analyse timing
2. Design sources
    a. Design, cornstraints, simulation/testbench
3. Editor
    a. VHDL files, summary, etc
4. Console
    a. Design flow outputs (Logs)
    b. Design analysis outputs: Timing, Power, etc.