Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA
Fondazione ICSC

Advanced course to FPGA programming

Piero Vicini, Ottorino Frezza, Francesca Lo Cicero, Francesco Simula,
INFN Rome1

INFN Istituto Nazionale di Fisica Nucleare

Course Overview

Introduction to FPGA

- Architecture
- Advantages and limitations
- Design Flow
- Simulation tool

Tools & Programming languages

- Advanced VHDL
- Vivado tool overview
  - Design Flow
    - Project creation
    - Ip core integration
    - Simulation
    - Synthesis
    - Implementation
    - Tcl scripting
  - Timing analysis
  - Custom IP Design & Integration
  - FPGA Test & Debug

FPGA interconnection

- Quick intro
- AURORA Xilinx
  - Test and Debug
  - Fine tuning
  - Timing and Resources analysis
  - Optimization

Vivado Design Suite

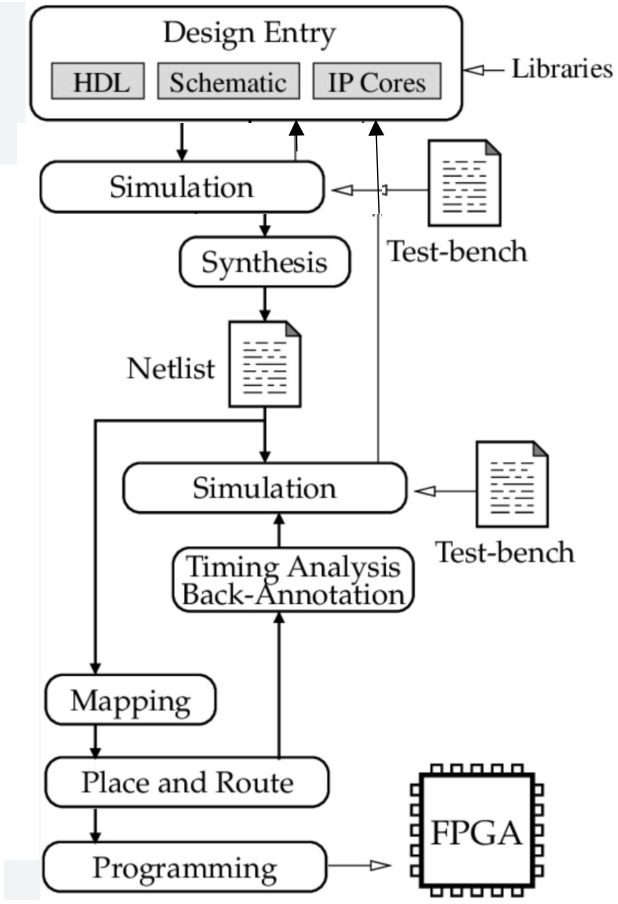
Vivado provides an end-to-end environment for VHDL design flow: everything from coding, simulation, synthesis, verification, and hardware programming is integrated in a single tool.

The ***-mode*** option in Vivado specifies the startup mode

- **gui** (default): Launches the graphical user interface.
  vivado -mode gui
- **tcl:** Starts the interactive Tcl console.
  vivado -mode tcl
- **Batch:**
  - Run scripts without interaction
  - Useful for automation with Tcl scripts.
  - Usually combined with -source <script.tcl> command.

  vivado -mode batch -source build.tcl

VHDL design flow



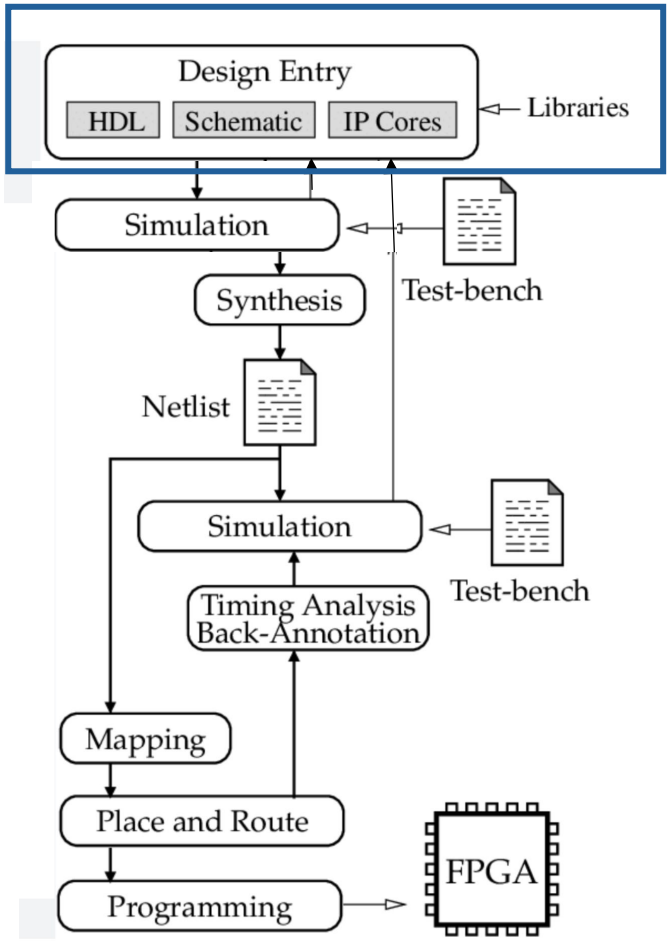
Vivado Design Entry

Design entry is the process of providing the design description to the tool

Design Entry Methods in Vivado:

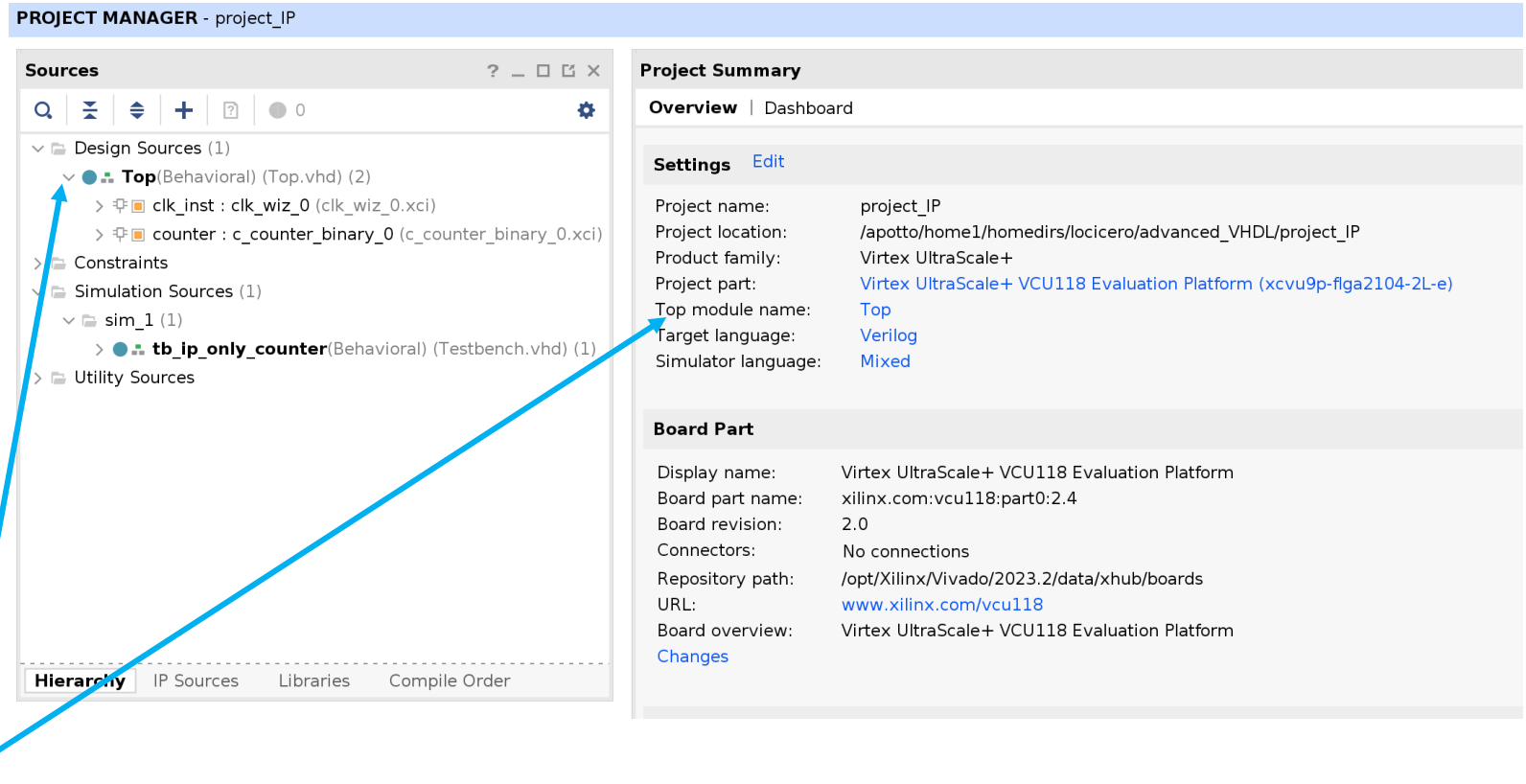
- HDL (VHDL / Verilog / SystemVerilog)
  - Design described as source code.
  - Most flexible and widely used method.
- IP Integrator (Block Design)
  - Uses pre-built IP cores connected in a block diagram.
  - Suitable for system-level designs (e.g., processors, memory controllers, AXI interfaces).
- Schematic Entry
  - Design described by drawing circuits with logic symbols.
  - Less common today, mainly for small designs.
- High-Level Synthesis (HLS) → Vitis HLS
  - Algorithms described in C/C++/SystemC.
  - Translated into HDL by Vivado HLS.

VHDL design flow



Vivado HDL-Based Design Entry

Describes digital designs using HDL source files



Vivado automatically identifies the top-level module of the design hierarchy and determines the elaboration, synthesis, and simulation order of all source files.

- The automatic configuration can be overridden by explicitly setting the top module and customizing the compile order in the project settings.

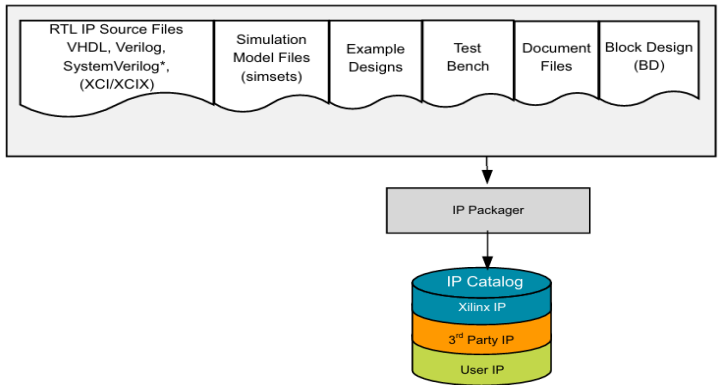
Vivado IP-Based Design Entry

IP (Intellectual Property) core: reusable logic/design block

Make FPGA development faster and modular

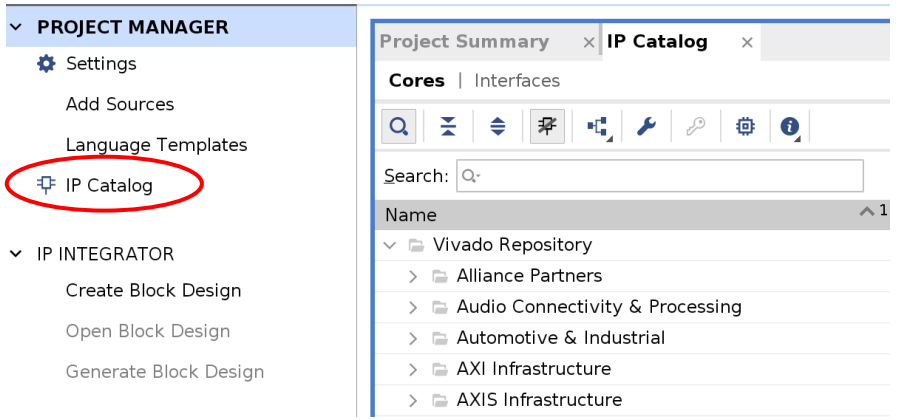
Vivado supported IP:

- **Vendor-supplied** (Xilinx)
- **User-defined IPs**
  - A custom hardware module (written in VHDL, Verilog, or HLS) that is packaged into an IP format



- **Third-party IPs**

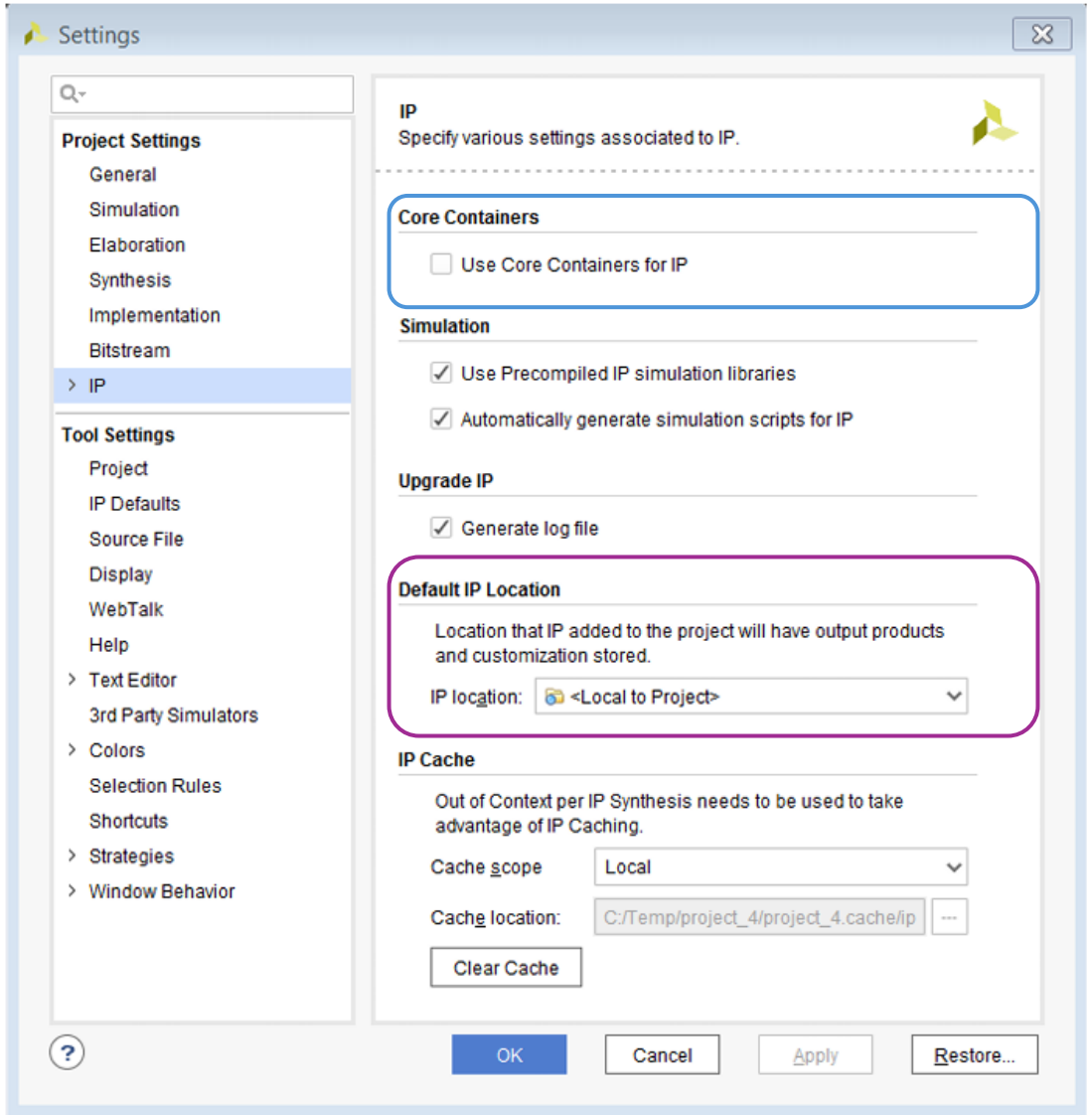
Vivado IP Catalog



- GUI interface listing all available IP cores
- Organized by category: memory, communication...
- Allows configuration and instantiation of IPs
- By default, the IP catalog only displays IP cores that are supported by the target part (or board) for the current project.

Vivado: General IP Setting

IP settings are used to define various project-specific options for IP

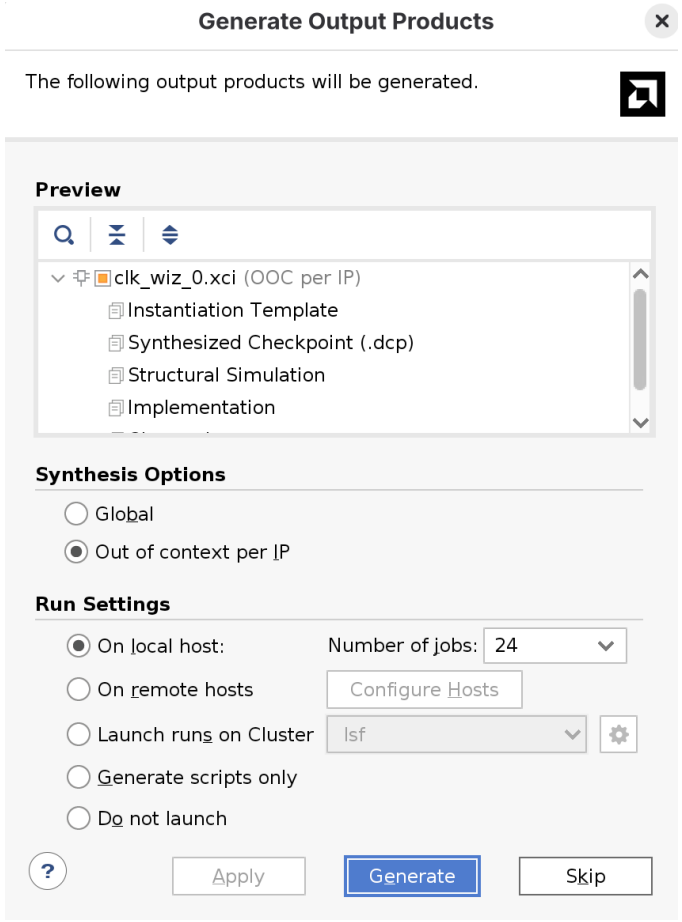


A **core container** is used in Vivado to store an IP core and all generated output files in a single compressed binary file with the .xcix extension.

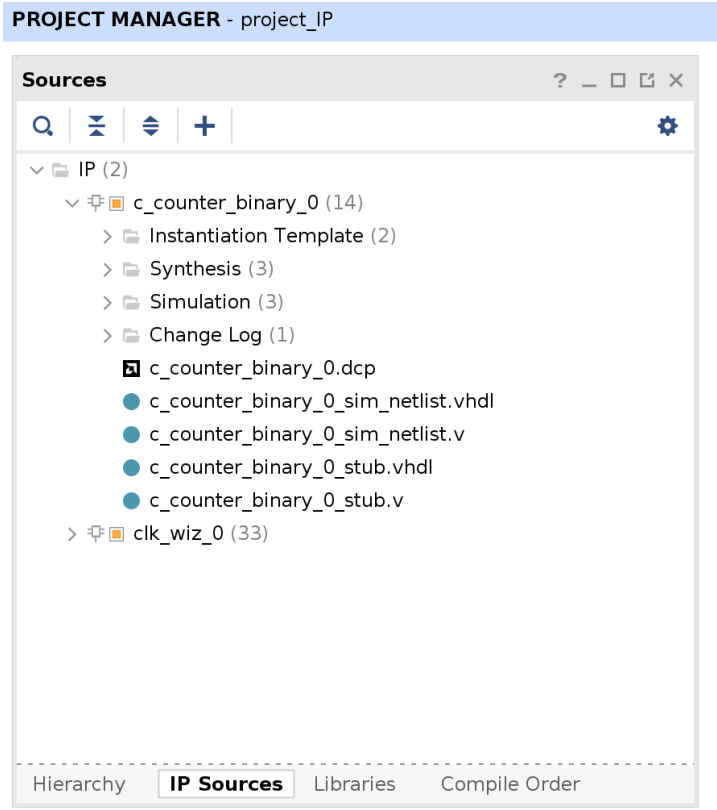
Setting the default IP location will persist across multiple Vivado sessions.

Vivado: IP Generate Output products

The Generate Output Products step creates all the files required for synthesis, simulation, and implementation of the IP

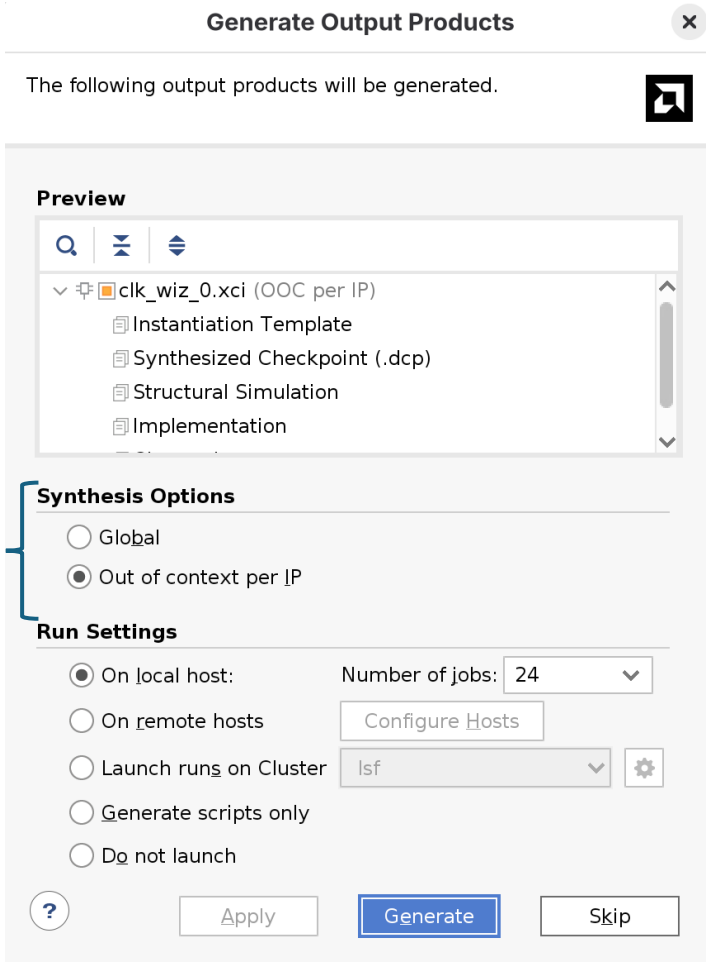


Output products delivered by the IP



Vivado: IP Generate Output products

For either Synthesis option, Vivado generates HDL and XDC files for the IP and uses those files during synthesis and during implementation.



- **Global Synthesis:** all design sources are synthesized together

- **Out-of-Context per IP (OOC):** the Vivado tools synthesize the IP as a standalone module and produces a design checkpoint (DCP).
  Advantage:
  - It improves synthesis time by avoiding IP re-synthesis during project runs.

  Disadvantege:
  - IP may not be fully optimized when integrated into the full design

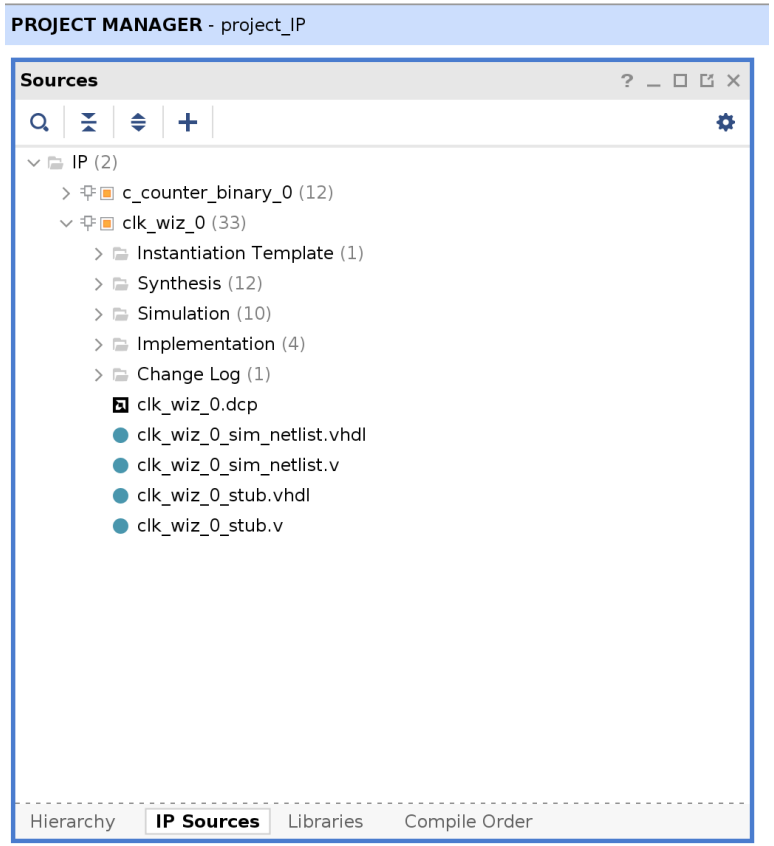
Vivado: IP Source tab

The **IP Sources** tab provides a hierarchical view of all IP cores added to the project.

- Allows access to IP configuration files, and generated output products (such as HDL wrappers, simulation models, and synthesis netlists)
- Tracks the generation status of each IP.

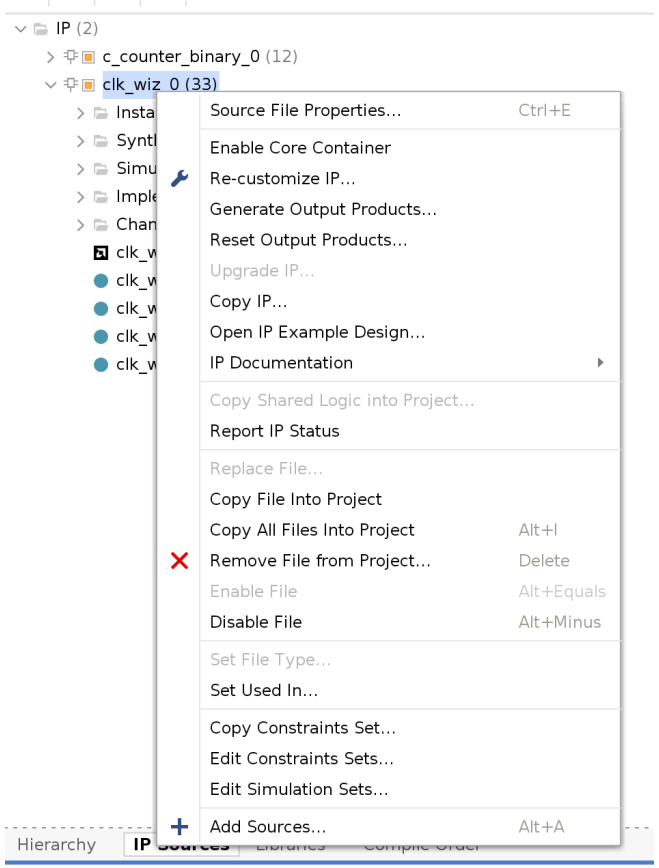
	IP to be synthesized OOC.
	IP to be synthesized with the project (global synthesis).
	Unmanaged IP. The purpose is for the user to make modifications to unencrypted HDL sources or constraints.
	Locked IP that cannot be customized or regenerated due to Vivado version mismatch or incompatible target device settings.

Run report_ip_status for more details and recommendations on how to fix this issue.

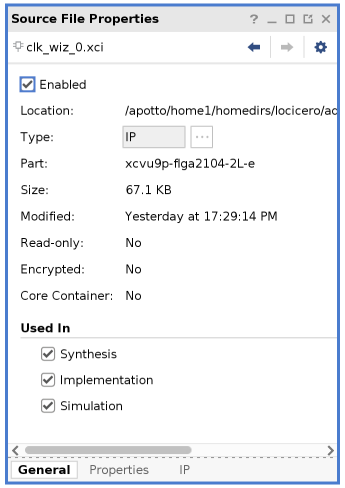


Vivado: IP Source tab

- By right-clicking on an IP core in the **IP Sources tab**, several operations can be performed



- The **Source File Properties** window shows file-specific settings such as file type, library, compile order, and usage (Synthesis, Implementation, Simulation) in the design flow.

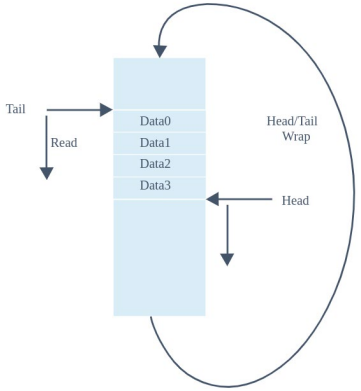


- The **Open IP Example Design** option allows opening a standalone Vivado project that demonstrates how to use the selected IP core.
  It includes pre-configured design sources, a testbench, constraints, and sometimes simulation scripts to help understand and validate the IP's functionality.

FIFO

FIFO (First In First Out) is a method to organize a data buffer as a queue.

- The FIFO is implemented using circular buffer.
- Write/read pointers act as selector for demux/mux on data.
- A Fifo control logic manage a write pointer (Head) and a read pointer (Tail), necessary to avoid over-flow (or write Full) or under-flow (read empty).



- **Port structure**
  - Standard / Native interface:
    Simple write and read ports, single data bus.
  - AXI or bus-based FIFOs:
    Compatible with AXI4/AXI-Stream interfaces.
- **Clocking scheme**
  - Single-clock FIFO (synchronous): Write and read share the same clock.
  - Dual-clock FIFO (asynchronous): Write and read operate on separate clocks.
- **Behavior / mode**
  - Standard FIFO:
    First word appears on output port only after asserting read_enable.
  - First-Word Fall-Through (FWFT or Show-Ahead):
    First written word appears immediately on the output without asserting read_enable.

Xilinx provides the FIFO Generator IP for creating FIFO memories.

Hands-on

Exercise 0

FIFO_project

- Create a new Vivado project (named FIFO_project)

Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	Ultra RAMs	DSPs	BUFGs	Gb Transceivers	GTI
xcu55c-fsvh2892-2L-e	2892	624	1303680	2607360	2016	960	9024	61440	40	0

- Add a FIFO IP core using the FIFO Generator (found in Memory&Storage category) leaving all default setting unchanged
- Create a top-level VHDL module named fifo_wrapper with the following interface:

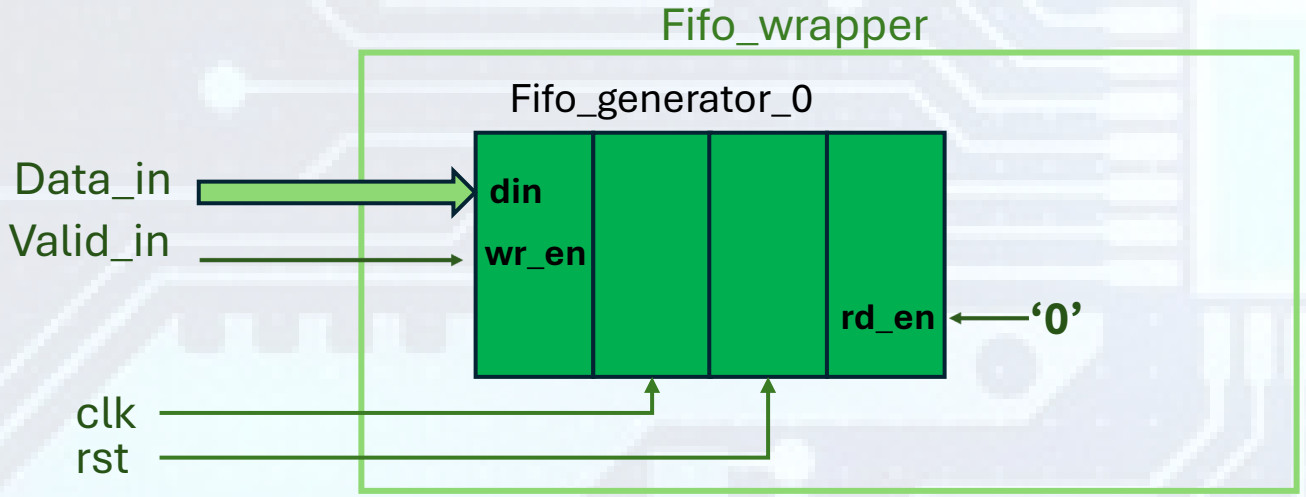
```
GENERIC ( data_width : integer:=32);
PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    data_in : IN STD_LOGIC_VECTOR(data_width-1 DOWNTO 0);
    valid_in : IN STD_LOGIC);
```

- Instantiate the FIFO in fifo_wrapper

Tip: After generating the FIFO, you can use the Instantiation Template provided by Vivado (in **IP Sources tab**) to correctly instantiate the component in your VHDL code.

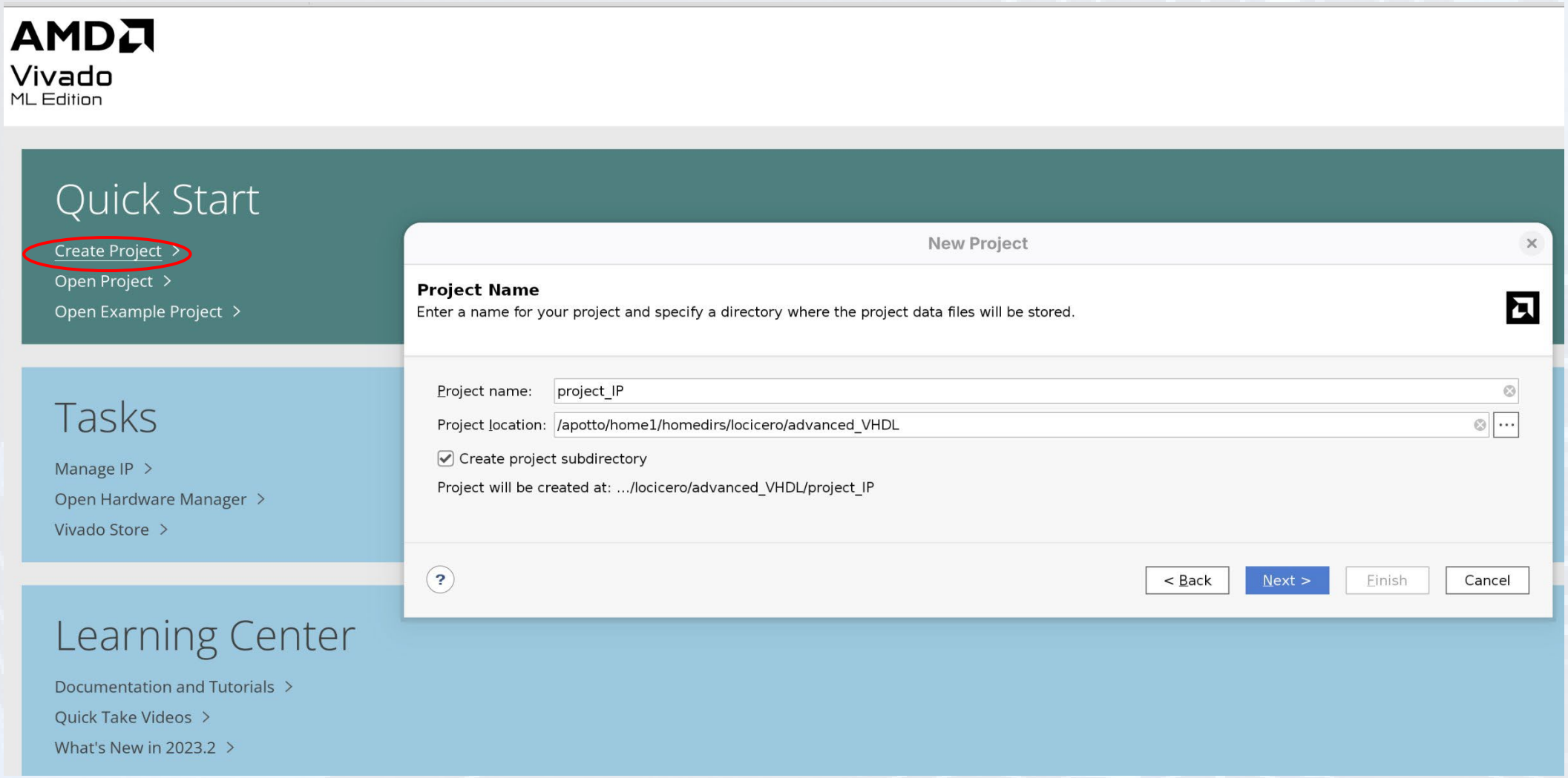
FIFO_project

- Connect the FIFO:
  - **din** ⟵ data_in
  - **wr_en** ⟵ valid_in
  - Connect **clk** and **rst** to both the FIFO and the top module.
  - Connect **rd_en** to '0' (the FIFO is not read)
  - Leave FIFO output unconnected (open)



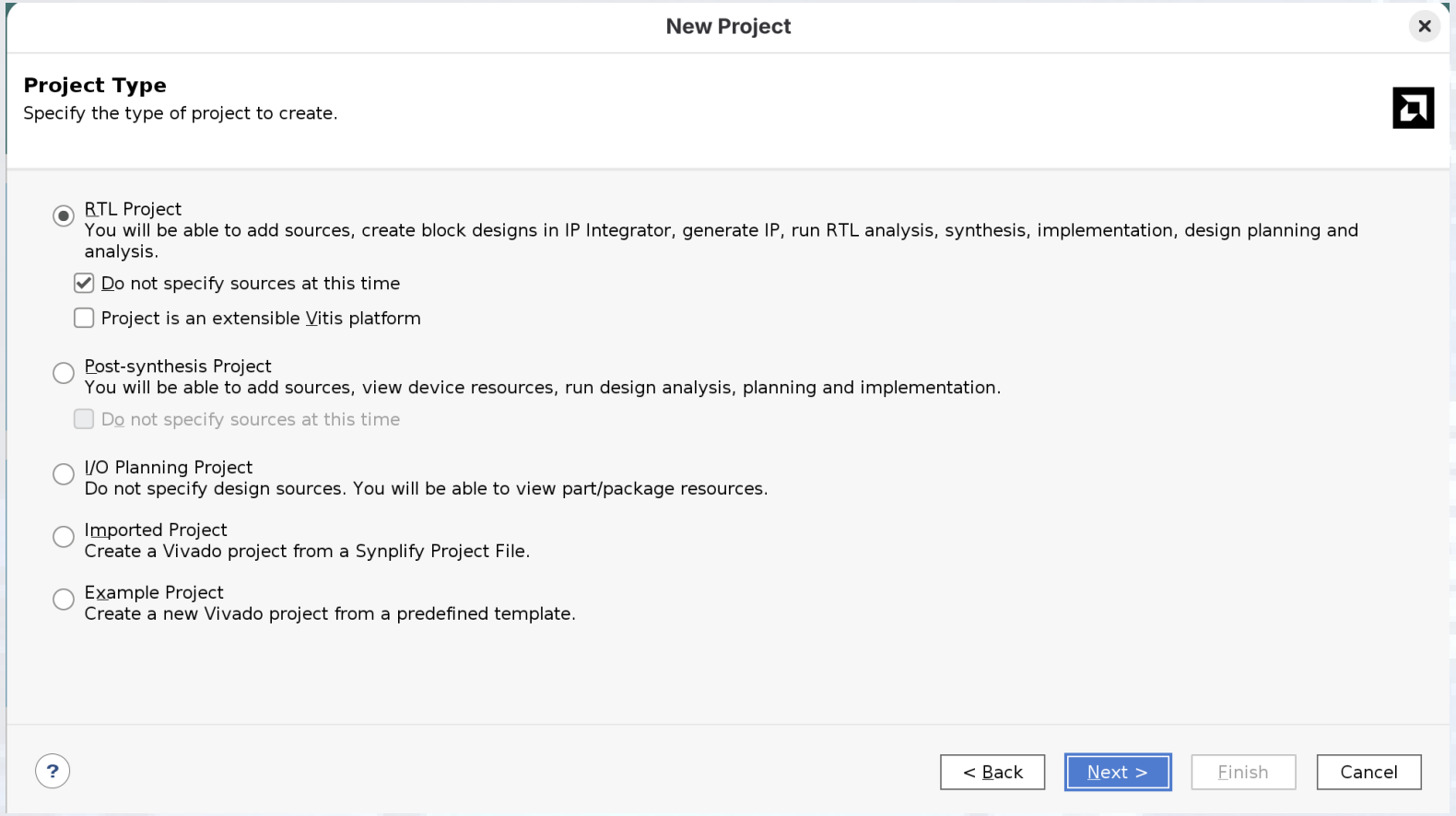
Quick Guide to Vivado GUI

Create Project



Quick Guide to Vivado GUI

Project Setup

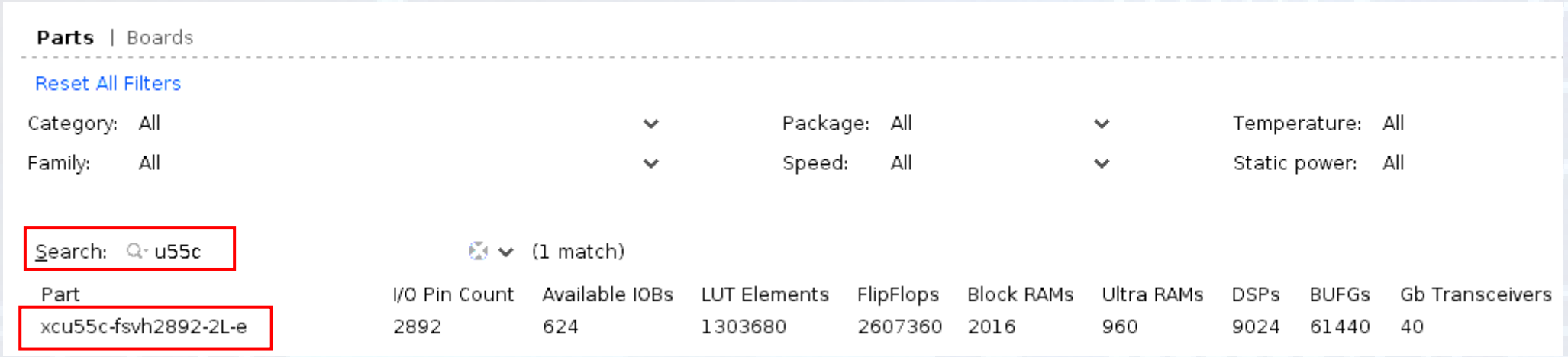


- Project Type determines the types of source files that are associated with the project.
  - **RTL project**
    - block design
    - IntellectualProperty
    - RTL sources

      VHDL, Verilog, SystemVerilog

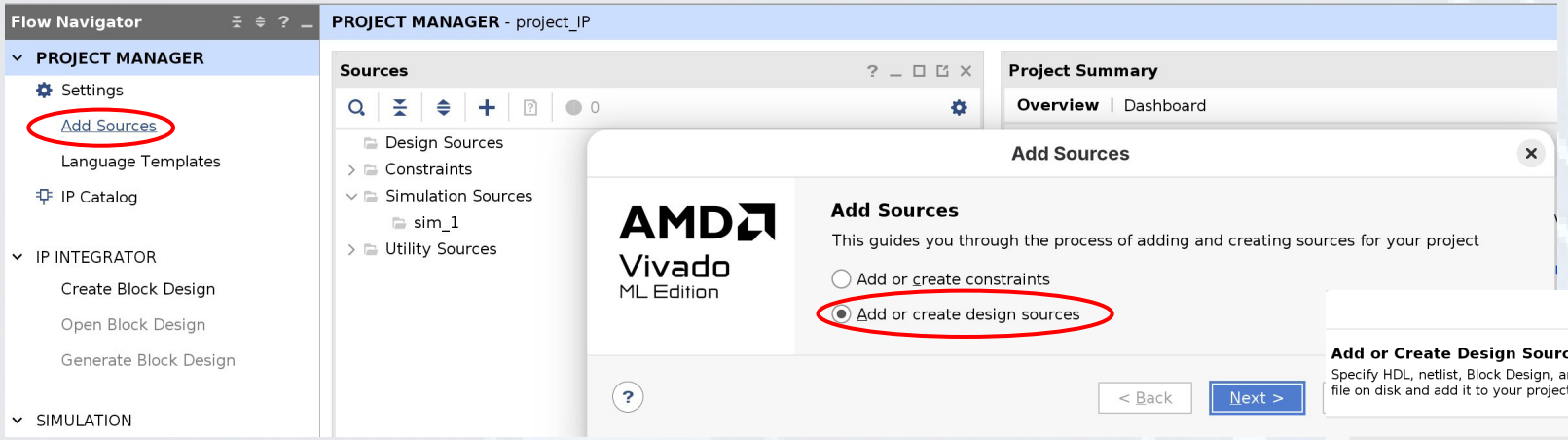
Quick Guide to Vivado GUI
Project Setup

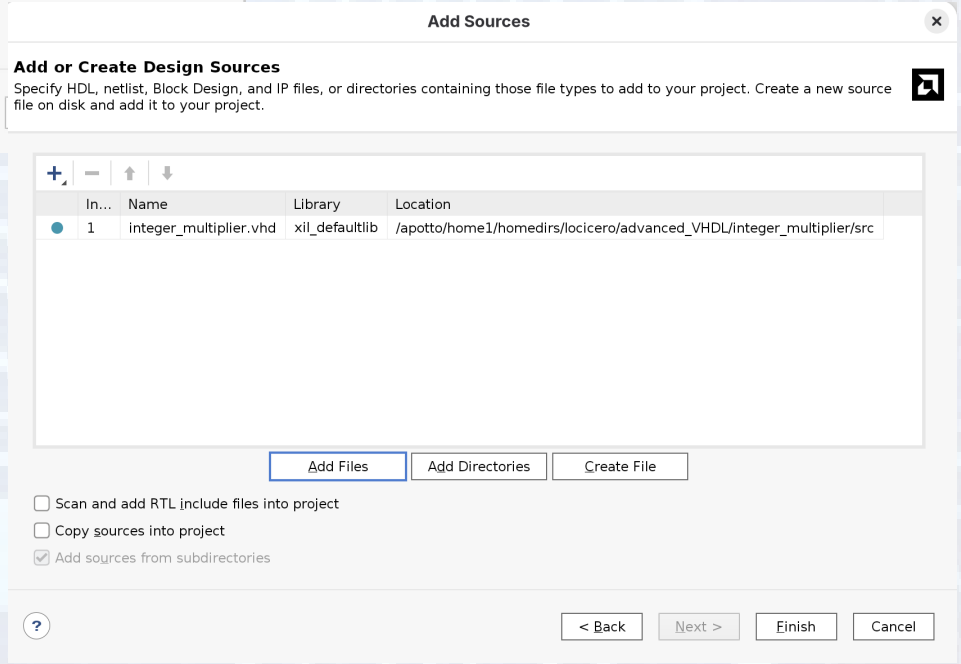
For the first projects (RTL code simulation only), selecting an FPGA/board is not technically required. However, Vivado requires a target to be specified, so we select the board we plan to use later.



Quick Guide to Vivado GUI

Add Design Sources



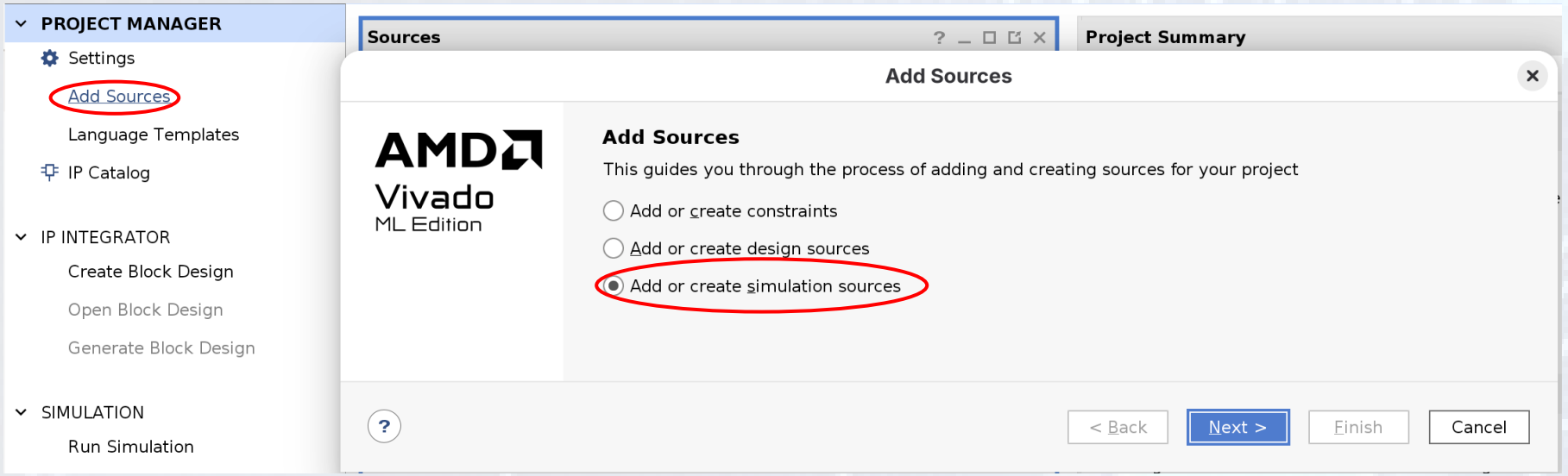


It is possible to add files individually (**Add Files**) or all files in a directory **(Add Directories**).

- To enable/disable source files select the Enable/Disable File right-click menu command.

Quick Guide to Vivado GUI

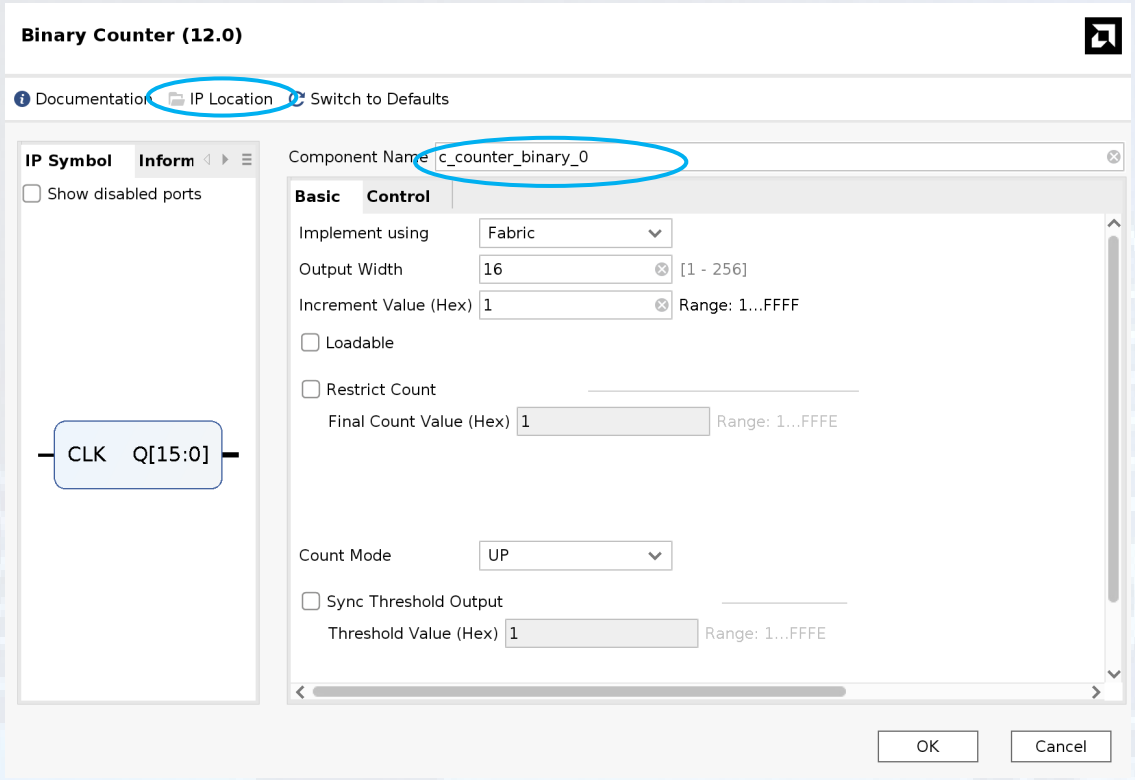
Add Simulation Sources



Quick Guide to Vivado GUI
Add IP from IP Catalog

To add an IP select it by double-clicking it in the IP catalog.

The Customize IP window shows available parameters, which vary based on the IP core.



Specify the name and the location on disk to store the IP.
Default: <project_name>.src/sources_1/ip/

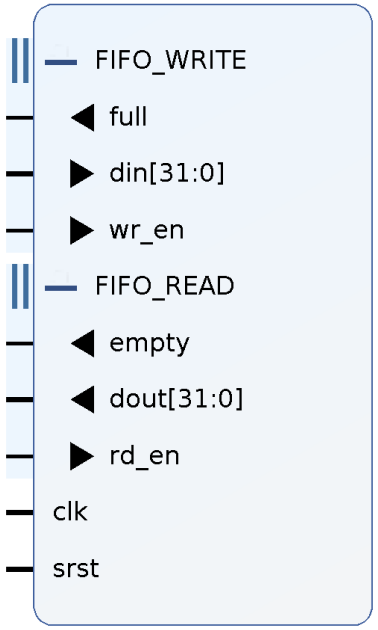
Previously created IP cores — typically in **XCI** format — can also be added to the project using **Add Sources**.

Vivado Native FIFO Configuration

FIFO Generator core is a fully verified first-in first-out (FIFO) memory queue
https://docs.amd.com/v/u/en-US/pg057-fifo-generator

Supported Interfaces

- **Native**
- AXI Memory Mapped
- AXI Stream



Native Interface Signals for Common Clock FIFOs

- **rst/srst**: asynchronous/synchronous reset that initializes all internal pointers and output registers.
- **clk**: all signals on the write and read domains are synchronous to this clock.
- **din**[n:0]: The input data bus used when writing the FIFO.
- **wr_en** : If the FIFO is not full, asserting this signal causes data (on din) to be written to the FIFO.
- **full** : When asserted, this signal indicates that the FIFO is full. Write requests are ignored when the FIFO is full, initiating a write when the FIFO is full is not destructive to the contents of the FIFO.
- **dout**[m:0]: The output data bus driven when reading the FIFO.
- **rd_en**: If the FIFO is not empty, asserting this signal causes data to be read from the FIFO (output on dout).
- **empty**: When asserted, this signal indicates that the FIFO is empty. Read requests are ignored when the FIFO is empty, initiating a read while empty is not destructive to the FIFO.

Vivado Native FIFO Configuration

In the native port tab, it is possible to configure FIFO settings

Read Mode:
- **Standard FIFO**
- **FWFT FIFO**
  - this implementation increases the depth of the FIFO by 2 read words.

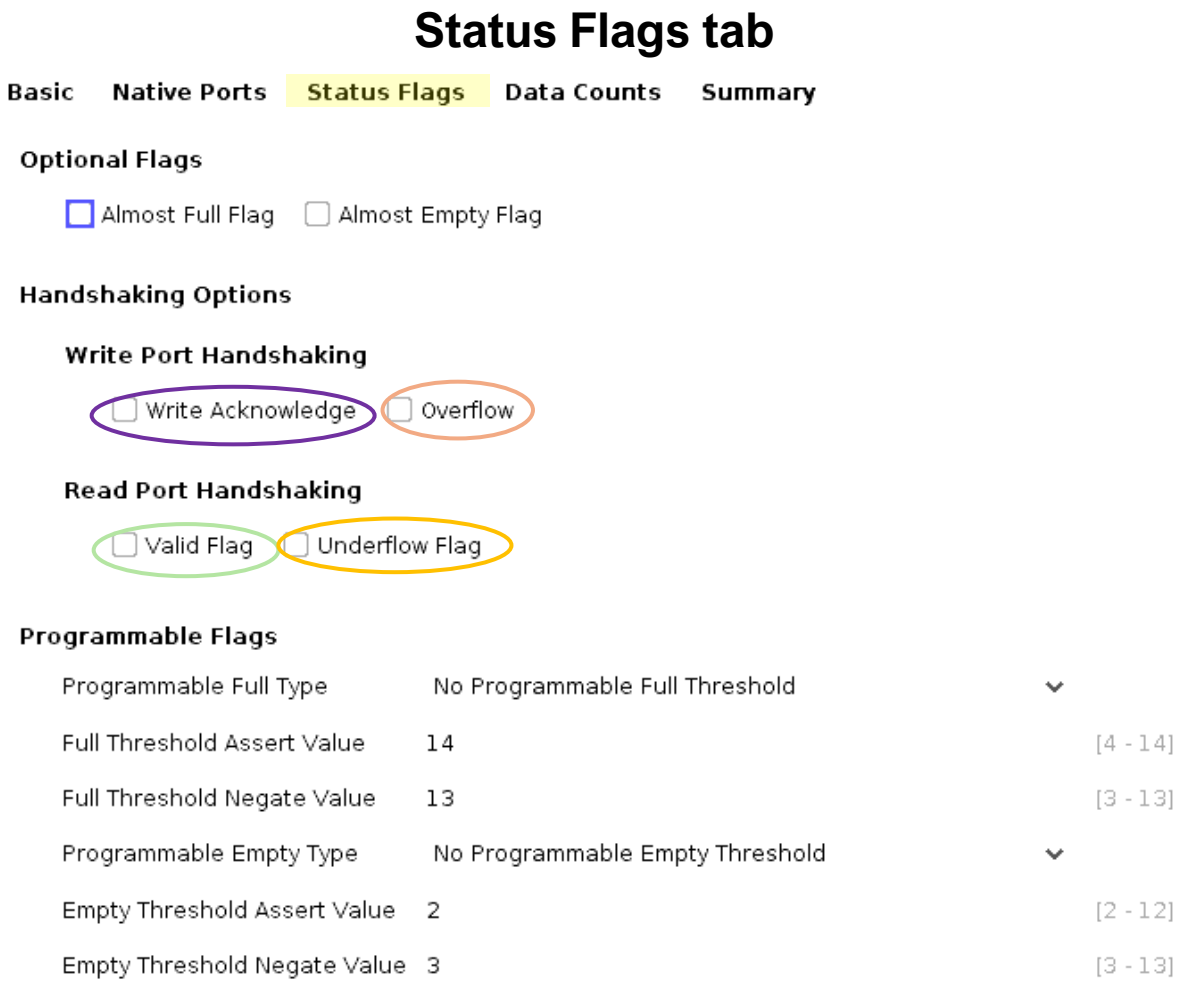
Data port parameters:
- Width: number of bits per entry
- Depth: total number of entries in the memory

ECC (Error Correction Code) is an optional feature that adds error detection and correction to the FIFO's internal memory.
It is useful in critical system where data may be corrupted by radiation, electrical noise, hardware faults, ensuring that the data read from the FIFO is the same as what was written.

Native Ports tab

Vivado Native FIFO Configuration

Optional ports can be enabled in the Status Flags tab



Write Acknowledge: Generates write acknowledge flag which reports the success of a write operation.

Overflow (Write Error): Generates overflow flag which indicates when the previous write operation was not successful.

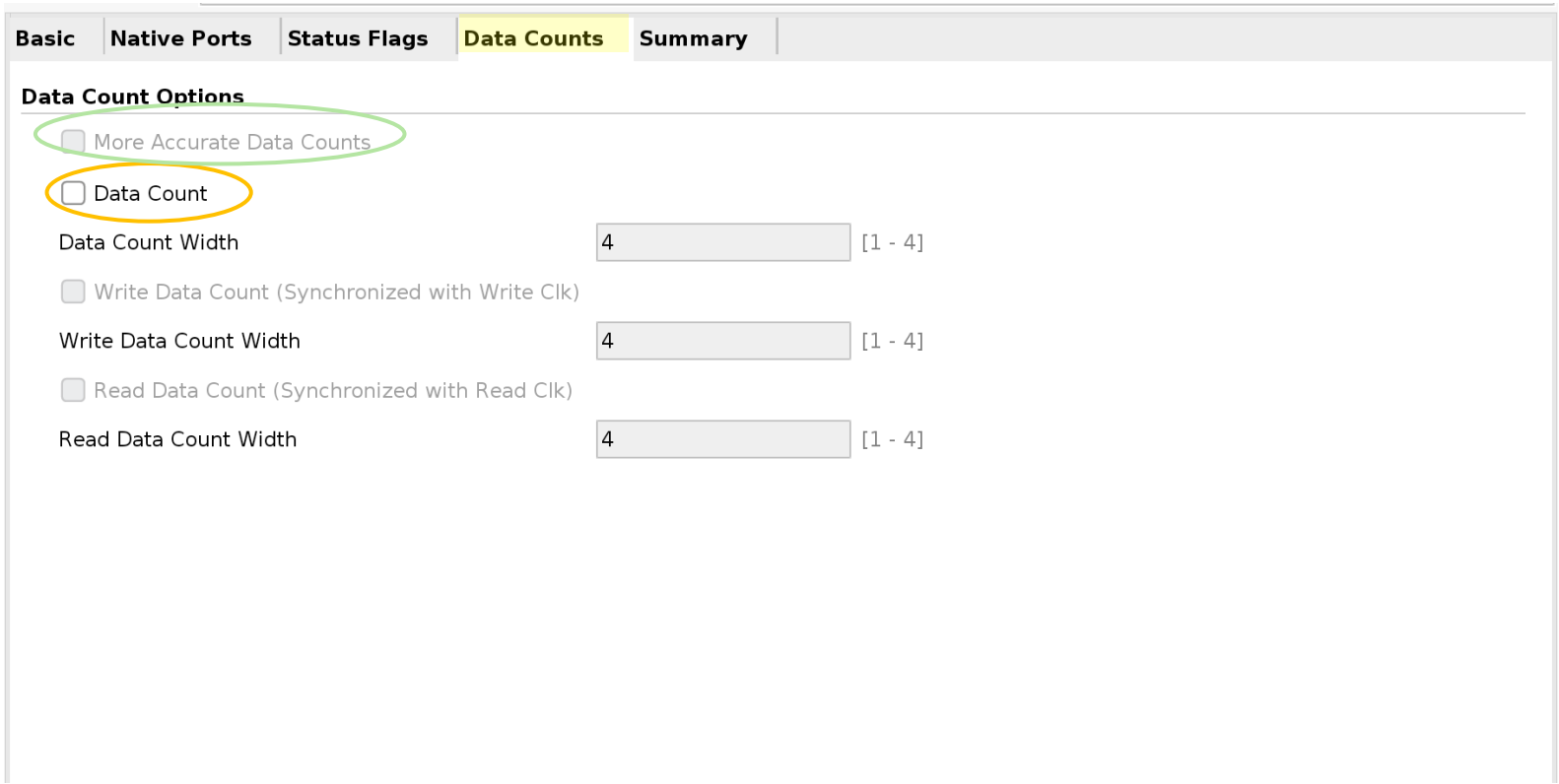
Valid Flag: Generates valid flag that indicates when the data on the output bus is valid.

Underflow (Read Error): Generates underflow flag to indicate that the previous read request was not successful.

Vivado Native FIFO Configuration

Data Counts tab: enables optional signals reporting the current FIFO fill level

Data Counts tab



Data Count: output signal that indicates how many data entries are currently stored in the FIFO buffer.

More Accurate Data Counts: This option uses additional external logic to generate more accurate data count signals, which indicate the number of data words currently stored in the FIFO.

Only available for independent clocks FIFO with block RAM or distributed RAM, and when using first-word fall-through.

Hands-on

Exercise 1

FIFO project

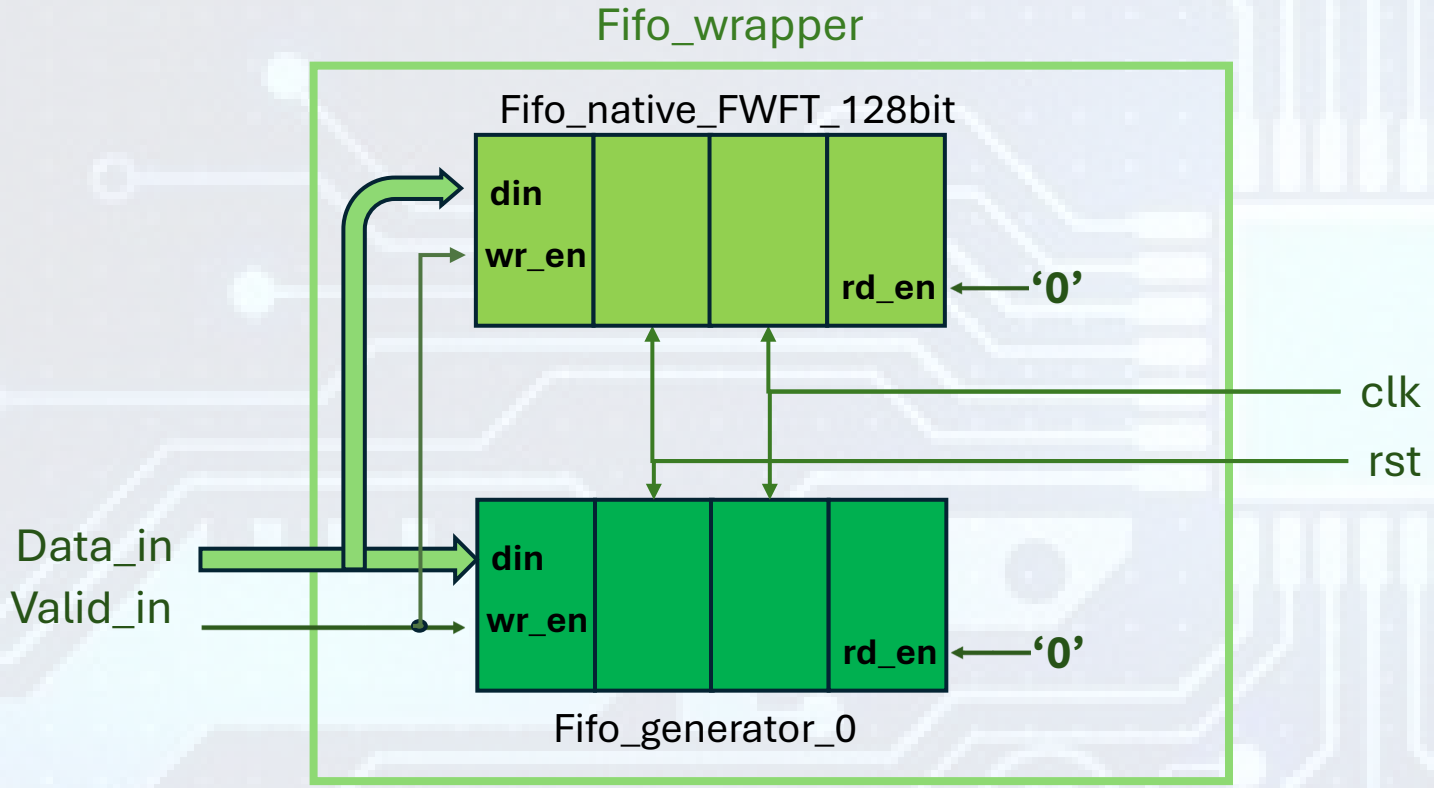
- Modify fifo_generator_0 with the following configuration:
  - Fifo implementation: Common Clock Distributed RAM — Basic tab
  - Write/Read Width: 128 bits — Native Ports tab
  - Write Depth: 16 — Native Ports tab
  - Write port handshaking - Overflow: Enabled — Status flag tab
  - Read port handshaking - Valid Flag: Enabled — Status flag tab
  - Read port handshaking - Underflow Flag: Enabled — Status flag tab
  - Data count Enabled — Data Counts tab
- Create another FIFO IP core, named fifo_native_FWFT_128bit, with the same configuration as fifo_generator_0, except
  - First-Word Fall-Through : Enabled — Native Ports tab

Double-click fifo_generator_0 to open the Customize IP dialog.

Tip: Right-click fifo_generator_0 and select “Copy IP..." to duplicate the existing FIFO IP core and modify only the necessary settings.

FIFO project

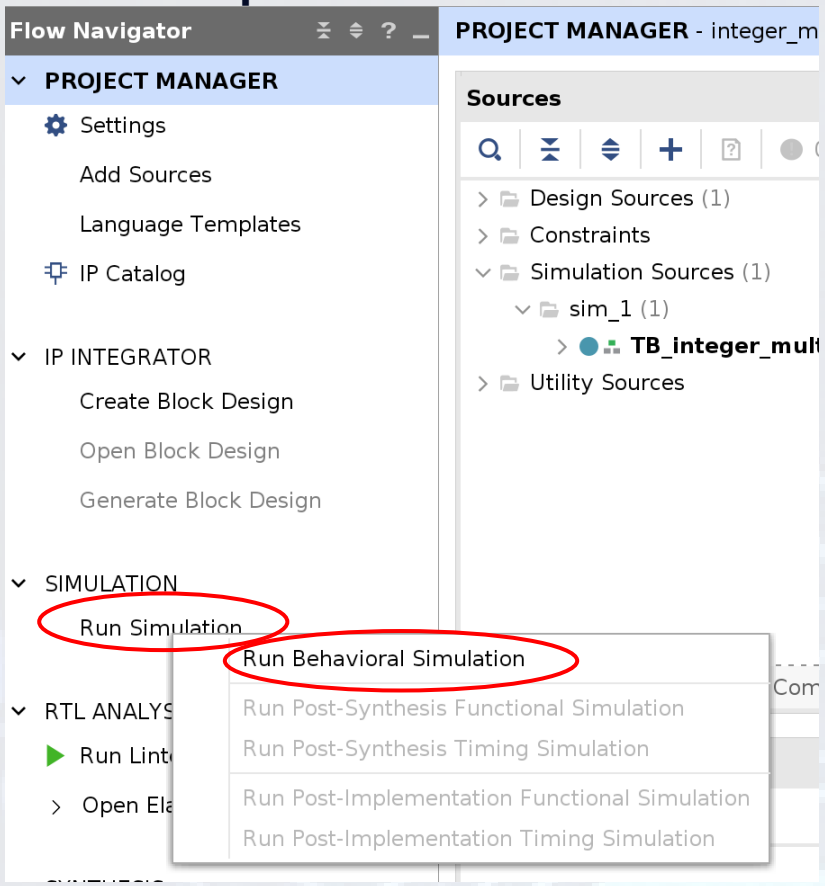
- Update the fifo_generator_0 component and its instance in the top-level VHDL module (fifo_wrapper)
- Add fifo_native_FWFT_128bit to the top-level VHDL module (fifo_wrapper)
- Connect the FIFOs as shown below (with all outputs left unconnected)
- Add the provided testbench (/sim/TB_fifo_wrapper_exercise_1.vhd) and waveform Configuration File testbench (/sim/tb_fifo_wrapper_behave.wcfg) .
- Run Behavioral simulation



Quick Guide to Vivado GUI

Simulation

Open simulation



Run simulation

Restart simulation

Add signals to waveform

- In the Objects window, locate the signal you want to observe
- Right-click on the signal and select "Add to Wave Window" or drag and drop it!
- Run or restart simulation to see signal transitions

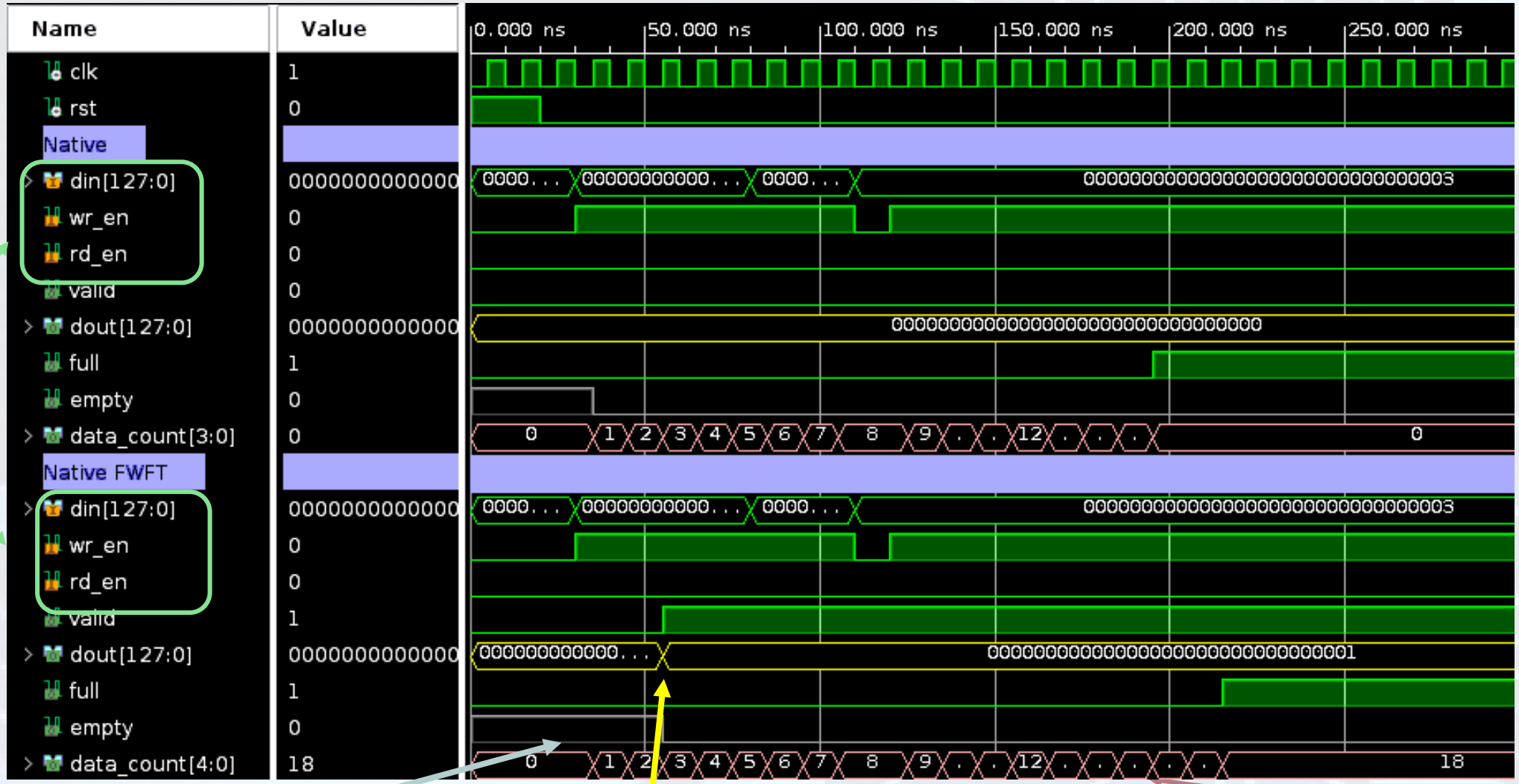
Add divider to waveform

- Right-click on the waveform background (the empty area where signals are shown).
- Select "New Divider"
- Drag and drop signals into groups separated by dividers for better organization.

Manage signal display

- Right-click on the signal in the waveform
  - Rename
  - Signal Color
  - Radix

FIFO project simulation



Same inputs

The empty flag has latency

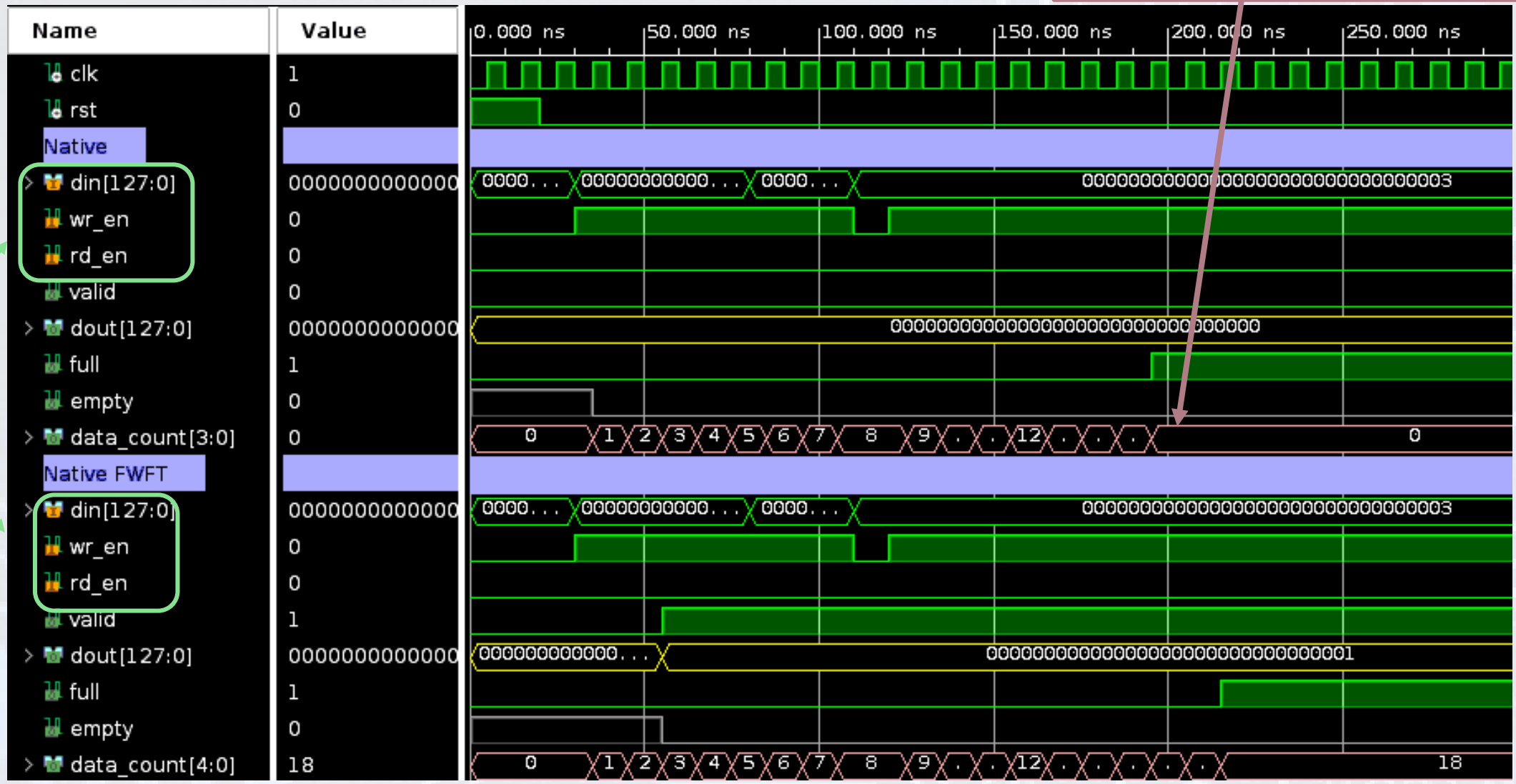
FIFO shows the first written data

first-word fall-through (FWFT) implementation increases the depth of the FIFO by 2 read words.

FIFO project simulation

The data_count signal reads 0 even when there are 16 words stored in the FIFO

Same inputs

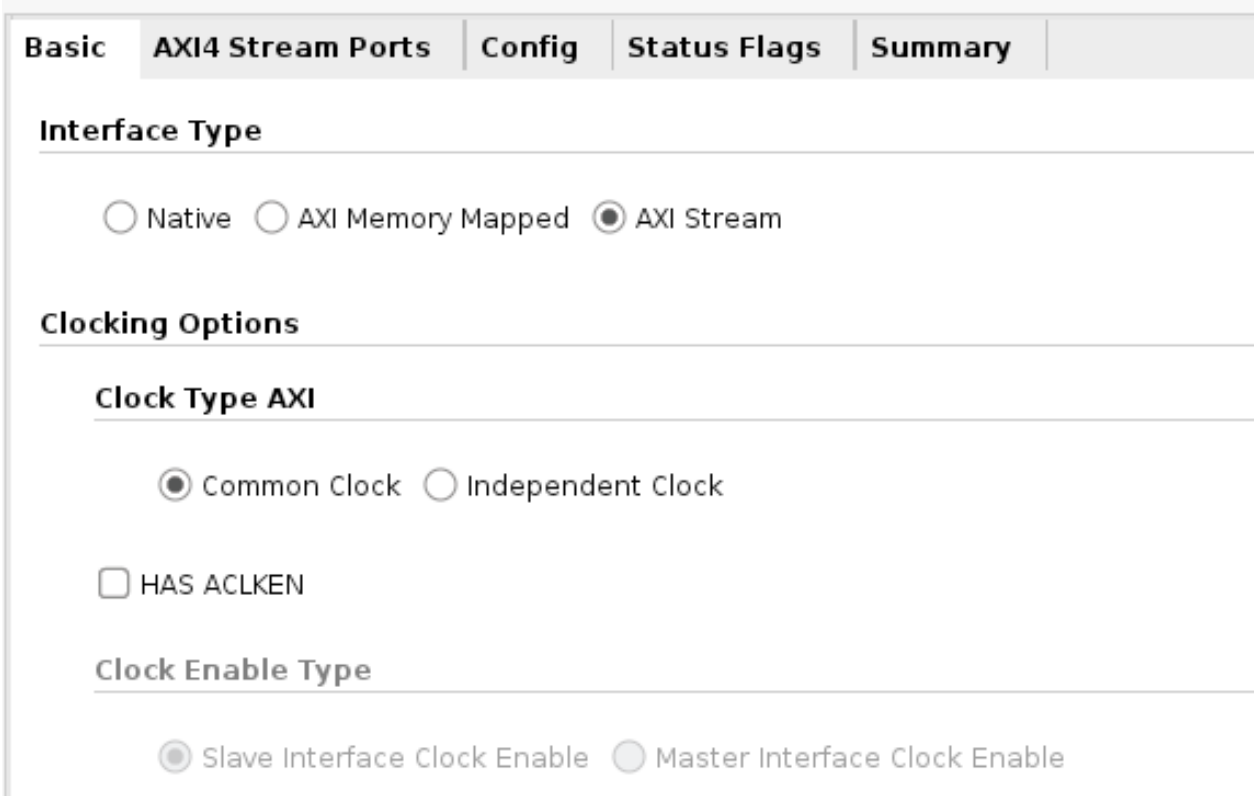


Vivado AXI-Stream FIFO Configuration

FIFO Generator core is a fully verified first-in first-out (FIFO) memory queue

Supported Interfaces
- Native
- **AXI Stream**
- AXI MemoryMapped

AXI FIFOs operate only in First-Word Fall-Through mode
- two additional read words added to the FIFO depth.



AXI interface overview

AXI is part of ARM AMBA, a family of micro controller buses first introduced in 1996.
https://docs.amd.com/v/u/en-US/ug1037-vivado-axi-reference-guide

- **AXI4**: For high-performance memory-mapped requirements
- **AXI4-Lite**: For simple, low-throughput memory-mapped communication
  - ❑ Provides separate data and address connections for reads and writes, which allows simultaneous, bidirectional data transfer.
  - ❑ Requires a single address and then bursts up to 256 words of data
- **AXI4-Stream**: For high-speed streaming data
  - ❑ Defines a single channel for transmission of streaming data.
  - ❑ Can burst an unlimited amount of data.

AXI-Stream overview

The AXI interface protocol uses a two-way valid and ready handshake mechanism.

- The information source uses the valid signal to show when valid data or control information is available on the channel.
- The information destination uses the ready signal to show when it can accept the data

AXI4-Stream FIFO Generator Interface Signals

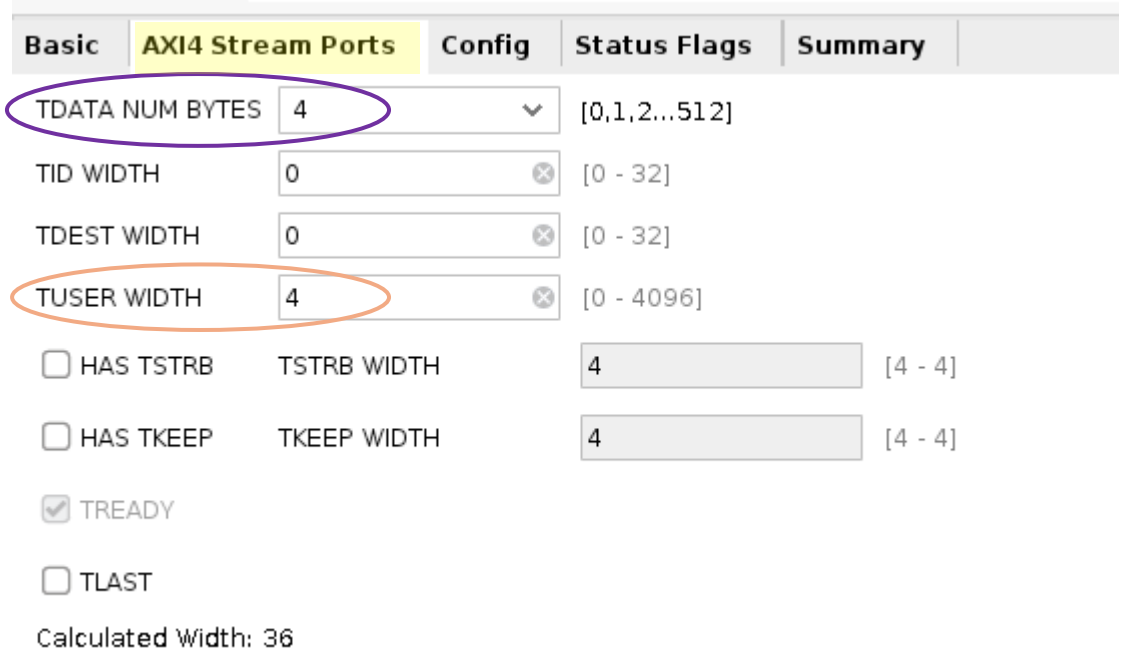
Name	Direction	Description
AXI4-Stream Interface: Handshake Signals for FIFO Write Interface		
s_axis_tvalid	Input	TVALID: Indicates that the master is driving a valid transfer. A transfer takes place when both TVALID and TREADY are asserted.
s_axis_tready	Output	TREADY: Indicates that the slave can accept a transfer in the current cycle.
AXI4-Stream Interface: Information Signals Mapped to FIFO Data Input (din) Bus		
s_axis_tdata[m-1:0]	Input	TDATA: The primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes.
s_axis_tuser[m:0]	Input	TUSER: The user-defined sideband information that can be transmitted alongside the data stream.

Name	Direction	Description
AXI4-Stream Interface: Handshake Signals for FIFO Read Interface		
m_axis_tvalid	Output	TVALID: Indicates that the master is driving a valid transfer. A transfer takes place when both tvalid and tready are asserted.
m_axis_tready	Input	TREADY: Indicates that the slave can accept a transfer in the current cycle.
AXI4-Stream Interface: Information Signals Derived from FIFO Data Output (dout) Bus		
m_axis_tdata[m-1:0]	Output	TDATA: The primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes.
m_axis_tuser[m:0]	Output	TUSER: The user-defined sideband information that can be transmitted alongside the data stream.

Vivado AXI-Stream FIFO Configuration

- **TSTRB**: The byte qualifier that indicates whether the content of the associated byte of TDATA is valid
- **TKEEP**: The byte qualifier that indicates whether the content of the associated byte of TDATA has to be trasfer
- **TLAST**: Indicates the boundary of a packet.

AXI4 Stream port tab



- **TDATA NUM BYTES**: The number of bytes transferred per cycle
- **TID**: Data stream identifier
- **TDEST**: Provides routing information for the data stream.

Hands-on

Exercise 2

FIFO project

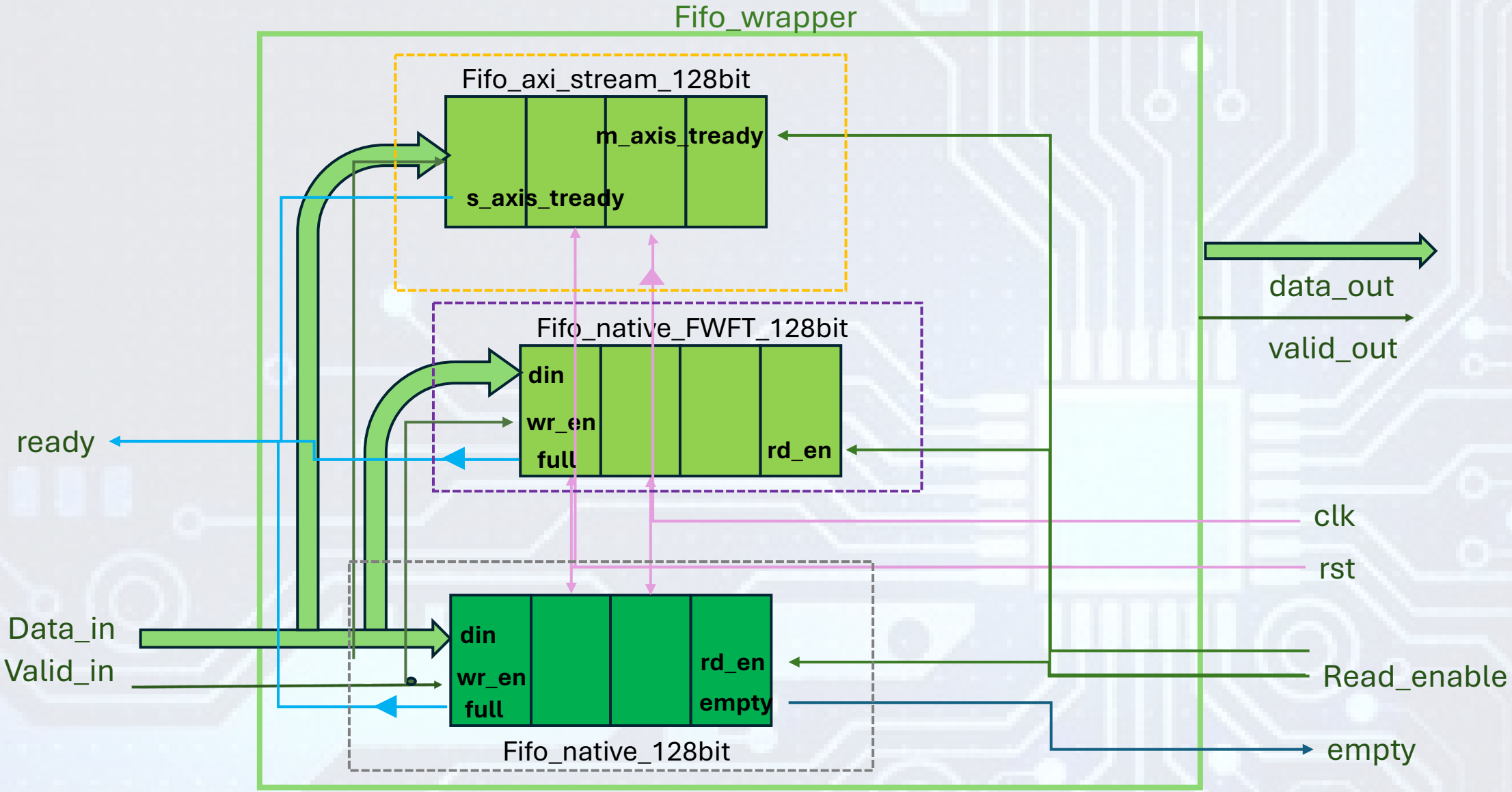
- Copy fifo_generator_0 in fifo_native_128bit
- Remove from the project fifo_generator_0
- Create another FIFO IP core, named fifo_AXI_stream_128bit, with the following configuration
  - Interface Type: AXI Stream } Basic tab
  - TDATA NUM BYTES: 16 } AXI4 stream Ports tab
  - FIFO Implementation type: Common Clock Distributed RAM
  - FIFO DEPTH:16

  } Config tab

FIFO project

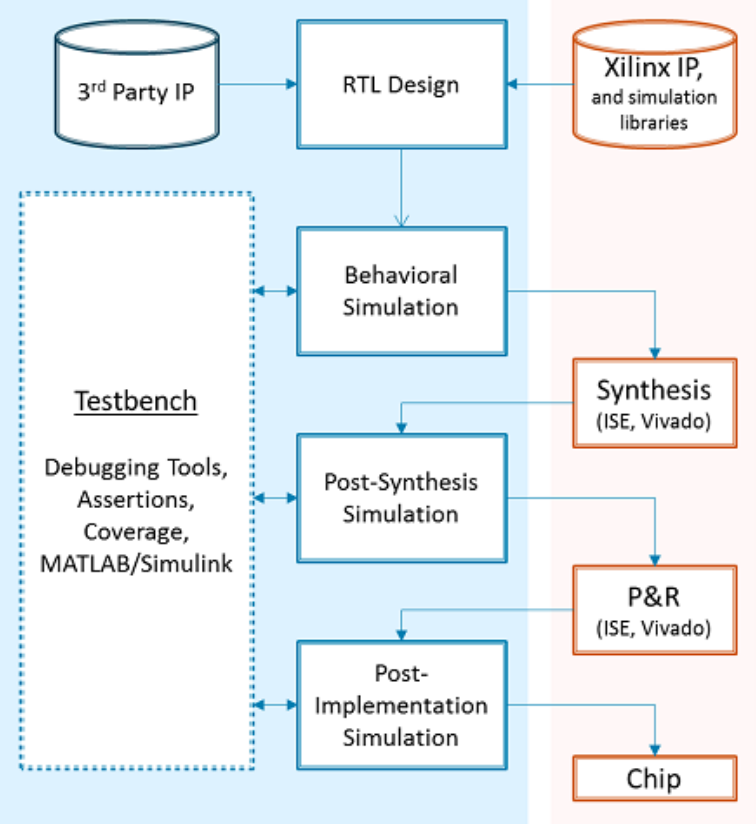
- Add to the top-level VHDL module:
  - An instance of fifo_native_128bit
  - An instance of fifo_AXI_stream_128bit;
  - Generic
    - FIFO_MODE: string - use this generic with a generate statement to select between the FIFO configurations (NATIVE FWFT / STANDARD NATIVE / AXI_STREAM)
  - Ports:
    - data_out : out std_logic_vector(31 downto 0);
    - valid_out : out std_logic
    - Read_enable: in std_logic (connected to NATIVE FIFO rd_en / AXI4 FIFO m_axis_tready)
    - Ready : out std_logic (connected to NATIVE FIFO not full/ AXI4 FIFO s_axis_tready)
    - Empty: out std_logic

FIFO project



Vivado Simulation

Simulation verifies the functional correctness of the design before hardware implementation.



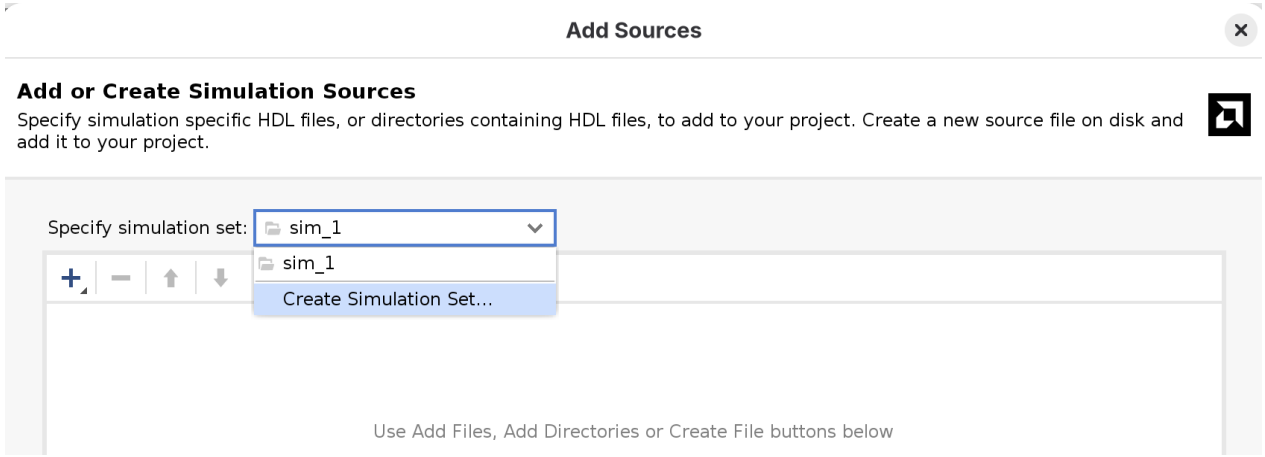
- Vivado provides a simulator to check functionality throughout the design process

Simulation Type	Stage	Timing Info	Purpose
Behavioral	Pre-synthesis	No	Functional correctness
Post-Synthesis	After synthesis	Minimal	Verify RTL vs synthesized netlist
Post-Implementation	After place & route	Yes	Functional + timing correctness

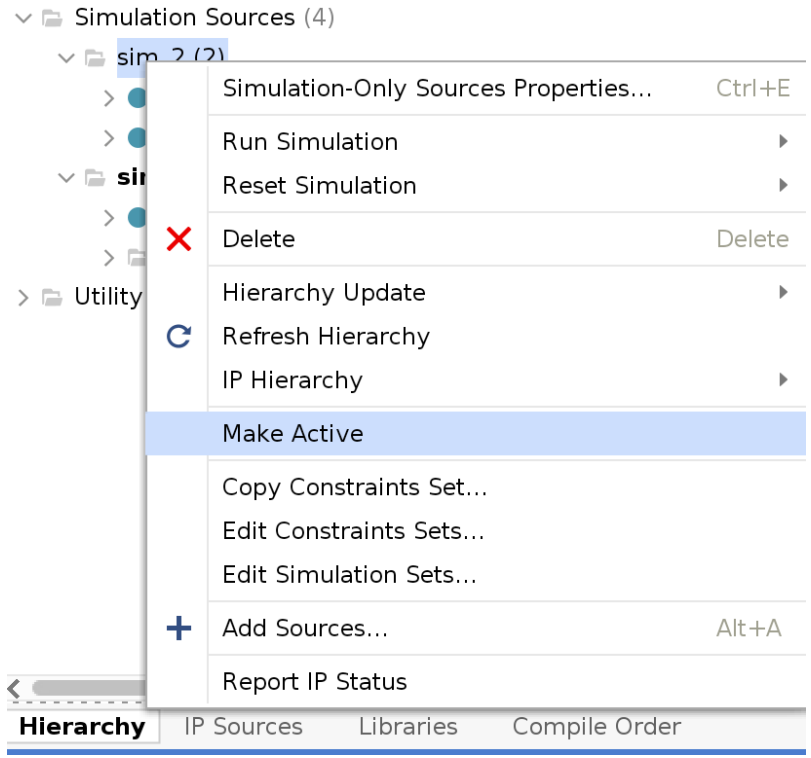
Vivado Simulator

- Simulation sources are organized into **simulation sets** in the Sources window
- Different simulation sets enable verification with multiple independent testbenches

- Simulation sources can be assigned to a specific set when added to the project



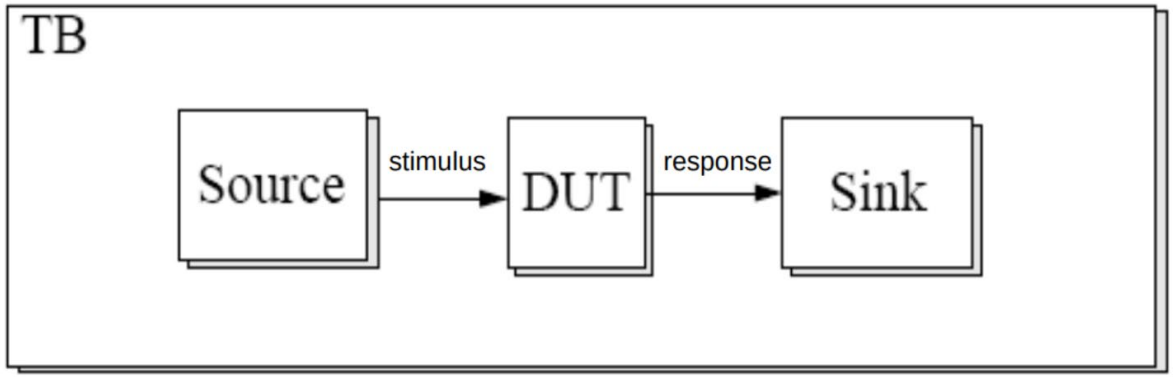
- Only one set can be active at a time



- Vivado simulator does not support waveform tracing of some HDL objects, such local variables.

VHDL TestBench

VHDL **test bench** (**TB**) is a piece of code meant to verify the functional correctness of HDL model



The source provides input to the DUT in several ways:

- Assign constant values - Useful for small, predefined test cases.
- Read values stored in a separate file - Useful for larger test datasets or data generated externally; allows flexibility and easy updates without modifying the code
- Algorithmically ”on-the-fly - Input vectors are created dynamically during simulation

The Sink collects DUT output, which should be verified against expected results:

- Expected response must be known exactly
- Comparison between DUT output and the expected response can be performed automatically
- Responses can be optionally saved to files for offline analysis or debugging.

VHDL TestBench

Entity (**no port**)

```
ENTITY test_tb IS
END test_tb;
```

Architecture

```
ARCHITECTURE structural OF test_tb
IS
  component entity_to_test is
       …
  end component;
-- Signal/constant declaration
…
BEGIN
-- Stimulus generator
…

DUT: enity_to_test
port map(
….
);
-- Response cheker
…
END behavior;
```

Components/signals declaration

Stimulus generation

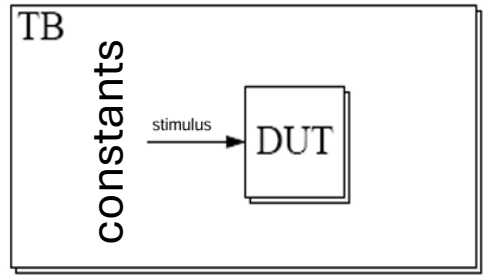
Component instantiation

Sink

- Making realistic TB is sometimes hard
- Verification cannot prove correctness: it can show the existence of bugs, but not their non-existence!
- Test bench may have mistakes
  - False interpretation
  - Test bench codes may have bugs
- Good to have different persons writing the actual code and test bench

VHDL Simple TestBench

- **Simple TB**:
  - Instantiates the design under test (DUT)
  - Generates stimulus
    - Not automatically– handwritten code trying to spot corner cases
    - Poor reusability
    - Provides limited verification coverage

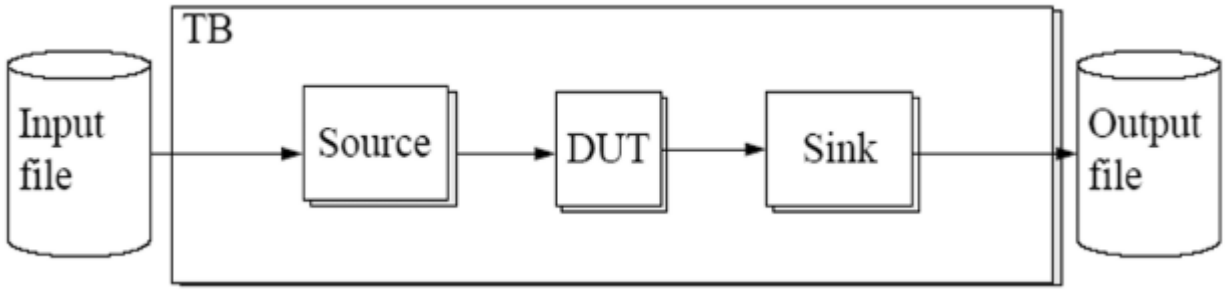


  - Verifying DUT output by manually inspecting simulation waveforms is error-prone and unreliable, especially for large datasets or subtle corner cases. Automatic checking ensures correctness, repeatability, and efficient detection of mismatches

Suitable only for very simple designs!

File- based TestBench

- Stimulus for DUT is read from an input file and modified in the source component
- The response is modified in the sink and written to the output file



For each stimulus file, the designer can prepare the expected output trace.
It can be automatically compared to the response of DUT, either in VHDL or using command line tool diff for file.

Not synthesizable!

File- based TestBench

1. **Required package**

To use file operations, you must include:

```
use std.textio.all;
use ieee.std_logic_textio.all;
```

2. **Declaring a file inside architecture**

```
file input_file : text open read_mode  is “in_file.txt";
file output_file: text open write_mode is “out_file.txt";
```

- read_mode → open file for reading.
- write_mode → open file for writing.

3. **Working with lines**

Add an intermediate type line to process file content.

```
variable L : line;
```

4. **A) Reading from a file**

```
variable val : <type>;
while not endfile(input_file) loop   -- keep reading until file ends
        readline(input_file, L);     -- get one line from file
        hread/read(L, val);          -- extract <type> from line
end loop;
```

4. **B) Writing to a file**

```
variable val : <type>;
write(L, val);                  -- write val into line
writeline(output_file, L);      -- flush line to file
```

File- based TestBench

read

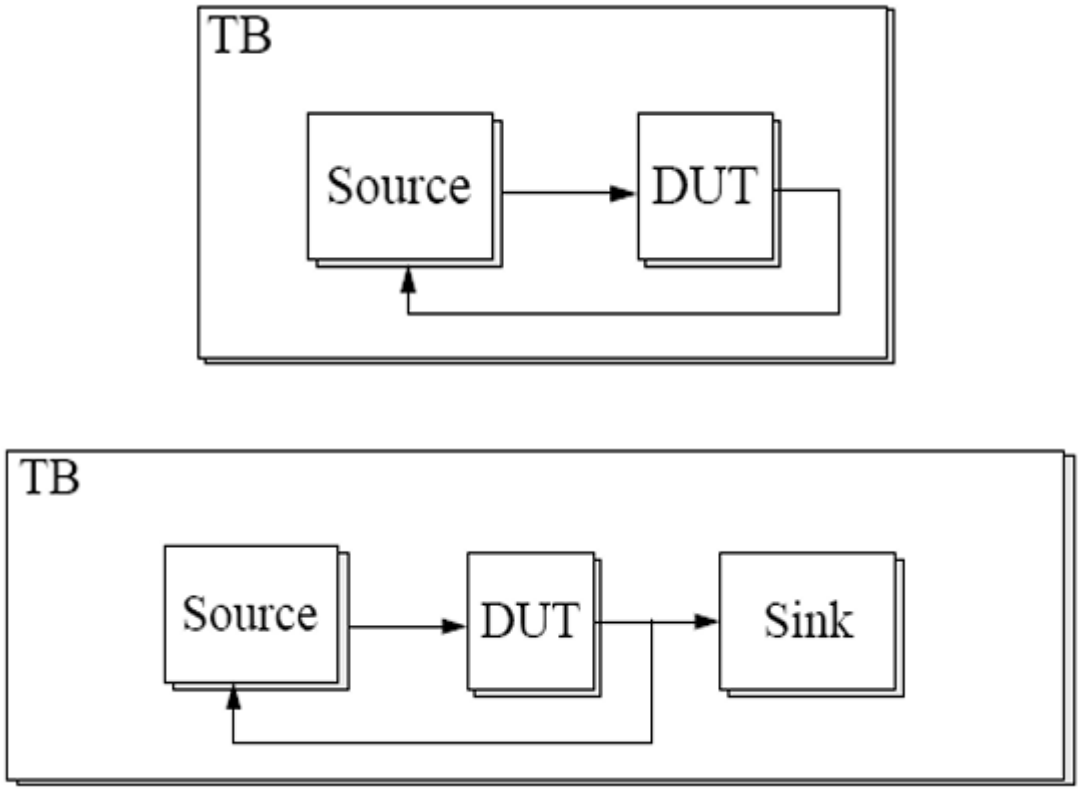
- Part of the textio library.
- Reads standard data types such as integer, real, bit, etc.

hread

- Part of the std_logic_textio library.
- Used to read std_logic or std_logic_vector values from a text file.
- Can read values in hexadecimal ("0F") or binary ("1010") notation.

VHDL Smart Test Bench

Circuit's response affects further stimulus - TB is reactive
E.g., source writes FIFO (=DUT) until it is full, then it does something else…

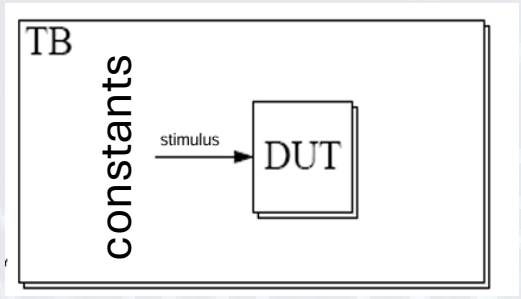


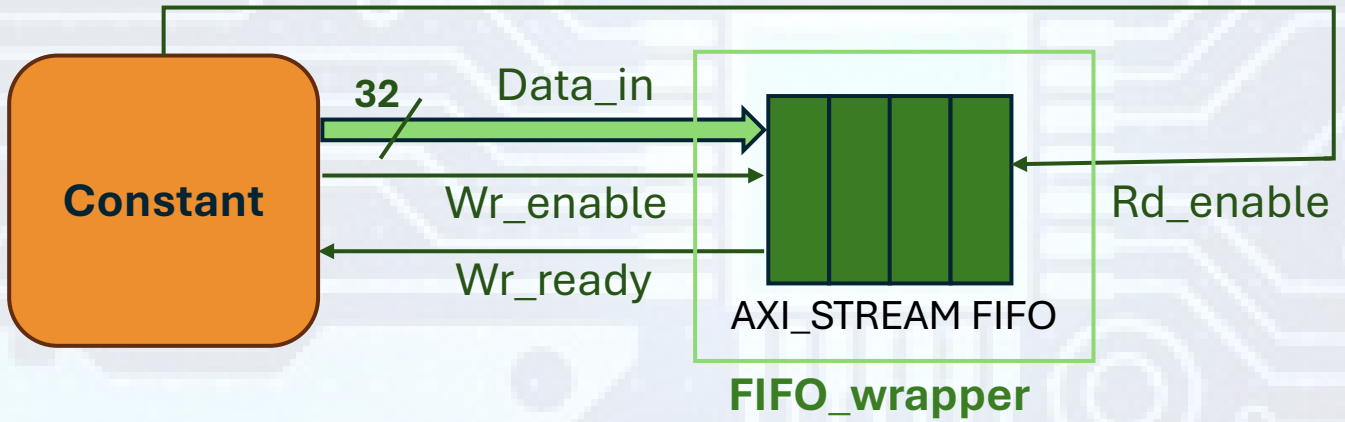
Hands-on

Exercise 3

FIFO Project simulation

- Use the simple testbench TB_fifo_wrapper_constant.vhd available in /sim to simulate the FIFO_wrapper entity configured in AXI_STREAM mode (add the TB in a new simulation_set named “sim_constant”)
- Remember to activate new simulation set!





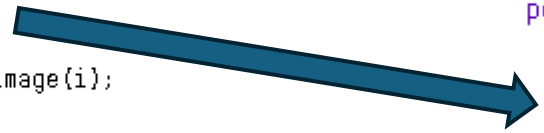
FIFO Project simulation

TB_fifo_wrapper_constant.vhd

```
-- Define data
type packet_array is array (natural range <>) of std_logic_vector(DATA_WIDTH-1 downto 0);
constant test_packet : packet_array := (
      (DATA_WIDTH-1 downto 32 => '0') & x"a0020001", -- DATA 0
      (DATA_WIDTH-1 downto 32 => '1') & x"50020001", -- DATA 1
      (DATA_WIDTH-1 downto 32 => '0') & x"050f0001", -- DATA 2
      (DATA_WIDTH-1 downto 32 => '0') & x"80020e01", -- DATA 3
      (DATA_WIDTH-1 downto 32 => '0') & x"300b0001", -- DATA 4
      (DATA_WIDTH-1 downto 32 => '0') & x"a0024001", -- DATA 5
      (DATA_WIDTH-1 downto 32 => '0') & x"07080001"  -- DATA 6
 );
```

- **integer'image** is a built-in VHDL function that converts an integer value into a string representation that can be printed or reported.

```
-- It sends the test data words (from test_packet) to the FIFO input, one per clock cycle,
-- whenever the FIFO is ready to accept them (ready = '1').
stimulus_process : process
    variable i : integer := 0;
begin
    wait until rst = '0';
    wait until rising_edge(clk);

    while i < test_packet'length loop
        if ready = '1' then
            data_in   <= test_packet(i);
            wr_enable <= '1';
            report "TX sent word " & integer'image(i);
            i := i + 1;
        else
            wr_enable <= '0';
        end if;
        wait until rising_edge(clk);
    end loop;

    wr_enable <= '0';
    wait;
end process;
```



```
uut_axi_stream : entity work.fifo_wrapper
 generic map (
    data_width => DATA_WIDTH,
    FIFO_MODE  =>  "AXI_STREAM"
 )
 port map (
    clk        => clk,
    rst        => rst,
    data_in    => data_in,
    valid_in   => wr_enable,
    ready      => ready,
    read_enable => re_enable,
    data_out   => data_out,
    valid_out  => valid
 );
```

FIFO Project simulation

TB_fifo_wrapper_constant.vhd

```
uut_axi_stream : entity work.fifo_wrapper
 generic map (
    data_width => DATA_WIDTH,
    FIFO_MODE  =>  "AXI_STREAM"
  )
  port map (
    clk        => clk,
    rst        => rst,
    data_in    => data_in,
    valid_in   => wr_enable,
    ready        => ready,
    read_enable => re_enable,
    data_out     => data_out,
    valid_out    => valid
  );
```



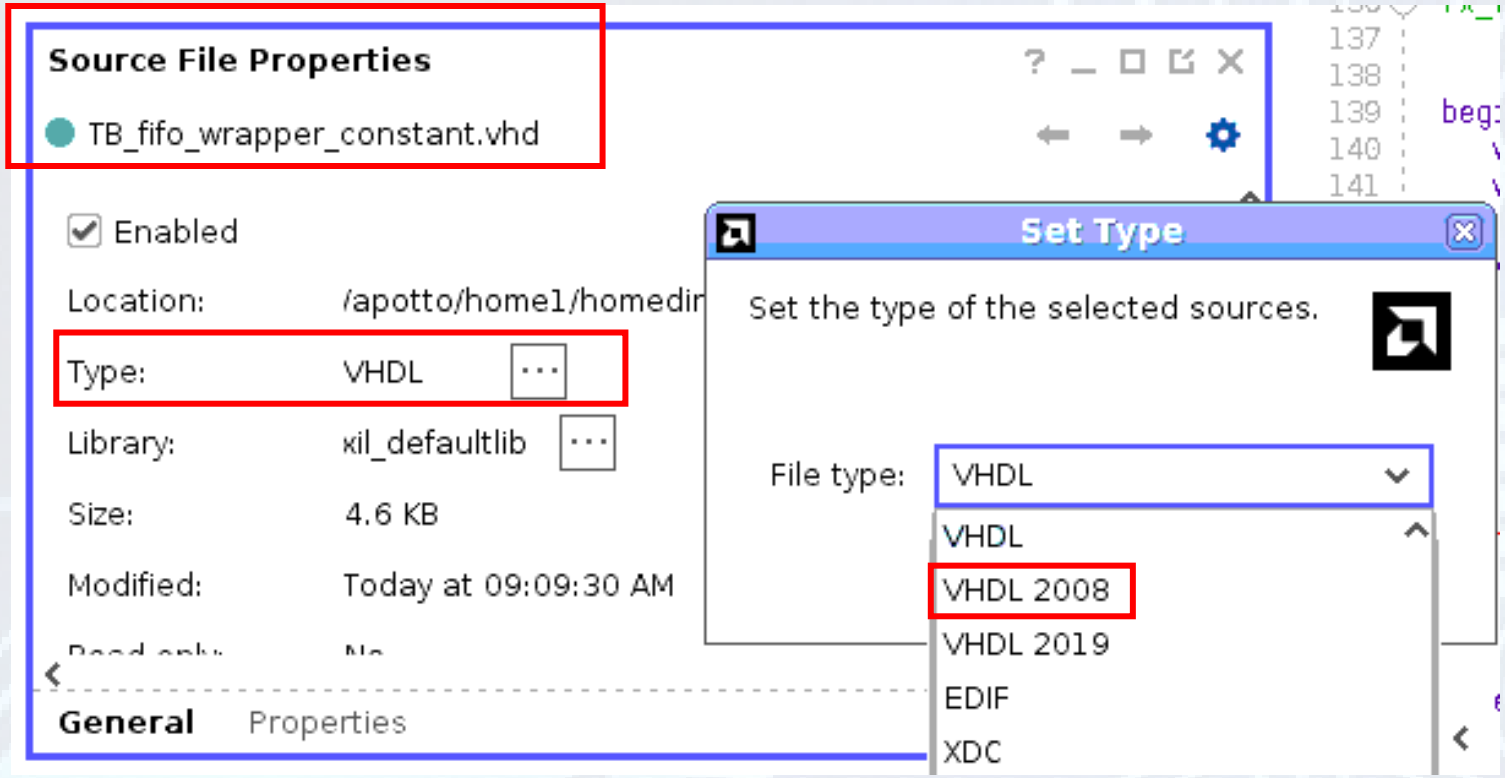
```
-- RX process: controls FIFO read and stop when expected words has been received
rx_read_process : process
        variable read_count     : integer := 0;
        constant expected_words : integer := test_packet'length;
begin
   wait until rst = '0';
   wait for CLK_PERIOD * 100;  -- Wait for RX data to start

   while read_count < expected_words loop
     wait until rising_edge(clk);

     if valid = '1' then
        re_enable <= '1';
        wait for CLK_PERIOD;
        re_enable <= '0';
        report "RX received word [" & integer'image(read_count) & "]: " & to_hstring(to_bitvector(data_out));
        read_count := read_count + 1;
     else
        re_enable <= '0';
     end if;
   end loop;
   -- All data received
   report "Simulation complete: All expected data received" severity note;
   stop(0);  -- Stop simulation cleanly
   wait;
 end process;
```

- **to_hstring** is a standard VHDL-2008 function (it doesn't exist in VHDL-93 or 2002). It converts a bit or logic vector into a hexadecimal string, very useful for readable messages in simulation.

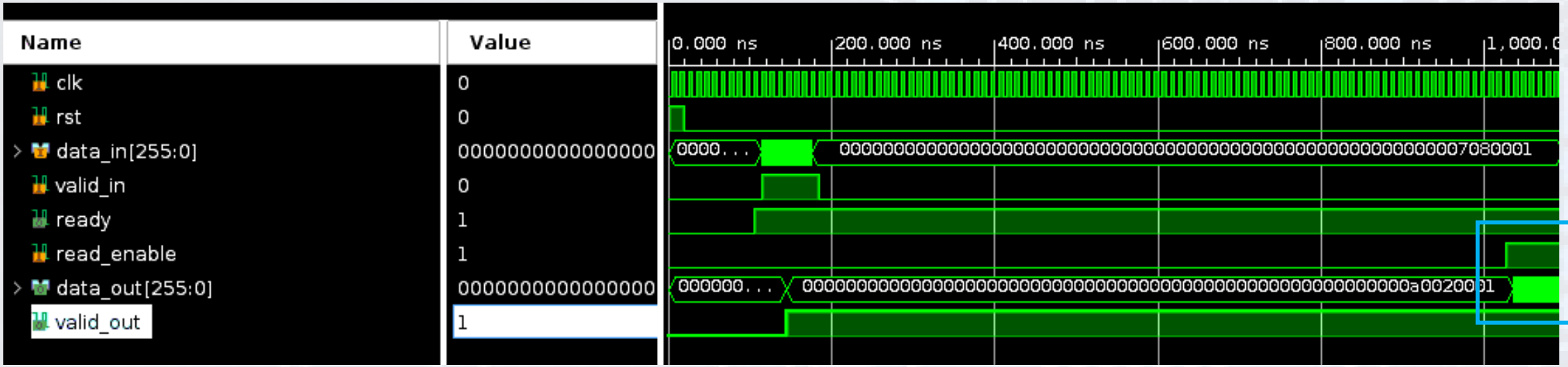
- The **stop** procedure from the std.env package terminates the simulation cleanly

FIFO Project simulation

To use Vivado-2008 **to_hstring** function, make sure that the file TB_fifo_wrapper_constant.vhd is set to VHDL-2008 language type.



FIFO Project simulation



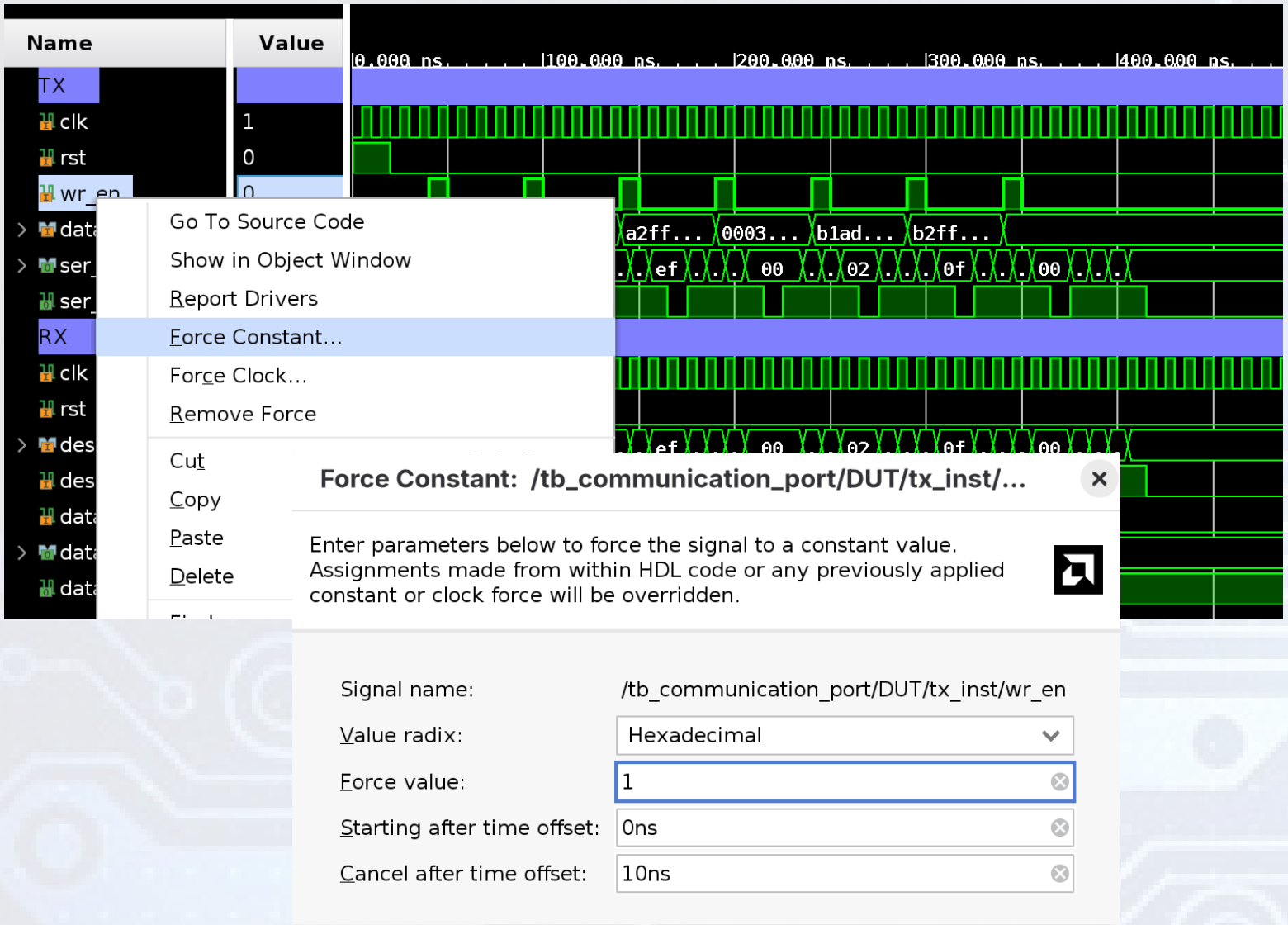
Tcl Console

```
Note: RX received word [0]: 0000000000000000000000000000000000000000000000000000000000000000A0020001
Time: 1035 ns  Iteration: 0  Process: /tb_fifo_wrapper/rx_read_process  File: /apotto/homel/
Note: RX received word [1]: FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF50020001
Time: 1045 ns  Iteration: 0  Process: /tb_fifo_wrapper/rx_read_process  File: /apotto/homel/
Note: RX received word [2]: 0000000000000000000000000000000000000000000000000000000000000050F0001
Time: 1055 ns  Iteration: 0  Process: /tb_fifo_wrapper/rx_read_process  File: /apotto/homel/
Note: RX received word [3]: 0000000000000000000000000000000000000000000000000000000000000080020E01
Time: 1065 ns  Iteration: 0  Process: /tb_fifo_wrapper/rx_read_process  File: /apotto/homel/
Note: RX received word [4]: 0000000000000000000000000000000000000000000000000000000000000300B0001
Time: 1075 ns  Iteration: 0  Process: /tb_fifo_wrapper/rx_read_process  File: /apotto/homel/
Note: RX received word [5]: 0000000000000000000000000000000000000000000000000000000000000A0024001
Time: 1085 ns  Iteration: 0  Process: /tb_fifo_wrapper/rx_read_process  File: /apotto/homel/
Note: RX received word [6]: 0000000000000000000000000000000000000000000000000000000007080001
Time: 1095 ns  Iteration: 0  Process: /tb_fifo_wrapper/rx_read_process  File: /apotto/homel/
Note: Simulation complete: All expected data received
```

Drawbacks

- Verifying correctness is difficult, typically done via waveform inspection or Tcl console
- Providing many input vectors is time-consuming

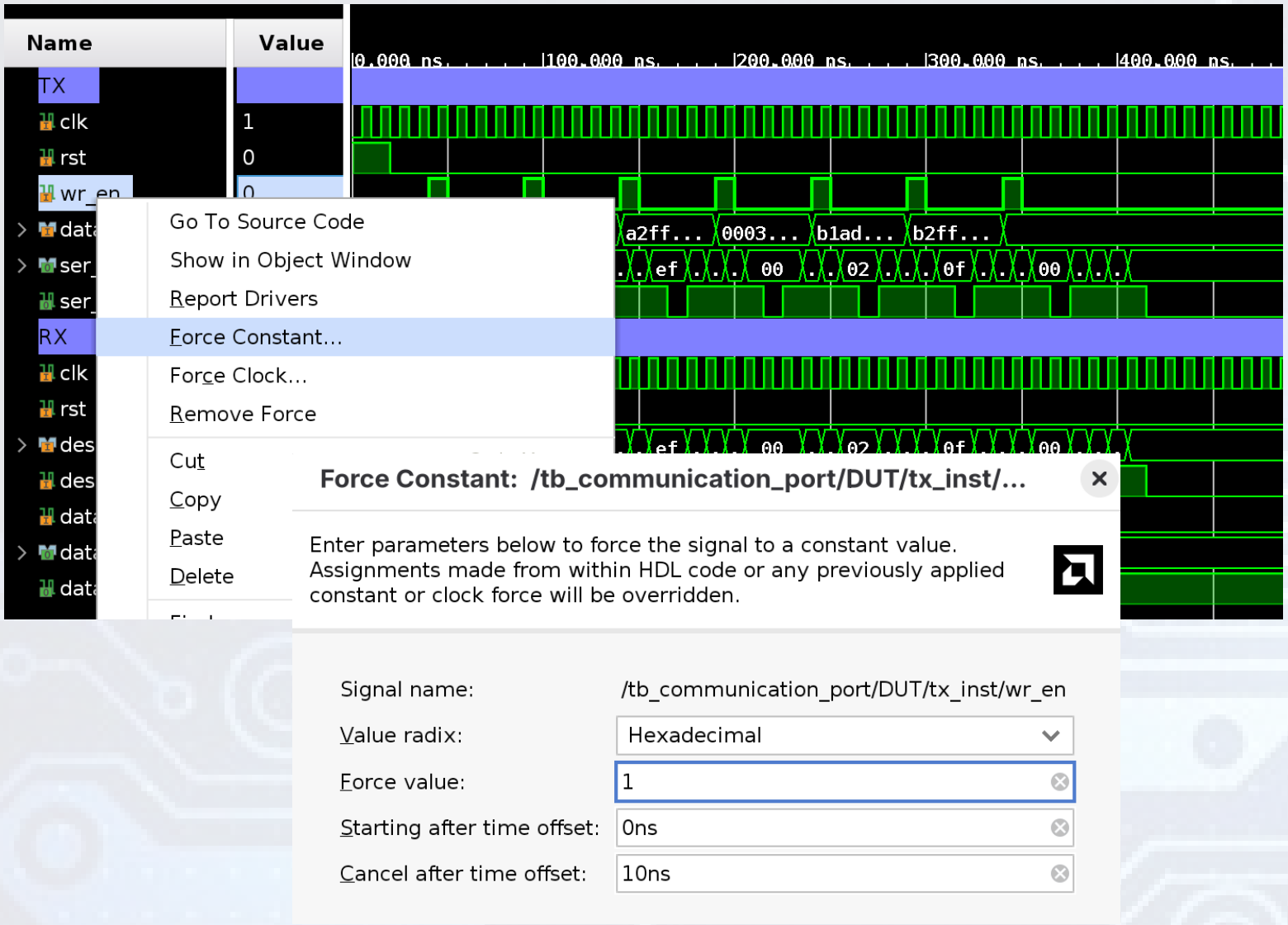
FIFO Project simulation



Input Handling Solution:

- Use the **Force Constant** option (available from wave window right-click menu)- allows fixing a signal to a constant value, overriding the assignments made within the HDL code
  - ❑ simple but time-consuming and not easily reproducible

FIFO Project simulation



Input Handling Solution:

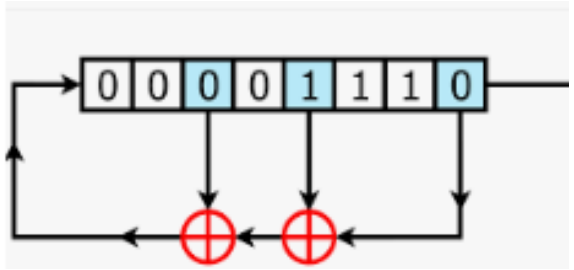
- Use the **Force Constant** option (available from wave window right-click menu)- allows fixing a signal to a constant value, overriding the assignments made within the HDL code
  - ❑ simple but time-consuming and not easily reproducible

Input Handling Solution 2:

- Automatically create input vectors

LFSR

Linear Feedback Shift Register (LFSR) is a shift-register whose input bit is a linear function of its previous state.



- The initial value of the LFSR is called **seed**
- The operation of the register is deterministic, so the stream of values produced by the register is completely determined by its current (or previous) state.

- An LFSR with a well-chosen feedback function can produce a sequence of bits that appears random and has a very long cycle.
  - Applications of LFSRs include generating pseudo-random numbers

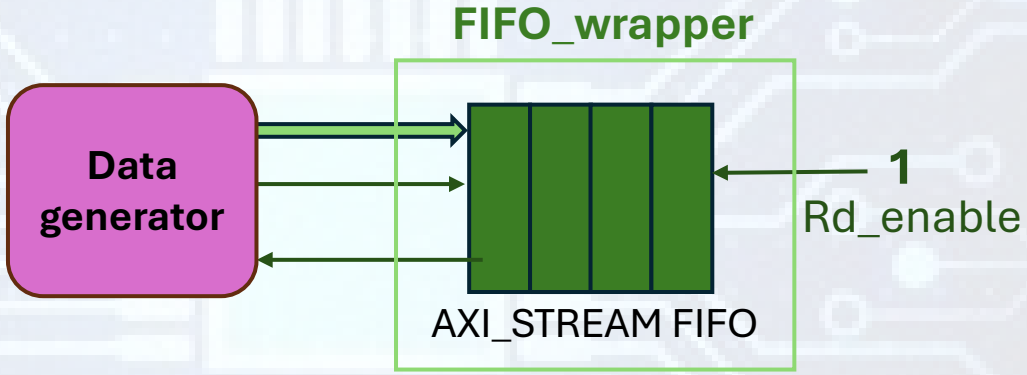
Hands-on

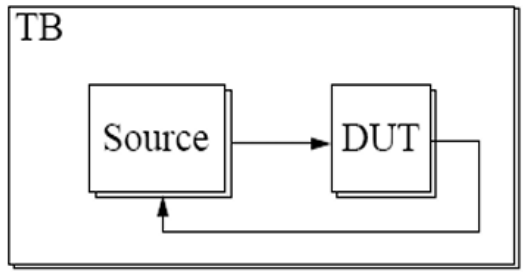
Exercise 4

FIFO Project simulation

- Create a new testbench (in a new simulation set called sim_lfsr) for the top-level module including the **Data_generator** component with its package (provided in /src directory).
  - generates NUM_DATA pseudo random data using a LFSR.
  - Stops writing when wr_ready is deasserted
- Activate the new simulation set.
- Run the simulation

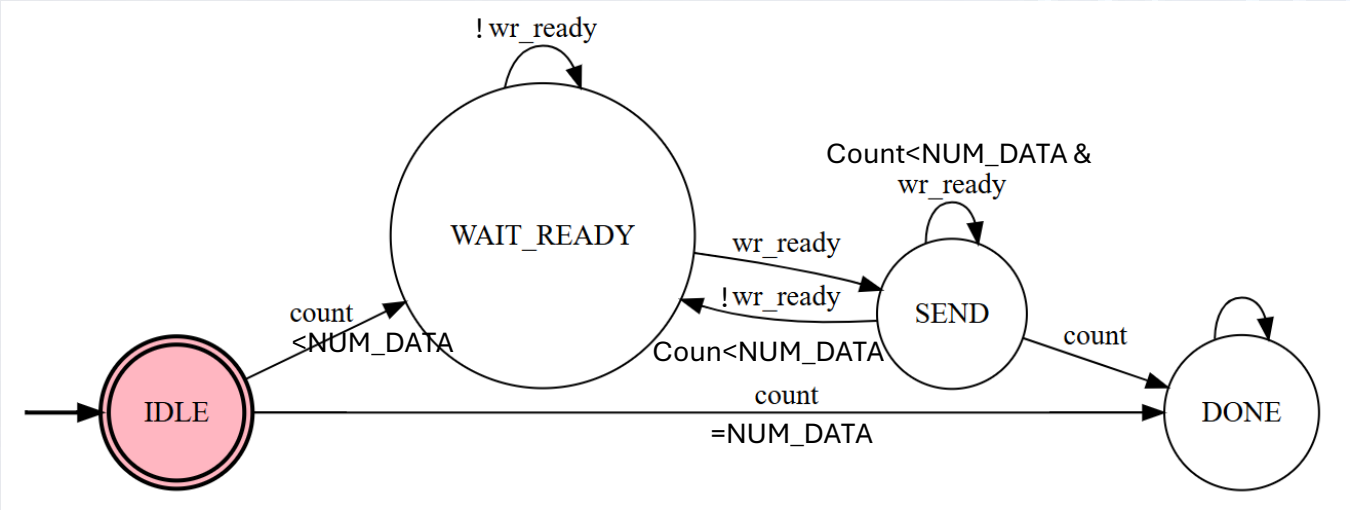
```
entity data_generator is
 generic (
   NUM_DATA      : integer := 2
   DATA_WIDTH    : integer := 32);
 port (
   clk       : in  std_logic;
   rst       : in  std_logic;
   wr_ready  : in  std_logic;
   data_out  : out std_logic_vector(DATA_WIDTH -1 downto 0);
   wr_en     : out std_logic );
end entity;
```





FIFO Project simulation

Data_generator.vhd




```
process(clk, rst)
begin
  if rst = '1' then
    t_data   <=  init_data(DATA_WIDTH, MODE);
    count  <= 0;
  elsif rising_edge(clk) then
    if (new_value = '1' and state=SEND) then
     count <= count + 1;
      t_data <= generate_next_data(t_data, MODE);
    end if;
  end if;
end process;
```

- The new value is calculated, according to the MODE constant, by the **generate_next_data** function defined in data_generator_pkg

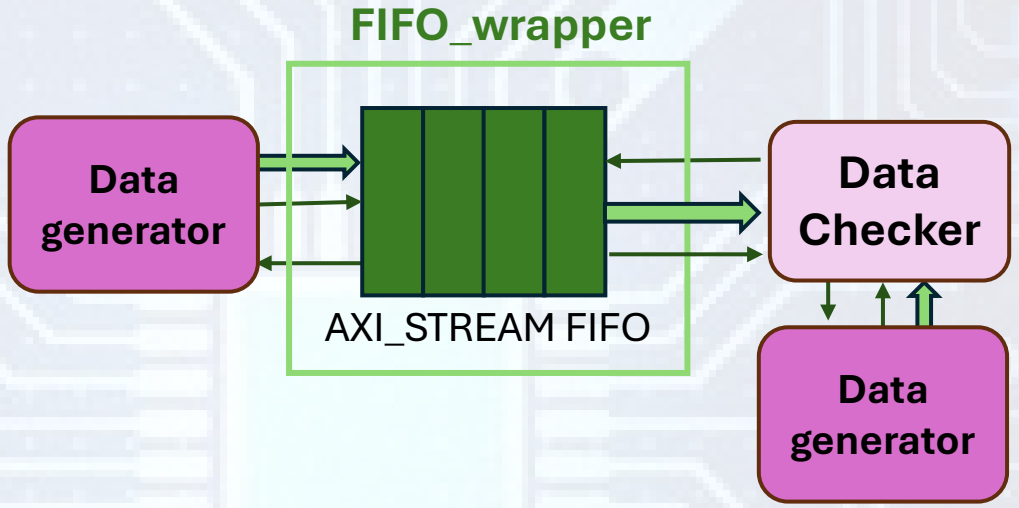
Hands-on

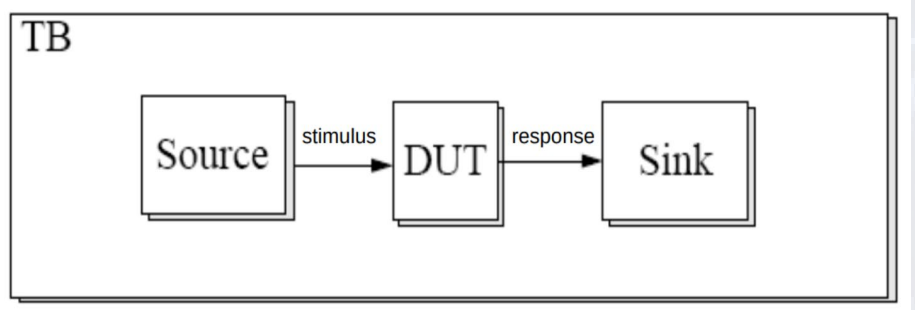
Exercise 5

FIFO Project simulation

- In the new testbench add a data_checker component to verify FIFO output against the expected sequence (generated by a second Data Generator instantation)
- Run behavioral simulation

```
entity data_checker is
  generic (
    DATA_WIDTH: integer := 5;
    NUM_DATA : integer := 2 );
 port (
  clk           : in  std_logic;
  rst           : in  std_logic;
  valid         : in  std_logic;
  data_in       : in  std_logic_vector(DATA_WIDTH-1 downto 0);
  re_enable     : out std_logic;
  test_ok       : out std_logic;
  data_received : out std_logic;
  expected_data : in std_logic_vector(DATA_WIDTH-1 downto 0);
  expected_valid: in std_logic;
  ready          : out std_logic );
end entity;
```





Advantages:

- Automatic verification of outputs against expected values
- Immediate detection of mismatches with error reports

Hands-on

Exercise 6

FIFO Project simulation

Create new simulation set adding:

- **Data_Reader** (provided in /sim):
  - Reads a data (std_logic_vector/integer) from a file (file_data)
  - Sends the data as std_logic_vector output (with length DATA_WIDTH)
  - Waits for the handshake signal (receiver_ready) before reading the next data

```
entity data_Reader is
 generic (
        FILE_TYPE       : string  := "integer";
        FILE_DATA       : string  := "file_data.txt";
        DATA_WIDTH      : integer := 32
    );
 port (
 clk             : in  std_logic;
 rst             : in  std_logic;
 receiver_ready  : in  std_logic;
 data            : out std_logic_vector(DATA_WIDTH - 1 downto 0);
 valid           : out std_logic ;
 data_read       : out integer;  -- Number of data words read
 eof             : out std_logic
    );
end entity;
```

FIFO Project simulation

- Add **output_writer** (provided in /sim)**:**
  - Receives std_logic_vector (of length DATA_WIDTH) data
  - Converts and writes **number_of_data** data as integers to an output file (**file_out_name**)

```
entity output_writer is
  generic (
    DATA_WIDTH  : integer := 5;
    FILE_OUT_NAME : string  := "file_results.txt");
   port (
    clk            : in std_logic;
    rst            : in std_logic;
    send_complete        : in std_logic; -- Data transmission process has finished sending all data
    data_in    : in  std_logic_vector(DATA_WIDTH - 1 downto 0);
    valid_in   : in  std_logic;
    number_of_data       : in integer;   -- Number of data words to write
    data_written         : out std_logic -- Indicates data has been written
);
end entity;
```

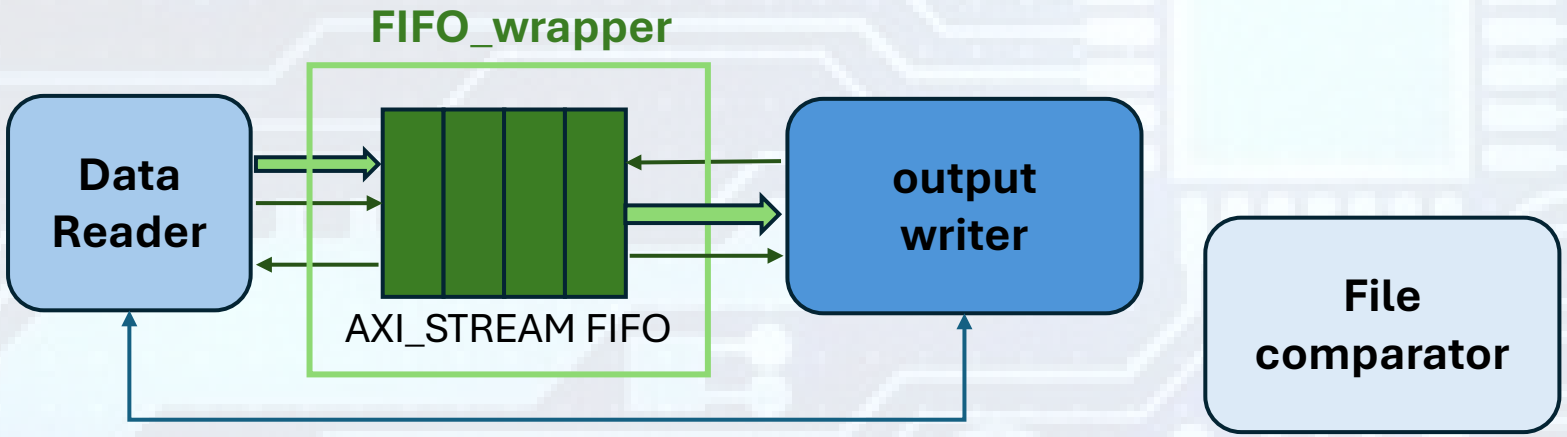
FIFO Project simulation

Add a comparator (provided in /sim):

- **file_comparator:**
  - Compare file_actual and file_expected
  - Check for mismatches

```
entity File_comparator is
  generic (
      FILE_EXPECTED : string := "file_in_0.txt";
      FILE_ACTUAL   : string := "file_in_1.txt");
  port (
      trigger: in std_logic; -- Start comparison when
                             -- asserted ('1')
      compare_ok : out std_logic);
end entity;
```

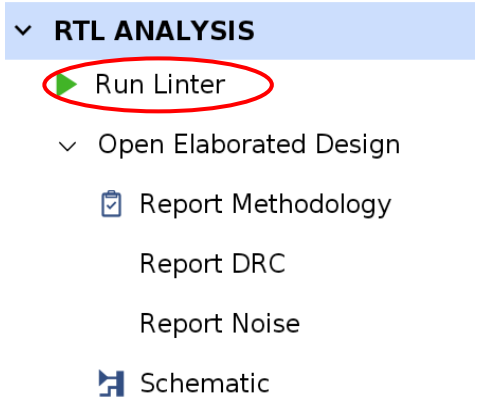
- Write a TB to connect all these entities
- Run simulation



Vivado RTL Linter

Vivado includes an **RTL linter** that checks for code pattern that, while legal, might cause potential issues in the design.

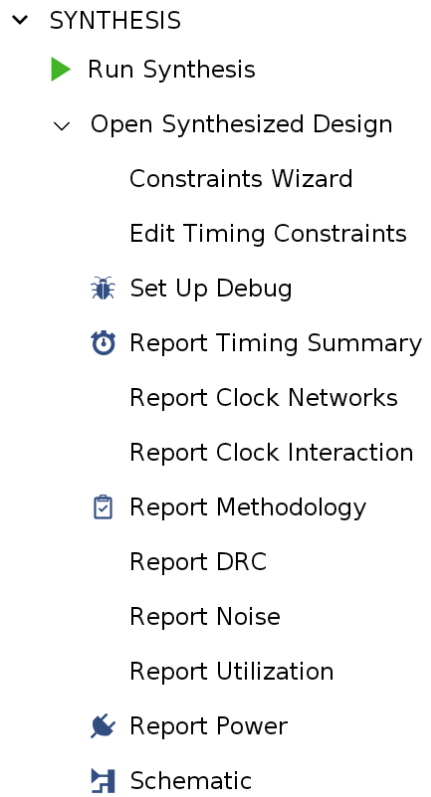
- Detects coding issues, unconnected signals, multiple drivers, inferred latches, and other potential RTL problems.
- Improves design quality
- Reduces debugging time during simulation and synthesis



https://docs.amd.com/r/en-US/ug901-vivado-synthesis/List-of-Linter-Rules

Vivado Synthesis

Synthesis is the process of transforming a Register Transfer Level (RTL) specified design into a gate-level representation.



Only a subset of VHDL is synthesizable — the part that can be mapped to hardware.

- **Synthesizable constructs:**
  - Clocked processes, combinational logic (if, case)
  - Registers, signals, arrays, fixed-range integers
- **Non-synthesizable constructs** (simulation only):
  - File I/O (textio), delays (wait for), assertions
  - Loops with dynamic or unknown bound
- Some construct, such as floating-point number or complicated operators, are too complex to be synthesized automatically.

- IEEE defines a subset of VHDL that is suitable for RT-level synthesis in IEEE standard 1076.6. E

Design Constraints

Design constraints specify the requirements that ensure correct functionality on the board.

- Over-constraining or under-constraining design makes timing closure difficult.
- If your project contains an IP that uses its own constraints, the corresponding constraint file does not appear in the constraints set.

- In Vivado constraints are usually specified in a **XDC** file (Xilinx Design Constraints).
- By default, all XDC files added to a constraint set are used for both synthesis and implementation.
- XDC constraints are applied sequentially and are prioritized based on clear precedence rules.

Synthesis Constraints

Categories of Synthesis Constraints:

- **RTL Attributes**
  - ❑ directives embedded in the HDL code; they usually choose the mapping style of certain part of the logic, preserving certain registers and nets, or controlling the design hierarchy in the final netlist.
    - DONT_TOUCH
    - BLACK_BOX
    - MARK_DEBUG

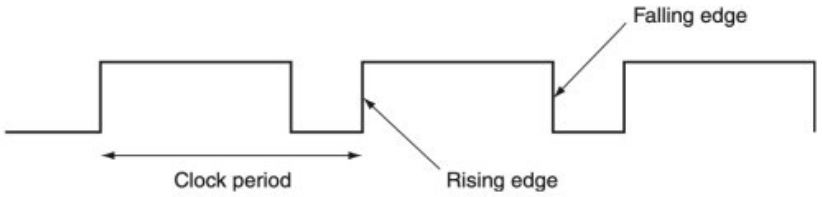
    https://docs.amd.com/r/en-US/ug901-vivado-synthesis/Synthesis-Attributes
- **Timing Constraints**
  - ❑ define clocks, delays, and timing requirements
    - create_clock
    - create_generated_clock
    - set_input_delay
    - set_output_delay
    - set_clock_groups
    - set_false_path
    - set_max_delay
    - set_multicycle_path

Synthesis Constraints

- **Create clock**

  A primary clock is a clock that defines a timing reference for the design

```
create_clock -period <arg> [-name <arg>] [-waveform <args>] [<objects>]
```



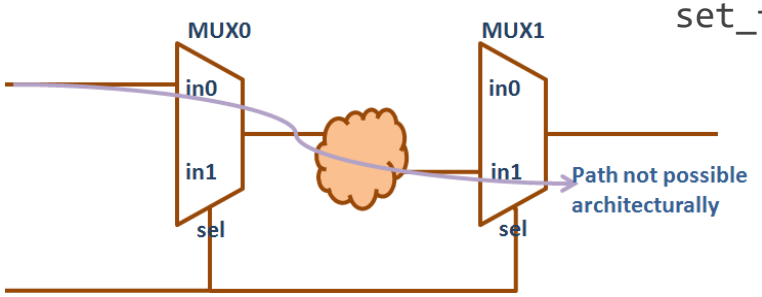
<**objects**>: List of clock source ports, pins or nets

Waveform: Clock edge specification (necessary to define a clock with a duty cycle other than 50%).

- **set false path**

  A false path is a logically existing but functionally non-existent connection in a circuit that is excluded from timing analysis to improve timing closure

```
set_false_path [-setup] [-hold] [-rise] [-fall] [-from <args>] [-to <args>]
```



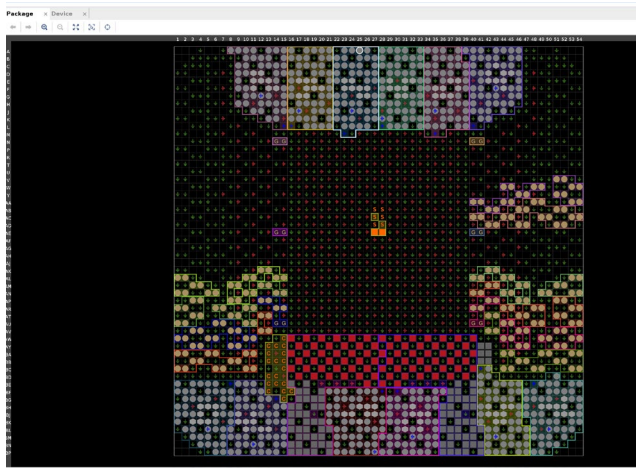
<**args**>: list of path startpoints/endpoint or clocks

Vivado Synthesis Results

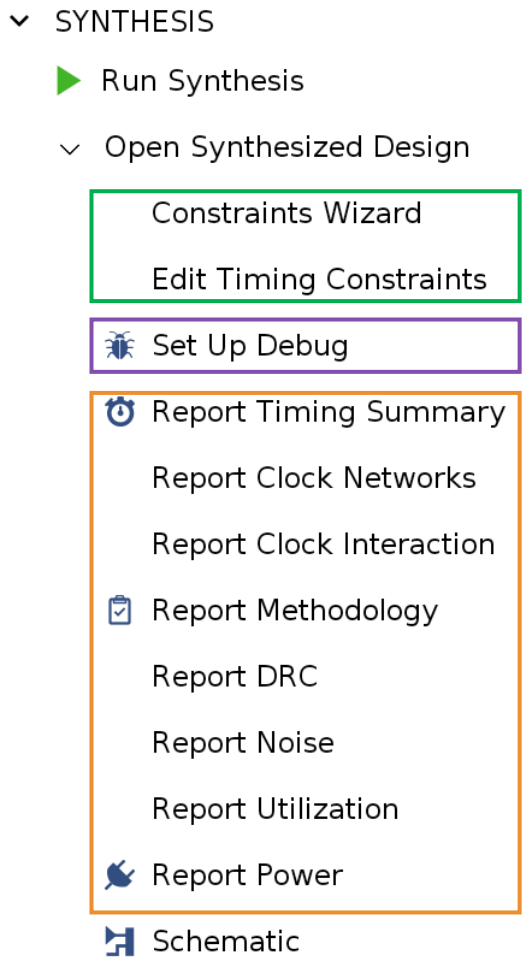
The Design View tools allow you to navigate between the logical and physical views of your design, making analysis, debugging, and implementation planning much easier

Design views

- The Device window provides a graphical view of the device, placed logic objects, and connectivity.
- Package window displays the physical characteristics of the target Xilinx part.
  - This window is used primarily during the I/O planning process or during port placement.
- The Schematic window allows selective expansion and exploration of the logical design.
  - You can generate a Schematic window for any level of the logical or physical hierarchy
    - select a logic element in an open window, such as a primitive or net in the Netlist window
    - use the Schematic command in the popup menu to create a Schematic window for the selected object.



Vivado Synthesis Results



Tasks on synthesized design:

- Update/add constraints
- Configure and implement debug cores for test and debug.
- **Reports** provide early feedback on area, timing, and power consumption.
  - Timing analysis is useful to ensure that paths have the necessary constraints for effective implementation. The synthesized design uses an estimate of routing delay to perform analysis (only timing analysis after implementation -place and route- includes the actual delays for routing).
  - Design Rule Checks (DRCs) check the design and report on common issues

Hands-on

Exercise 7

FIFO Project IP

Create a new project (FIFO_project_IP)

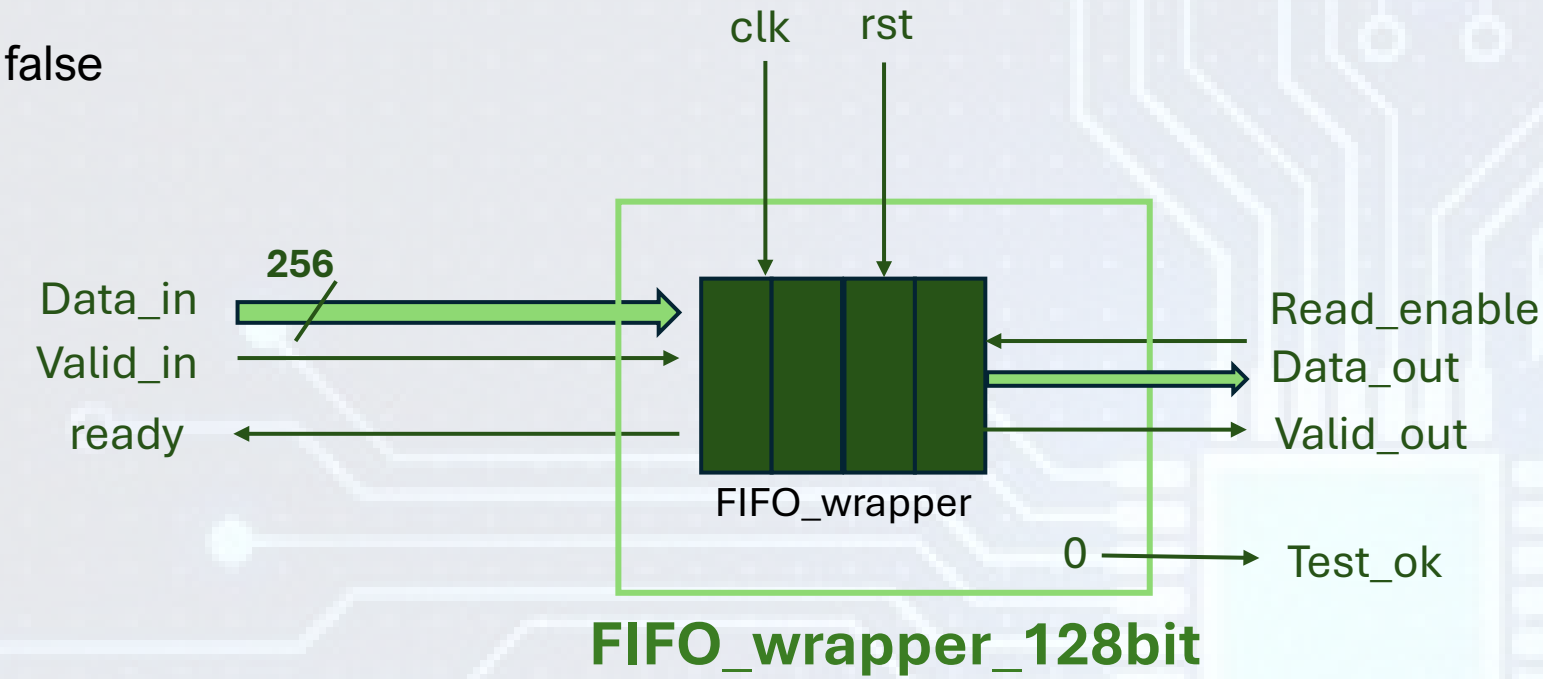
- Add fifo_wrapper_128bit (in /src)
- Add design sources (used in FIFO_project):
  - Fifo_wrapper
  - Data generator
  - Data checker
  - FIFO IPs
  - Data Reader
  - Output writer
  - File_comparator

} Simulation sources

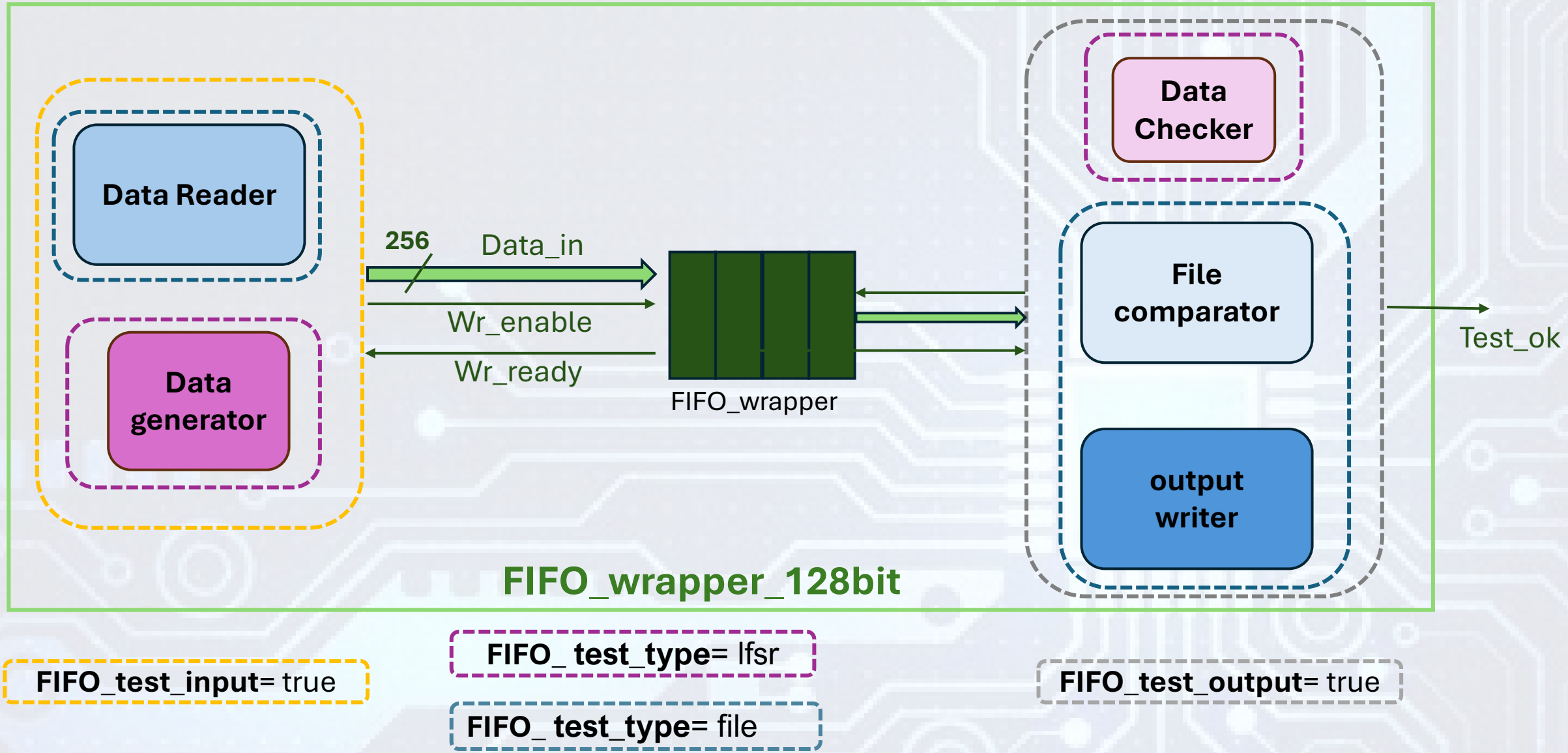
```
entity fifo_wrapper_128bit is
  generic (
    FIFO_MODE    : string := “AXI_STREAM";
    FIFO_TEST_INPUT  : boolean := true;
    FIFO_TEST_OUTPUT  : boolean := true;
    FIFO_TEST_MODE : string  := "lfsr" ;
    --lfsr test
    TESTED_DATA    : integer := 10;
    -- file
    file_data_integer     : string := "input.txt";
    file_data_out_integer : string := "output.txt"

  );
  port (
    clk          : in  std_logic;
    rst          : in  std_logic;
    data_in      : in  std_logic_vector(127 downto 0);
    valid_in     : in  std_logic;
    ready        : out std_logic;
    read_enable  : in  std_logic;
    data_out     : out std_logic_vector(127 downto 0);
    valid_out    : out std_logic;
    test_ok      : out std_logic
  );
end fifo_wrapper_128bit;
```

FIFO Project IP

- **FIFO_test** = false



FIFO Project IP



FIFO_test_input= true

FIFO_ test_type= lfsr

FIFO_ test_type= file

FIFO_test_output= true

FIFO Project IP

Exercise:

- Add FIFO_wrapper_top_synt (in /src) and use this entity only for synthesis
- Run simulation (using sim/TB_fifo_wrapper_128bit.vhd)
- Perform RTL linter and eventually solve reported issues
- Run synthesis
- Open the synthesized design

FIFO Project IP
RTL Linter

Linter

Violations (12) Waived (0) All (12)

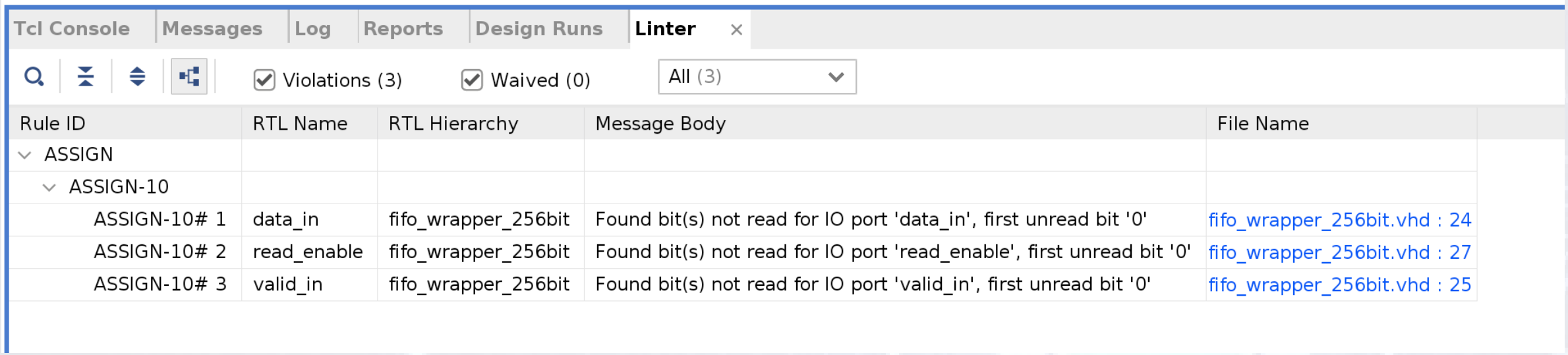
Rule ID	RTL Name	RTL Hierarchy	Message Body	File Name
ASSIGN				
ASSIGN-5				
ASSIGN-5# 1	compare_ok	fifo_wrapper_128bit	Found bit(s) not assigned for signal 'compare_ok', first unassigned bit '0'	fifo_wrapper_128bit.vhd : 53
ASSIGN-5# 2	data_received	fifo_wrapper_128bit	Found bit(s) not assigned for signal 'data_received', first unassigned bit '0'	fifo_wrapper_128bit.vhd : 52
ASSIGN-5# 3	eof	fifo_wrapper_128bit	Found bit(s) not assigned for signal 'eof', first unassigned bit '0'	fifo_wrapper_128bit.vhd : 51

WARNING ASSIGN-5: Found bit(s) not assigned for signal ‘signal’

- Some bits of a signal are never assigned a value.
- Check if the signal is really needed (if it’s not connected to anything, remove the assignment or the signal)
- If signal is used only in some conditional generate blocks, but not in all configurations, define it in generate block

```
gen_input_file : if FIFO_TEST_INPUT = true and FIFO_TEST_MODE="file"  generate
   signal eof     : std_logic;
begin
```

FIFO Project IP
RTL Linter



Rule ID	RTL Name	RTL Hierarchy	Message Body	File Name
ASSIGN				
ASSIGN-10				
ASSIGN-10# 1	data_in	fifo_wrapper_256bit	Found bit(s) not read for IO port 'data_in', first unread bit '0'	fifo_wrapper_256bit.vhd : 24
ASSIGN-10# 2	read_enable	fifo_wrapper_256bit	Found bit(s) not read for IO port 'read_enable', first unread bit '0'	fifo_wrapper_256bit.vhd : 27
ASSIGN-10# 3	valid_in	fifo_wrapper_256bit	Found bit(s) not read for IO port 'valid_in', first unread bit '0'	fifo_wrapper_256bit.vhd : 25

WARNING ASSIGN-10: Found bit(s) not read for IO port ‘port’

- One or more bits of an input/output port are never used
- Check whether the port is really needed
- If port is used only in some conditional generate blocks (and not in all configurations), the warning can be safely ignore.

FIFO Project IP
RTL Linter

WARNING INFER-1: inferred latch for signal ‘signal’

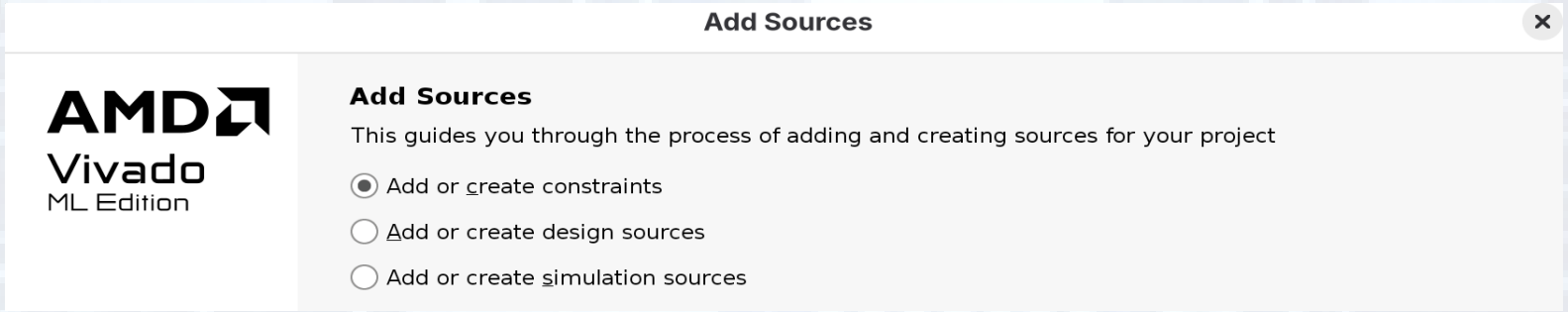
- combinational logic doesn’t assign a value to a signal in every possible condition, so it infers a latch to “remember” the previous value.

- Try to solve this issue…

FIFO Project IP
Synthesized design

- Create a Source constraint
- Add the required timing constraints (using Edit Timing constraints or Constrants Wizard under the Synthesized Design section)
- Rerun the synthesis

- Create a Source Constraint.

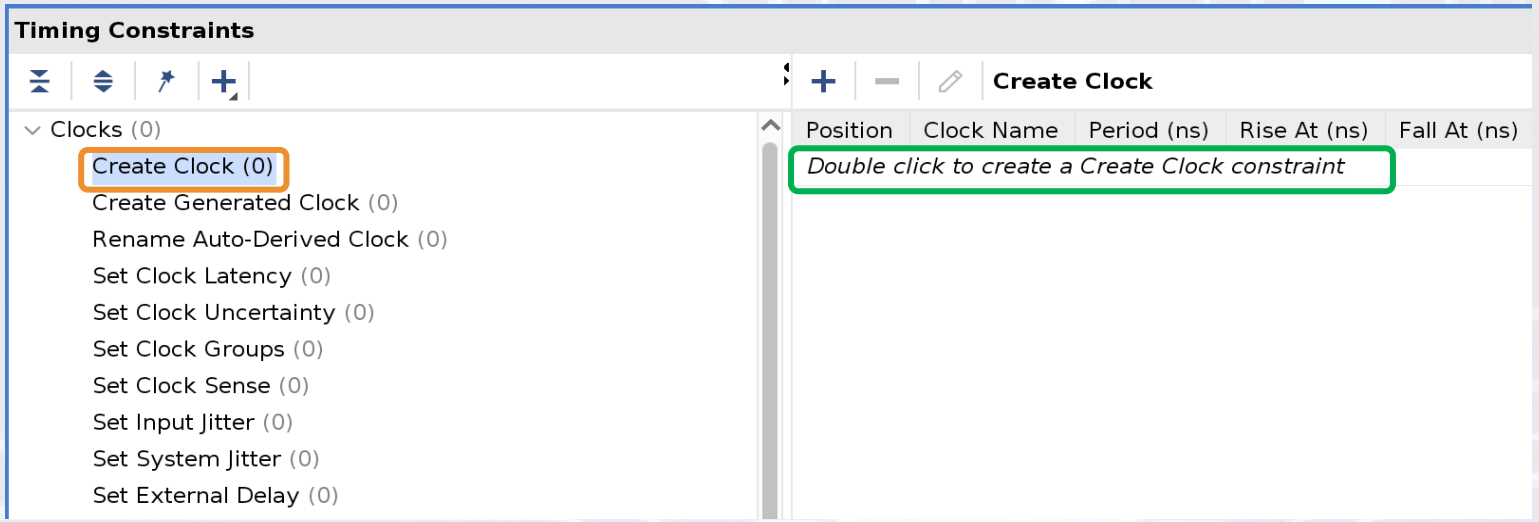


- In the Sources window right-click xdc file and select Set as Target Constraint File.

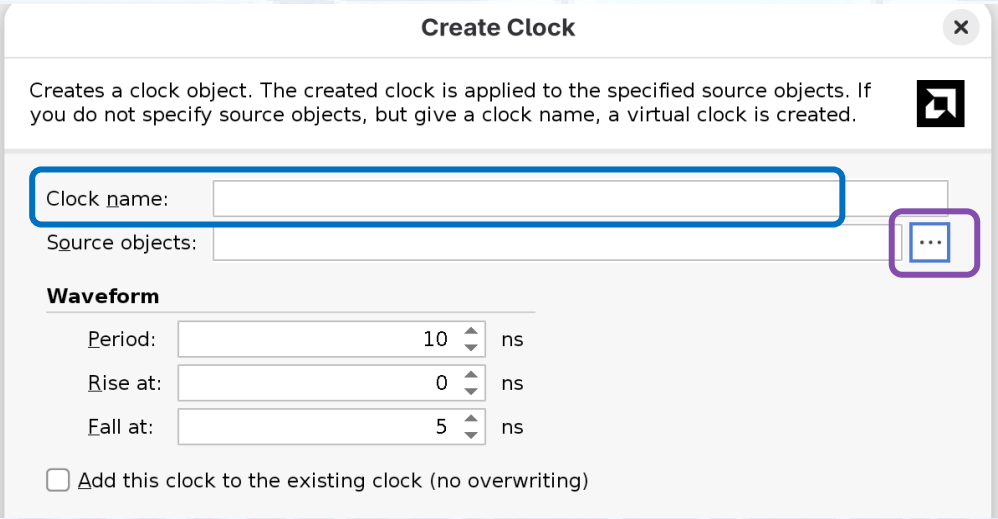
FIFO Project IP

Edit Timing Constraints

- Select Create Clock
- Double click to create a clock constraint



- Set Clock name (clk)
- Set Source object



FIFO Project IP

Edit Timing Constraints

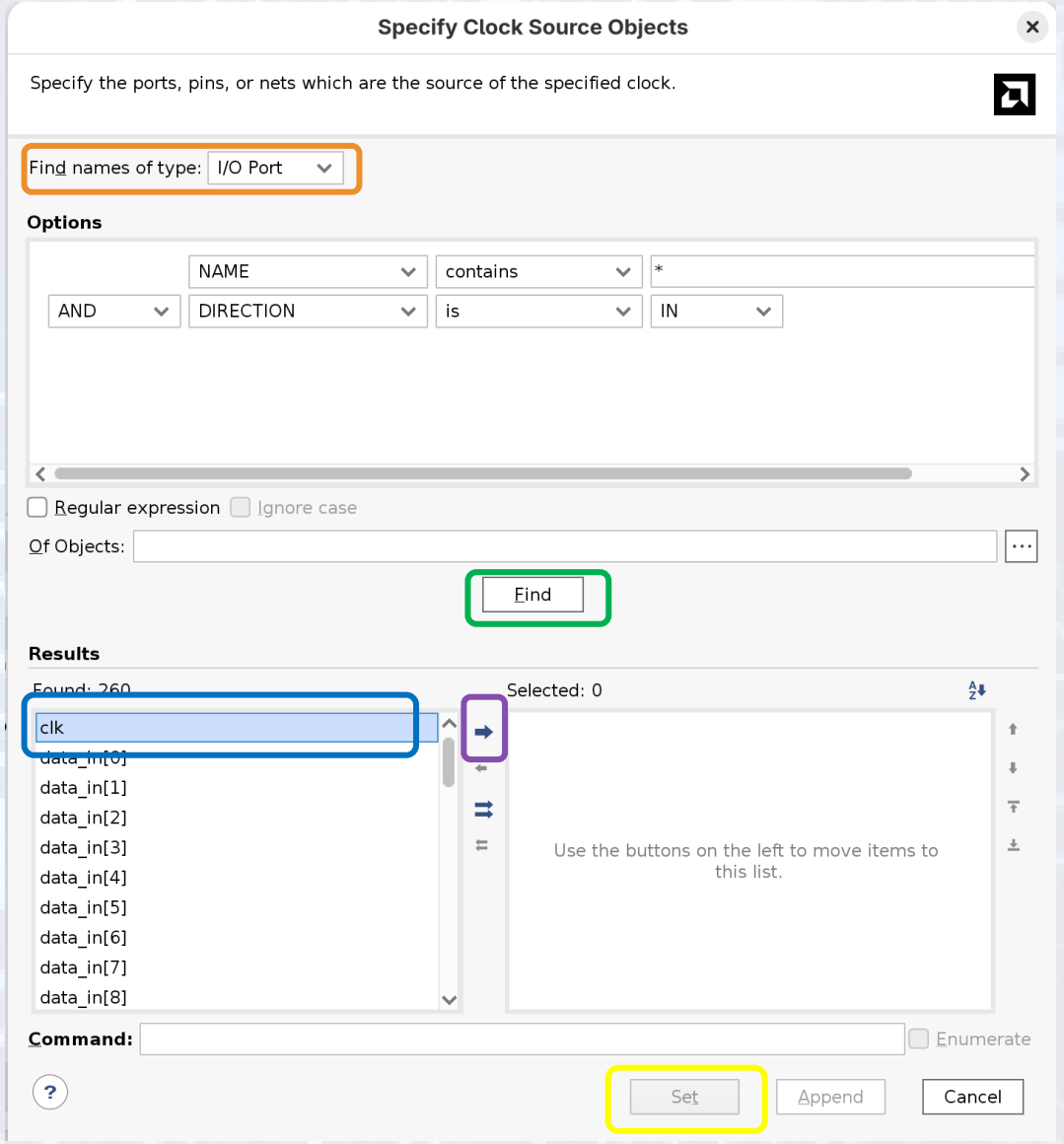
- From the **Find names of type** drop-down list, select I/O Port

- Click the **Find** button

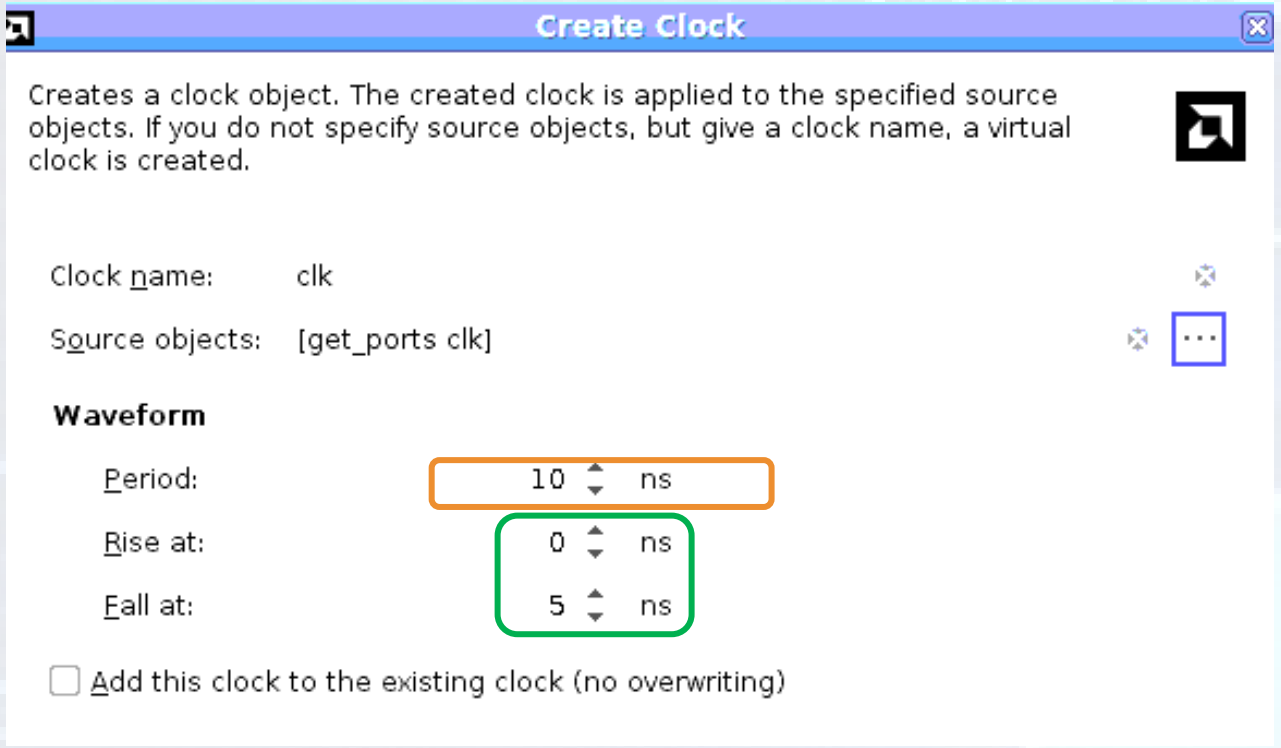
- Select clk in the found **results** text box

- press the right arrow to move clk to the **selected** names text box.

- press **Set**



FIFO Project IP
Edit Timing Constraints

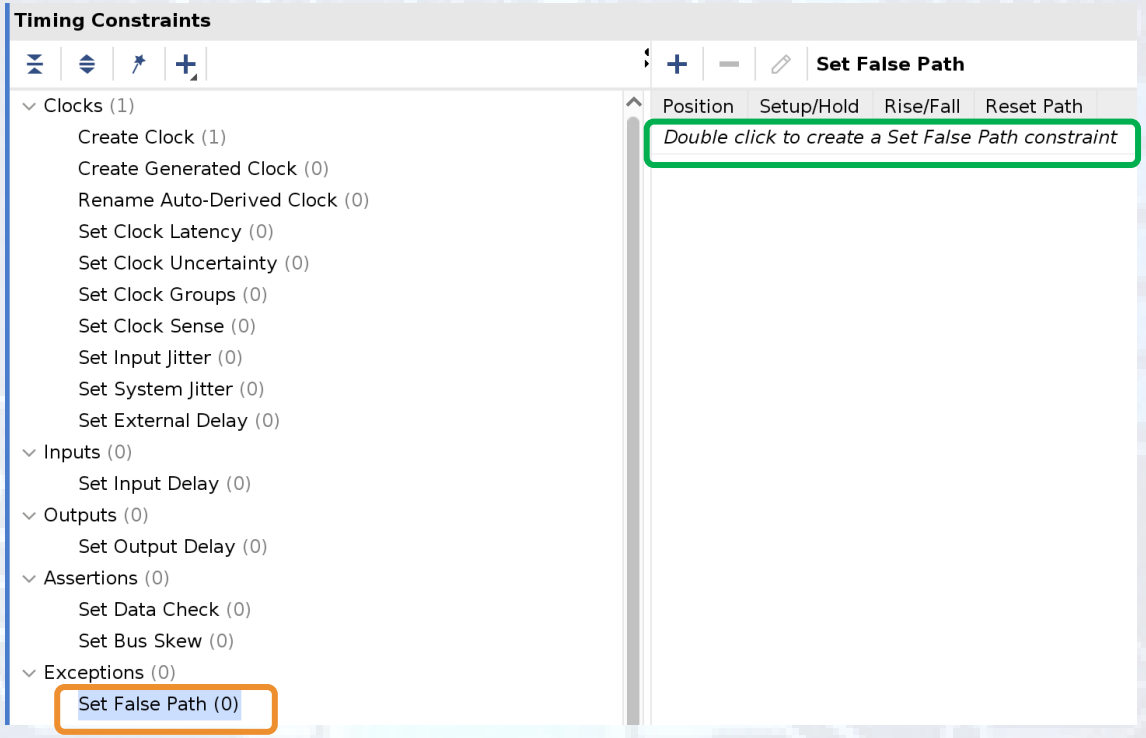


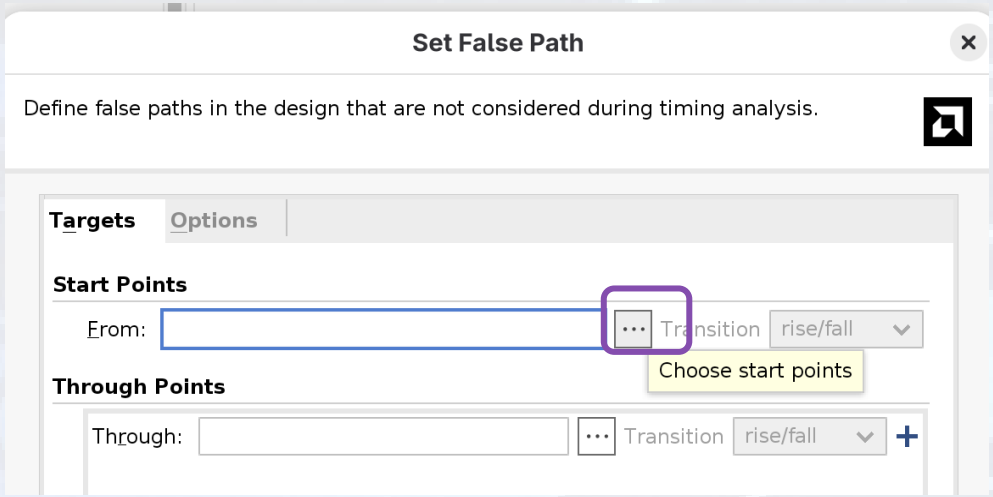
- Clock Period

- Specify Rise/Fall edge (define the duty cycle)

FIFO Project IP
Edit Timing Constraints

- Select Set False Path in Exception category
- Double click to create a false-path constraint



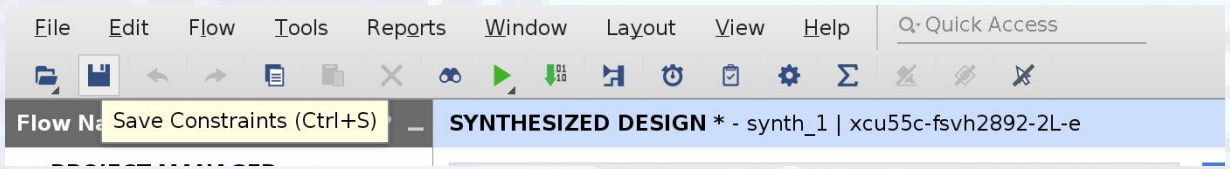


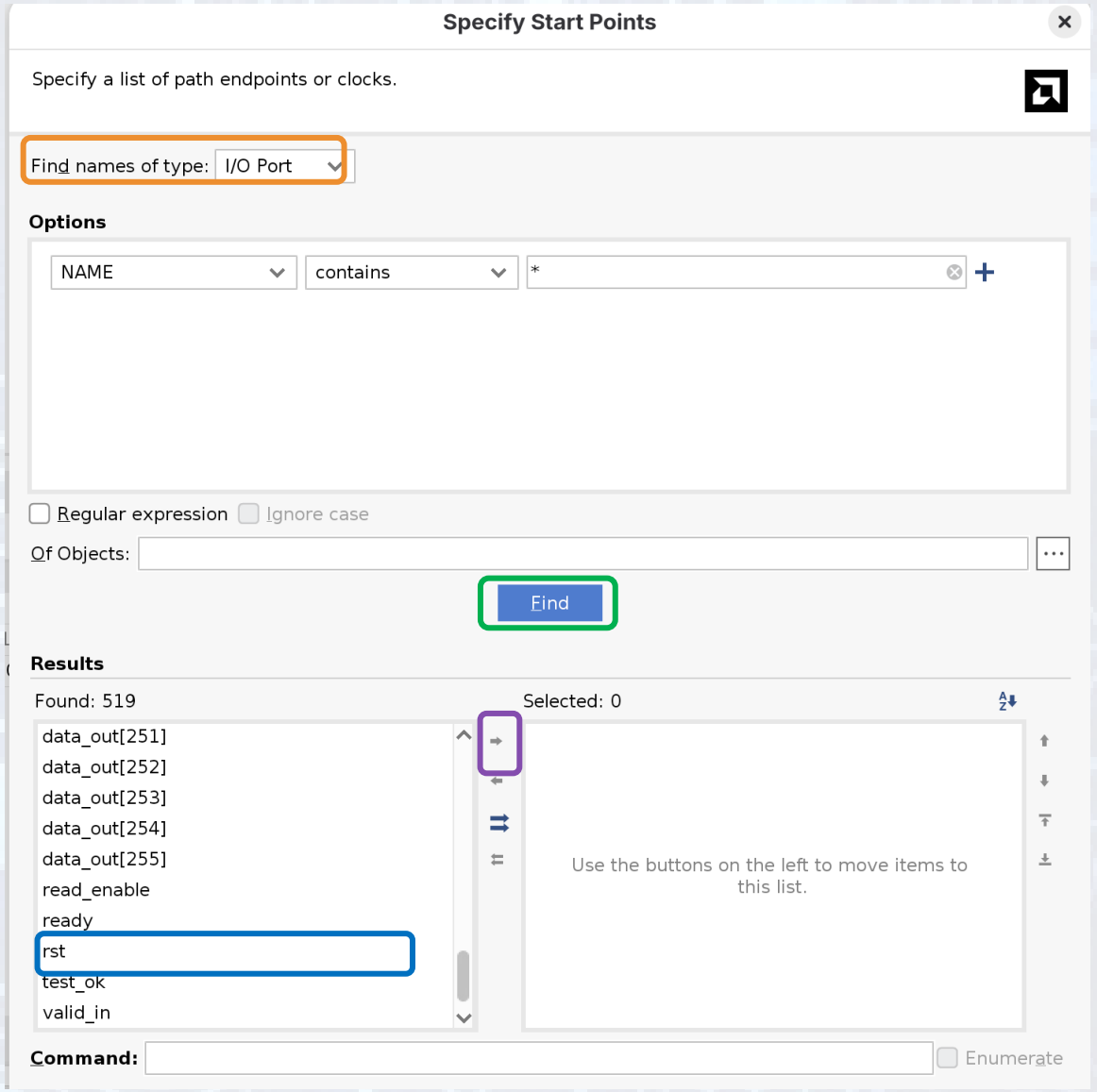
- Set Start Point

FIFO Project IP

Edit Timing Constraints

- From the **Find names of type** drop-down list, select I/O Port
- Click the **Find** button
- Select rst in the found **results** text box
- press the right arrow to move clk to the **selected** names text box.
- press **Set**
- Save constraints





FIFO Project IP
Synthesized design

- [Synth 8-7129] Port data_in[127] in module fifo_wrapper_128bit is either unconnected or has no load

- [Synth 8-7080] Parallel synthesis criteria is not met: appears when the tool decides that using multiple threads would not improve (or might even slow down) the compile time
    - For Small or Simple Design the overhead of parallelizing tasks outweighs the benefit

- [Synth 8-3332] Sequential element (gen_checker_lsfr.data_checker_inst/FSM_onehot_state_reg[4]) is unused and will be removed from module fifo_wrapper_128bit.
  Vivado usually encodes FSMs as one-hot, meaning that each state is represented by a separate flip-flop (one bit per state).
  If one of these bits is never asserted during synthesis analysis, then Vivado determines that:
  The state is never reached, or
  The state is reached but has no observable impact on outputs.
  As a result, it removes that flip-flop and the logic connected to it.

FIFO Project IP
Constraints wizard

Primary Clocks

Primary clocks usually enter the design though input ports. Specify the period and optionally a name and waveform (rising and falling edge times) to describe the duty cycle if not 50%. More info

Recommended Constraints

Object	Name	Frequency (MHz)	Period (ns)	Rise At (ns)	Fall At (ns)	Jitter (ns)
clk	clk	undefined	undefined			

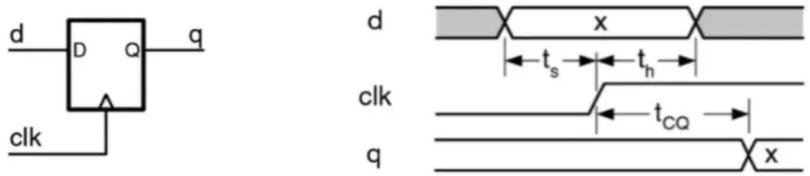
Fill in only the **frequency** (100 MHz) for the missing primary clock in the design

Tcl Command Preview (1) | Existing Create Clock Constraints (0)

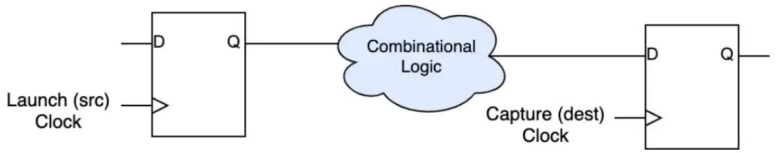
```
create_clock -period 10.000 -name clk -waveform {0.000 5.000} [get_ports {clk}]
```

Timing analysis

The data fed to a flip-flop must be stable before the clock edge and after the clock edge.



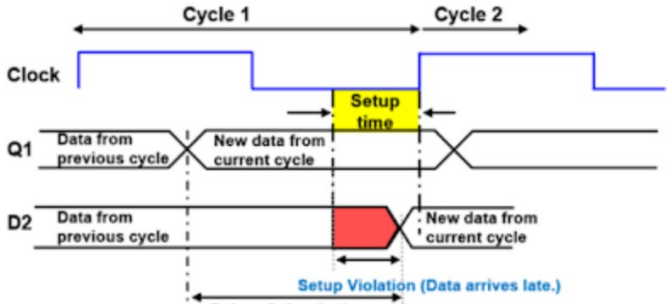
- The setup time is the amount of time required for the input to a flip-flop to be stable before a clock edge.
- The hold time is the minimum amount of time needed for the input to a flip-flop to be stable after a clock edge.

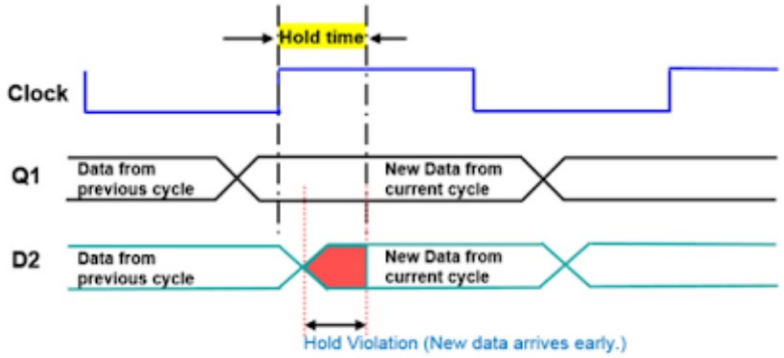


register-to-register timing path

The data:

- is launched inside the device by a sequential cell, which is clocked by the source clock.
- propagates through some internal logic before reaching a sequential cell clocked by the destination clock

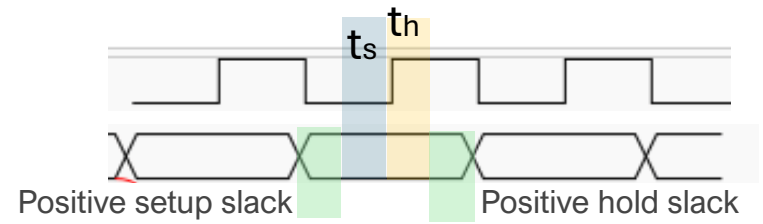


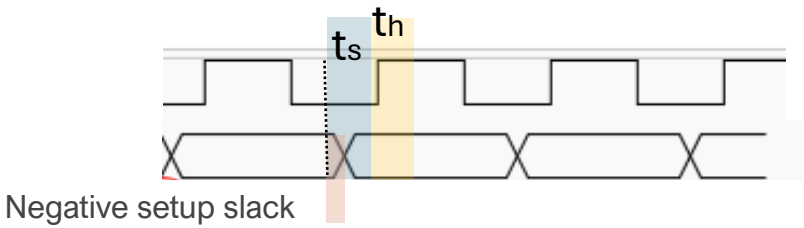


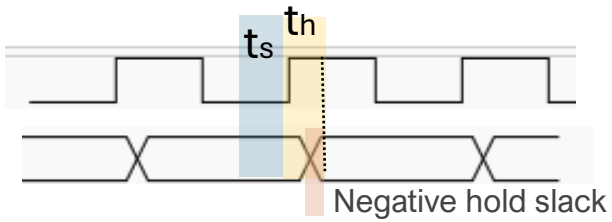
To reliably save the data into the flip-flop, the arrival time of the data required to meet the setup and hold time requirements

Timing analysis

Setup and **hold slack** is defined as the difference between the required time (based on setup and hold time) and the arrival time of the data at the endpoint.







Setup Slack Optimization Strategies

- Simplify or pipeline long combinational paths
- Break critical paths into shorter stages
- Constrain Clock Properly
- Analyze Multi-cycle & False Paths

Hold Slack Optimization Strategies

- Balance Data Paths
- Add intentional delay (LUTs or buffers) to short combinational paths

Hands-on

Exercise 8

FIFO Project IP

- Analyze Timing Summary

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	8.628 ns	Worst Hold Slack (WHS):	-0.066 ns	Worst Pulse Width Slack (WPWS):	4.458 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	-22.779 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	456	Number of Failing Endpoints:	0
Total Number of Endpoints:	793	Total Number of Endpoints:	793	Total Number of Endpoints:	318

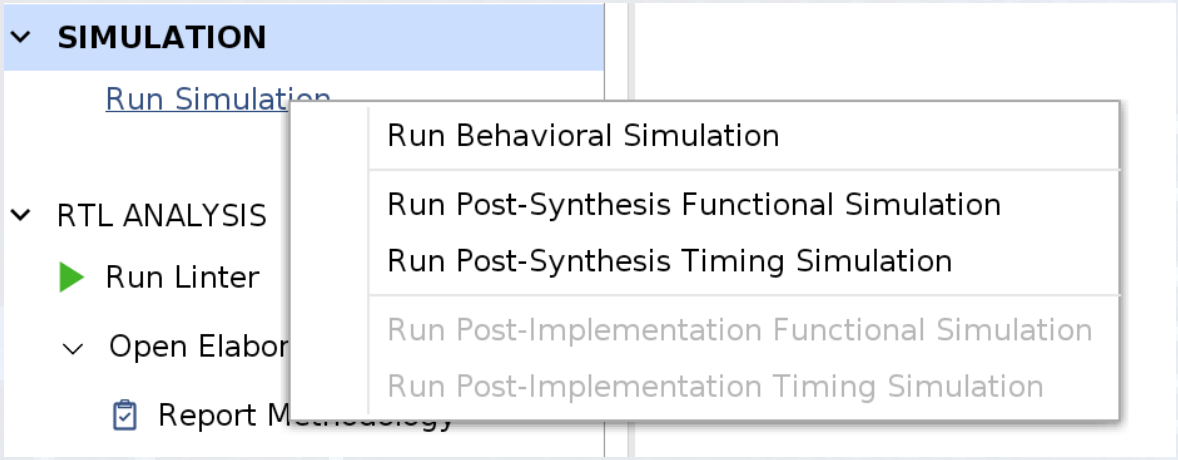
- Create a new TB (DUT: FIFO_wrapper_top_synt)
- Run post-synthesis simulations

```
ERROR: [VRFC 10-719] formal port/generic <tested_data> is not declared in <\FIFO_wrapper_top_synt\>
ERROR: [VRFC 10-9458] unit 'behavioral' is ignored due to previous errors [/home/flocicero/advanced_
```

FIFO Project IP

Vivado Post Synthesis Simulation verifies the functionality of the synthesized netlist.

- Check that synthesized RTL (with or without estimated delays) behaves as expected.
- Detect logical errors introduced by synthesis optimizations.
- Compare results with RTL simulation to ensure correctness.
- Not include FPGA routing or placement delays (timing is idealized).



In post-synthesis simulation:

- The design being simulated is no longer VHDL source code, but a netlist generated by synthesis (e.g., .dcp).
- All generic values are already resolved and hard-coded during synthesis (generic map cannot be used in the testbench)

- Delete the generics from the entity and re-run simulation

Vivado Implementation

Implementation is the process of mapping, placing, and routing the synthesized design onto the FPGA.

- Converts the synthesized netlist into a design that can physically run on the target device.
- Optimizes the logical design trying to ensure the design meets timing, area, and resource constraints.

    - **Physical constraints** define a relationship between logical design objects and device resources such as:
        - ❑ Package pin placement.
        - ❑ Absolute or relative placement of cells, including Block RAM, DSP, LUT, and flip-flops.
        - ❑ Floorplanning constraints that assign cells to general regions of a device.
        - ❑ Device configuration settings.

Compile time is impacted by :

- Netlist complexity and utilization
- Timing constraints and optimization

Physical constraints

PACKAGE_PIN defines a specific assignment, or placement, of a top-level port to a physical package pin on the device.

- XDC command

```
set_property PACKAGE_PIN <pin> [get_ports <port_name>]
```

- Port_name must exactly match the top-level HDL port names.
- Pin_name must correspond to valid pins for specific FPGA package.

```
set_property IOSTANDARD <standard> [get_ports <port>]
```

- IOSTANDARD: Specifies the electrical standard used by the I/O pin.
- <standard> is the name of the I/O standard (e.g., LVCMOS33, LVTTL, SSTL15, etc.).

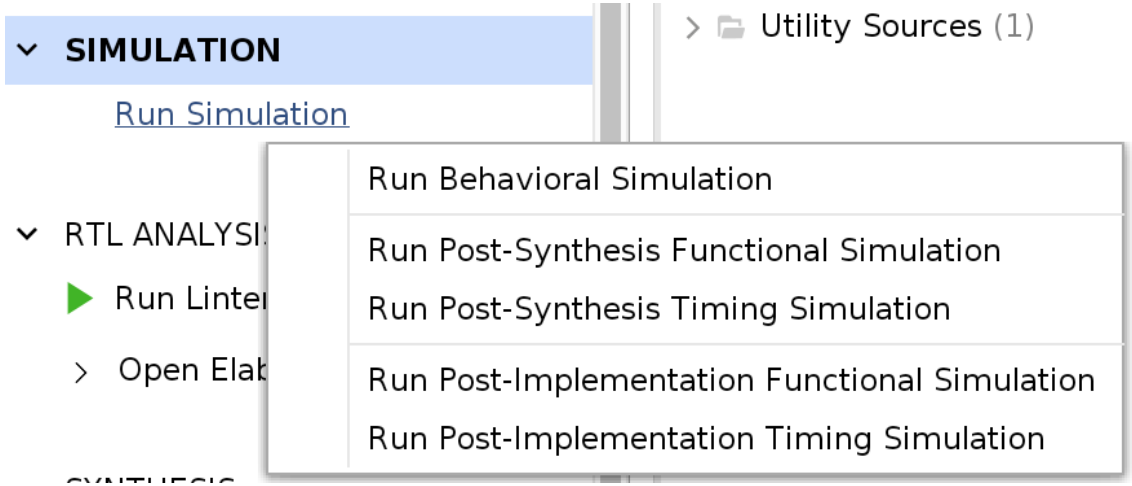
- Automatic Pins Assignment (when Synthesized Design is opened)
  - Tools → I/O Planning → Autoplace I/O Ports

- GUI Pins Assignment
  - I/O Port tab (in Implemented Design)

Vivado Post Implementation simulation

Post-Implementation Simulation (Post-SYNT)

- Verify the actual behavior of the implemented design.
- Detect timing violations (setup, hold, propagation delays).
- Compare results with post-synthesis simulation to ensure correctness.



- Includes all routing and placement delays.
- Crucial for timing-critical designs.
- Helps catch errors not visible in RTL simulation.

Vivado: Bitstream Generation and Flash Programming

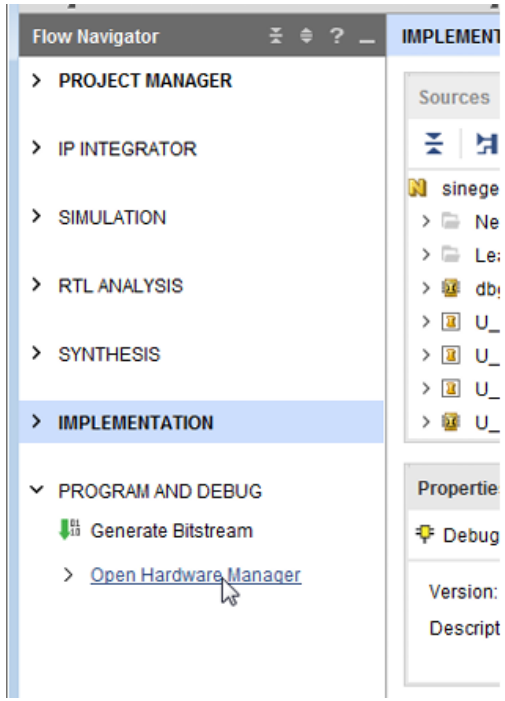
Bitstream Generation converts the implemented design into a binary file (bitstream).

- Bitstream contains all configuration data to program the FPGA.
- Bitstream can be directly loaded to FPGA via JTAG or stored in non-volatile Flash memory.
- Flash memory allows FPGA to boot with the design automatically at power-up.

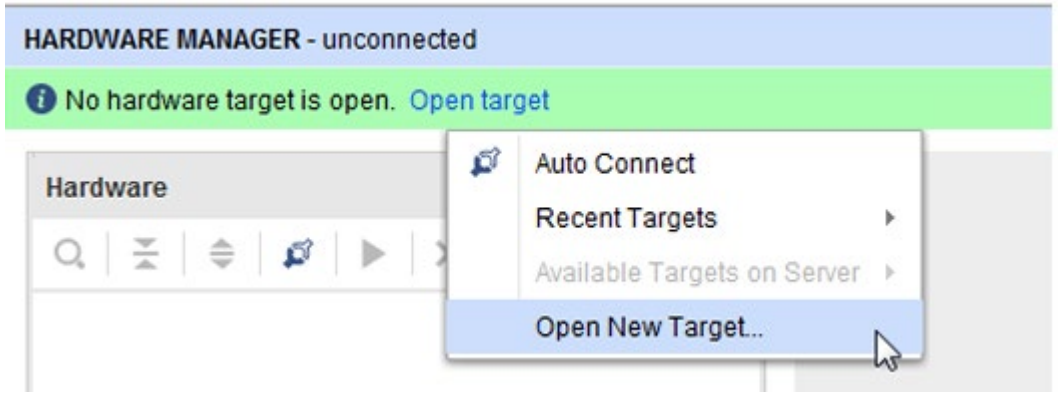
- Bitstream is device-specific.

Vivado Hardware manager

Vivado Hardware Manager allows you to connect to physical FPGA devices, program the bitstream, and perform real-time debugging.



- Click Open Target → Open New Target.
- Wait for the connection to the hardware to complete



- In the Hardware Server Settings page, type the name of the server (or select Local server if the target is on the local machine) in the Connect to field.
- Program the device using the previously created .bit bitstream by right clicking the device and selecting Program Device.

Hands-on

Exercise 9

FIFO Project IP

- Add the required physical constraints (in const_pin.xdc)
- Run implementation
- Open the implemented design
- Generate Bitstream

Synthesis Report Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	8.563 ns	Worst Hold Slack (WHS):	-0.066 ns	Worst Pulse Width Slack (WPWS):	4.458 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	-71.354 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	1364	Number of Failing Endpoints:	0
Total Number of Endpoints:	2493	Total Number of Endpoints:	2493	Total Number of Endpoints:	1086

Timing constraints are not met.

Vivado automatically fixes hold time violations during implementation, inserting delay buffers on short paths

Implementation Report Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	6.598 ns	Worst Hold Slack (WHS):	0.012 ns	Worst Pulse Width Slack (WPWS):	4.458 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	2480	Total Number of Endpoints:	2480	Total Number of Endpoints:	1079

All user specified timing constraints are met.

Vivado: tcl

TCL (Tool Command Language) is a scripting language used to automate tasks in Vivado.

- Supports design creation, synthesis, implementation, simulation, and bitstream generation.
    - Automate repetitive tasks and workflows.
    - Ensure design reproducibility.

- Vivado provides a Tcl interface that can be accessed through:
    - Integrated Tcl console in the GUI
    - Interactive Tcl mode (*vivado –mode tcl*)
    - batch mode for running scripts automatically (*vivado –mode batch*)

- The **source** command is used to execute a tcl script in Vivado

By default, source command prints in the Tcl console all Tcl statements that are executed as comments (a comment always starts with the # character).
Command echoing can be suppressed by adding the -notrace option as follows:

source -notrace run.tcl

Vivado: tcl

Common tcl uses in Vivado

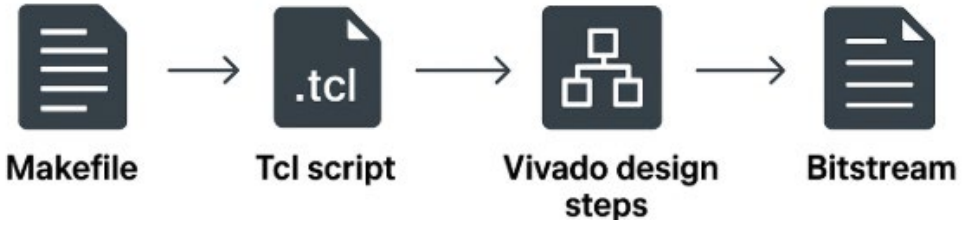
- Creating and managing projects:
  create_project
- Adding design and constraint files:
  add_files
- Generating IP:
  create_ip
- Running synthesis and implementation:
  launch_runs synth_1 wait_on_run synth_1 launch_runs impl_1
- Running simulation:
  launch_simulation

Note: Every GUI operation is mirrored in Tcl console and Vivado keeps track of all these commands in the so journal file (.jou).

Tip: You can easily reproduce the entire flow at any time by executing the journal file with the -source option when invoking vivado at the command line (save the original journal file as vivado.tcl before, otherwise the file will be overwritten)

Vivado: Makefile

Makefiles can be used to automate Vivado builds by calling Tcl scripts.



Note: According to Makefile syntax, instructions inside each target MUST BE IDENTED USING A TAB CHARACTER !

Benefits:

- Automates repetitive tasks
- Reproducible builds

Hands-on

Exercise 10

FIFO Project IP

Exercise:
In the /tcl dir

- Open create_project.tcl and gen_wrapper.tcl
- Configure the directory paths (e.g., for sources, constraints, outputs) according to your environment.
- Type `make` to see the list of available targets
- Generate and simulate AXI_STREAM mode with FIFO_TEST=true

Improvements:

- Generate and silmulate all configurations.

Improvements:

- Write a Tcl script to automate the generation of bitstream.
- Add the Tcl command to the Makefile

Course Overview

Introduction to FPGA

- Architecture
- Advantages and limitations
- Design Flow
- Simulation tool

Tools & Programming languages

- Advanced VHDL
- Vivado tool overview
  - Design Flow
    - Project creation
    - Ip core integration
    - Simulation
    - Synthesis
    - Implementation
    - Tcl scripting
  - Timing analysis
  - Custom IP Design & Integration
  - FPGA Test & Debug

FPGA interconnection

- Quick intro
- AURORA Xilinx
  - Test and Debug
  - Fine tuning
  - Timing and Resources analysis
  - Optimization

Vivado: User-defined IP

A **User-Defined IP** is a custom hardware module (written in VHDL, Verilog, or HLS) that is packaged into an IP format

Before packaging RTL as an IP, it is recommended to:

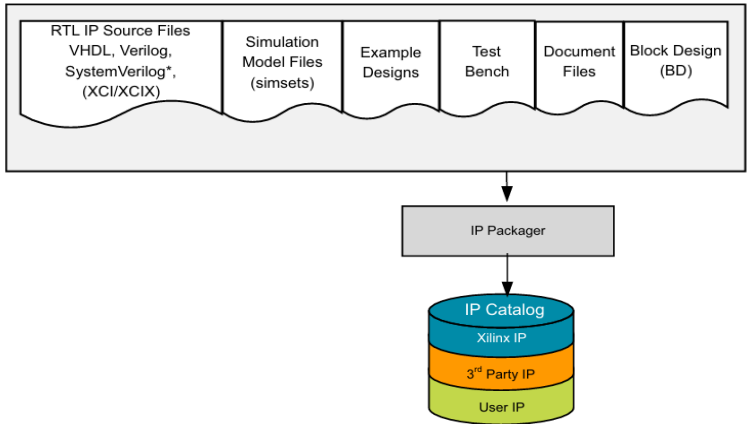
- Verify simulation results
- Validate sources through synthesis and implementation

Vivado IP packager allows

- Create and package files and associated data in an IP-XACT standard format.
- Add IP to the Vivado IP catalog.
- Deliver packaged IP to an end-user in a repository directory or in an archive (.zip) file.

Benefits

- Reusability across multiple projects
- Integration into block designs
- Parameterization and GUI support



Hands-on

Exercise 11

FIFO Project IP

- Open project FIFO_project_axi_stream_test
- Disable FIFO_wrapper_top_synt
- Generate project IP

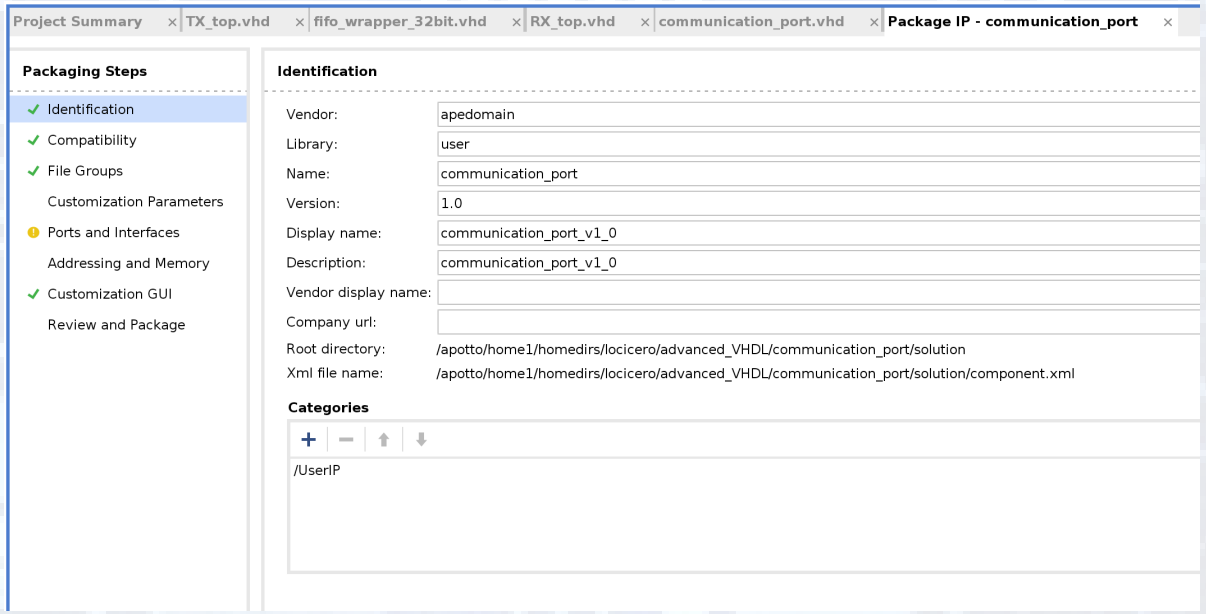
Improvements:

- Write a Tcl script to automate the creation and configuration of the IP.
- Add the Tcl command to the Makefile to easily launch IP generation from the command line.

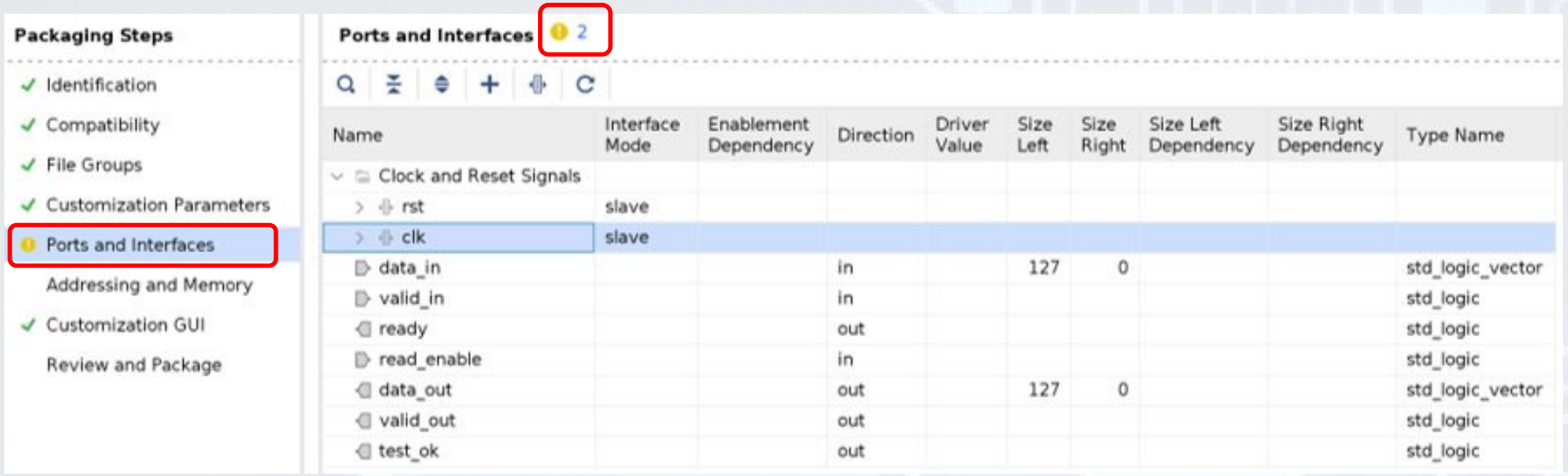
FIFO Project IP

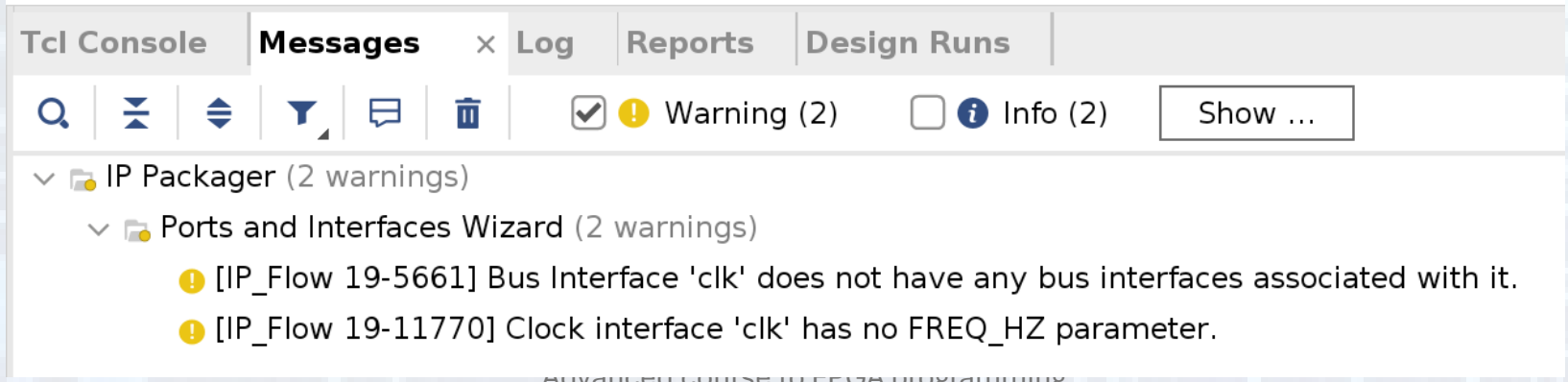
To Create a Custom IP

- Prepare RTL project
- Tools → Create and Package New IP
- Fill out metadata (Name, version, vendor, library ..)
- Define interfaces and ports (AXI, clock, reset, etc.)
- Customize IP behavior (parameters, GUI options)
- Package IP



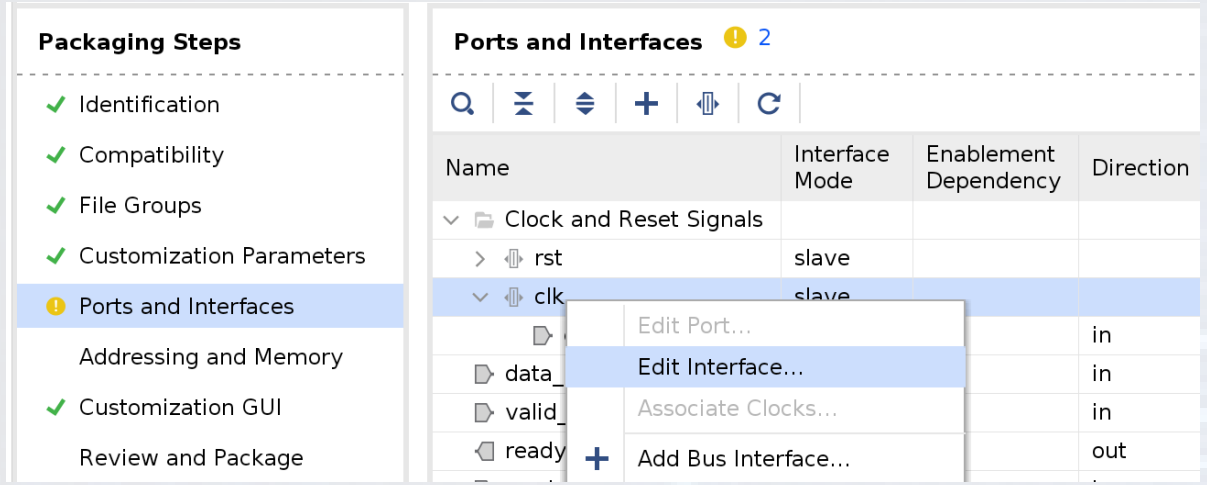
FIFO Project IP

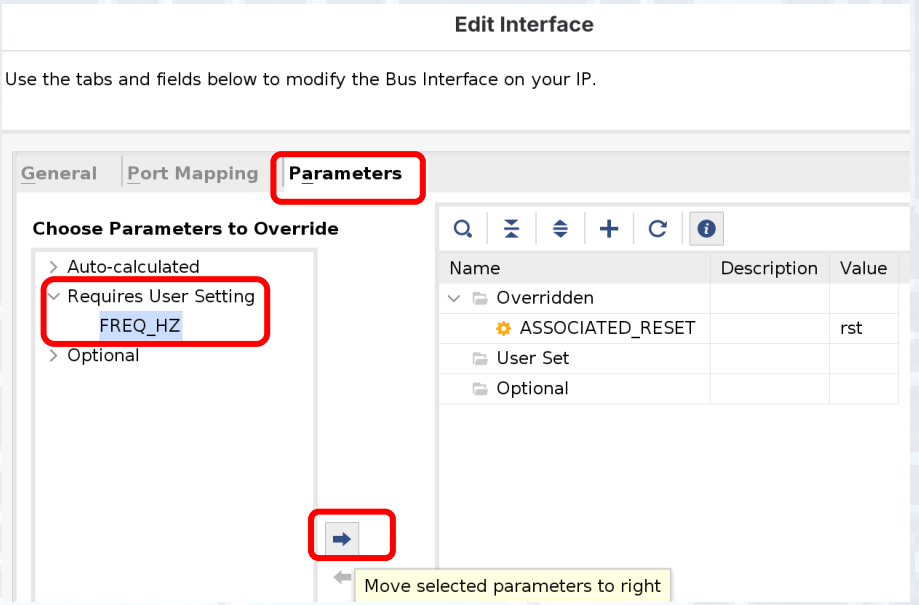


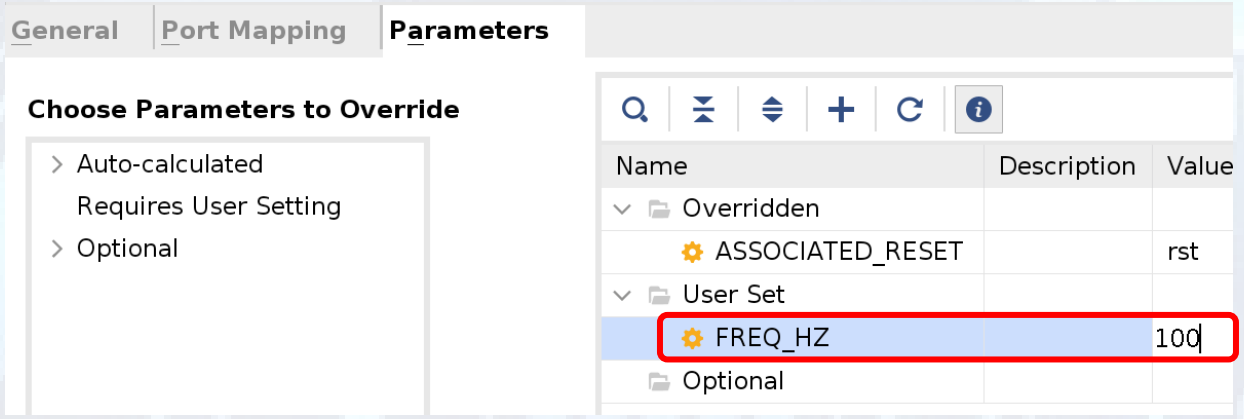


FIFO Project IP

WARNING: The clock interface clk does not specify a frequency value (FREQ_HZ).
The frequency parameter is used by the tools for simulation and timing analysis.



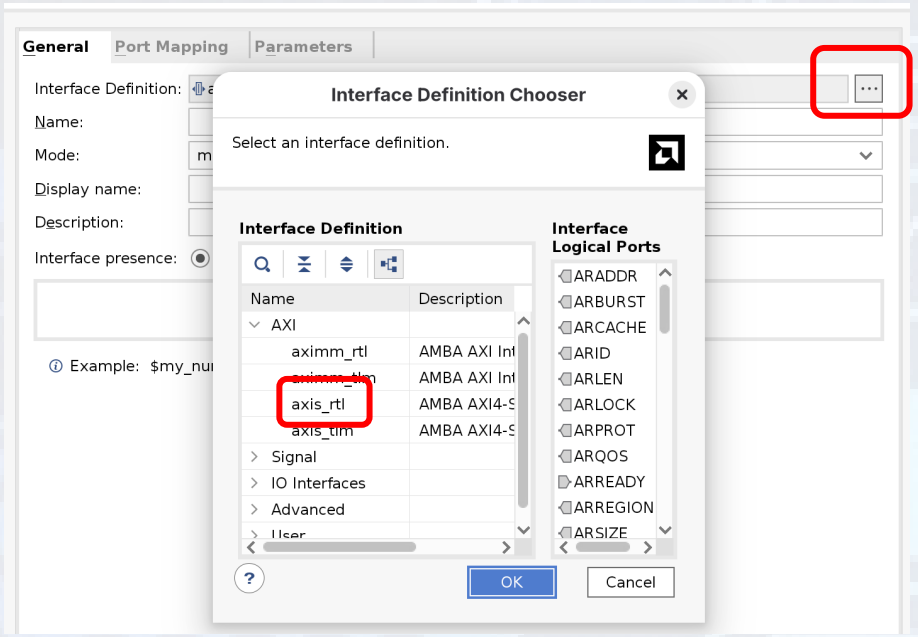


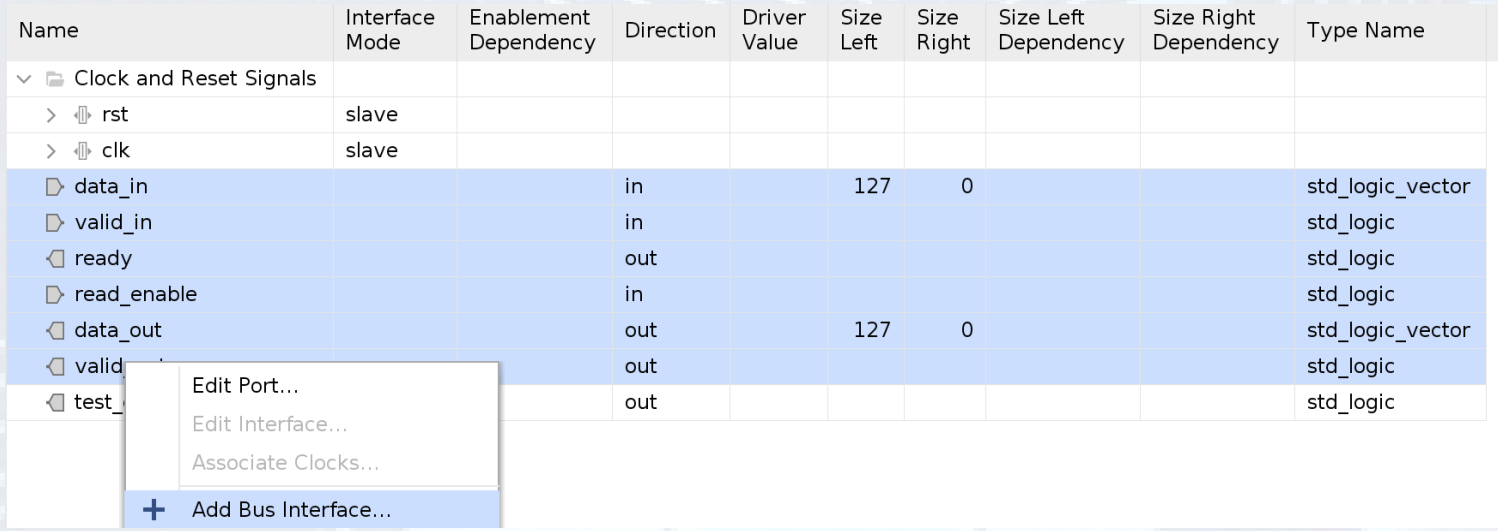


Set 100000000

FIFO Project IP

WARNING: Bus Interface 'clk' does not have any bus interfaces associated with it.
The tool expects some bus interfaces to be connected to clk but finds none.
If the IP do not use any bus interface (e.g., AXI, APB), the warning is harmless

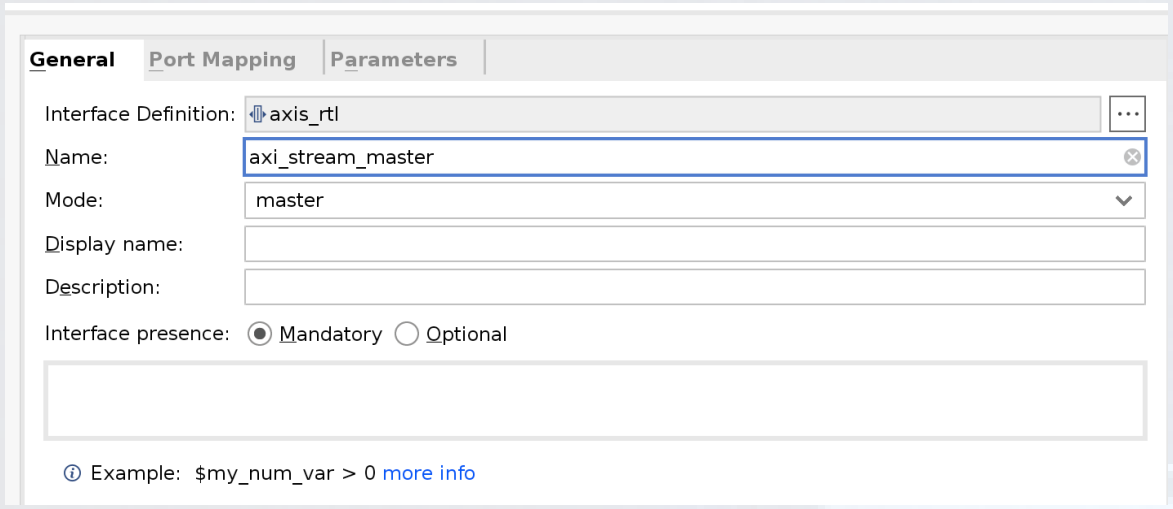


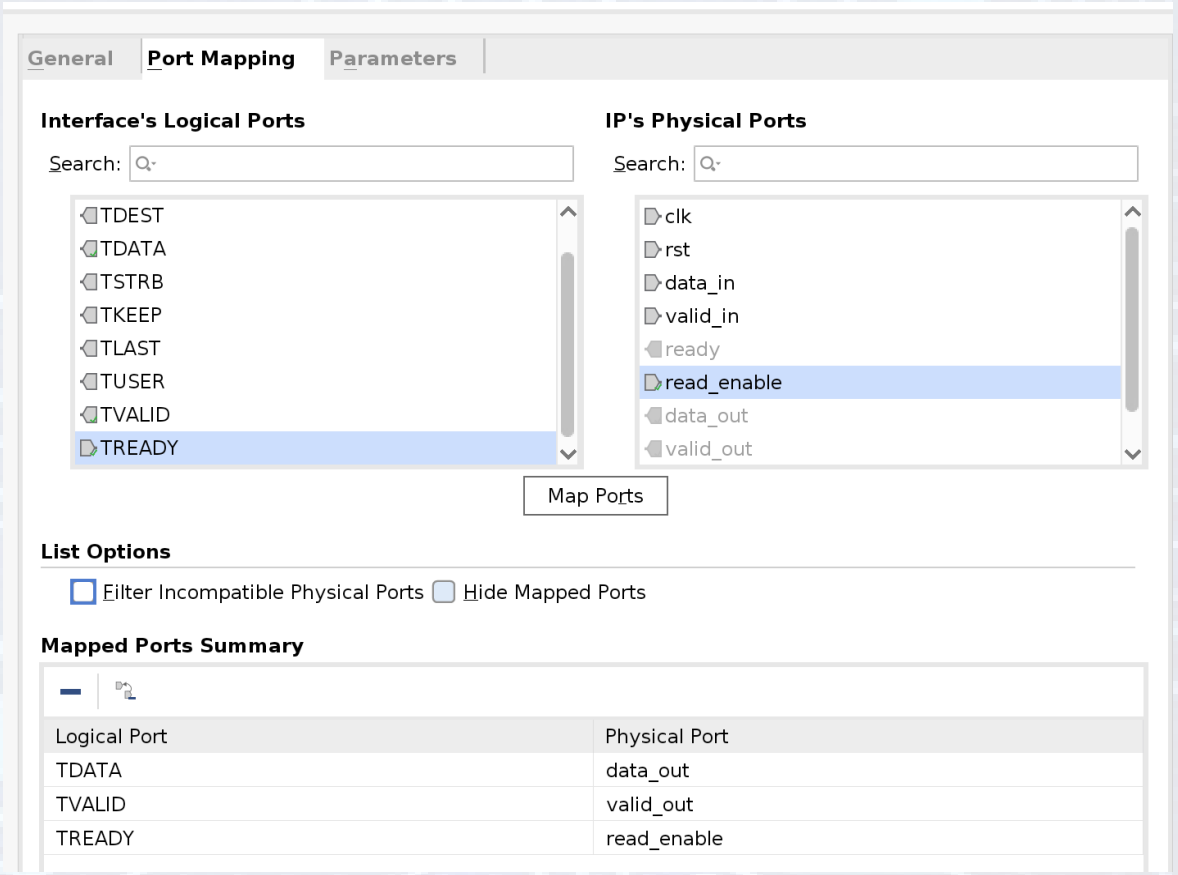


- Set Interface definition axis_rtl

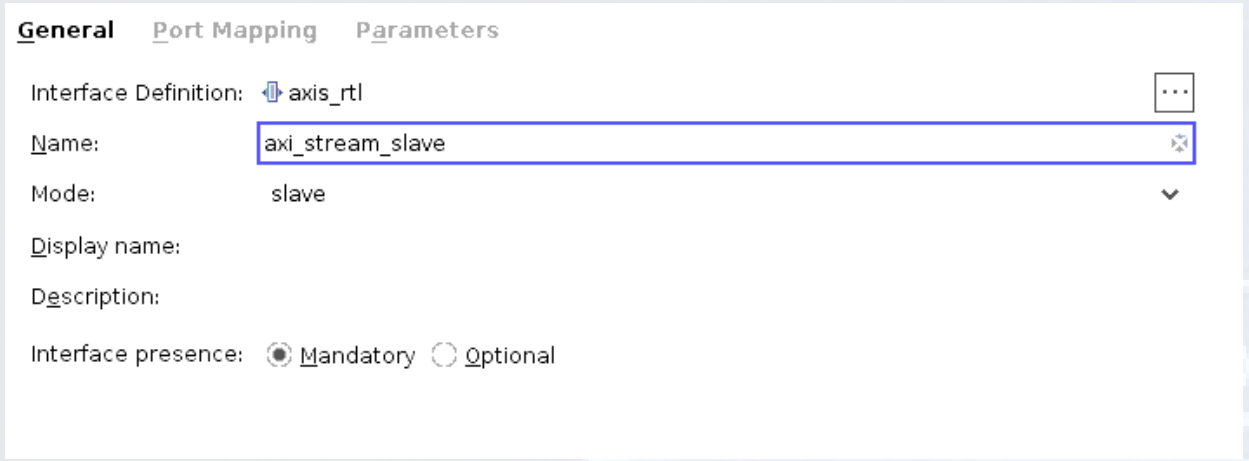
FIFO Project IP

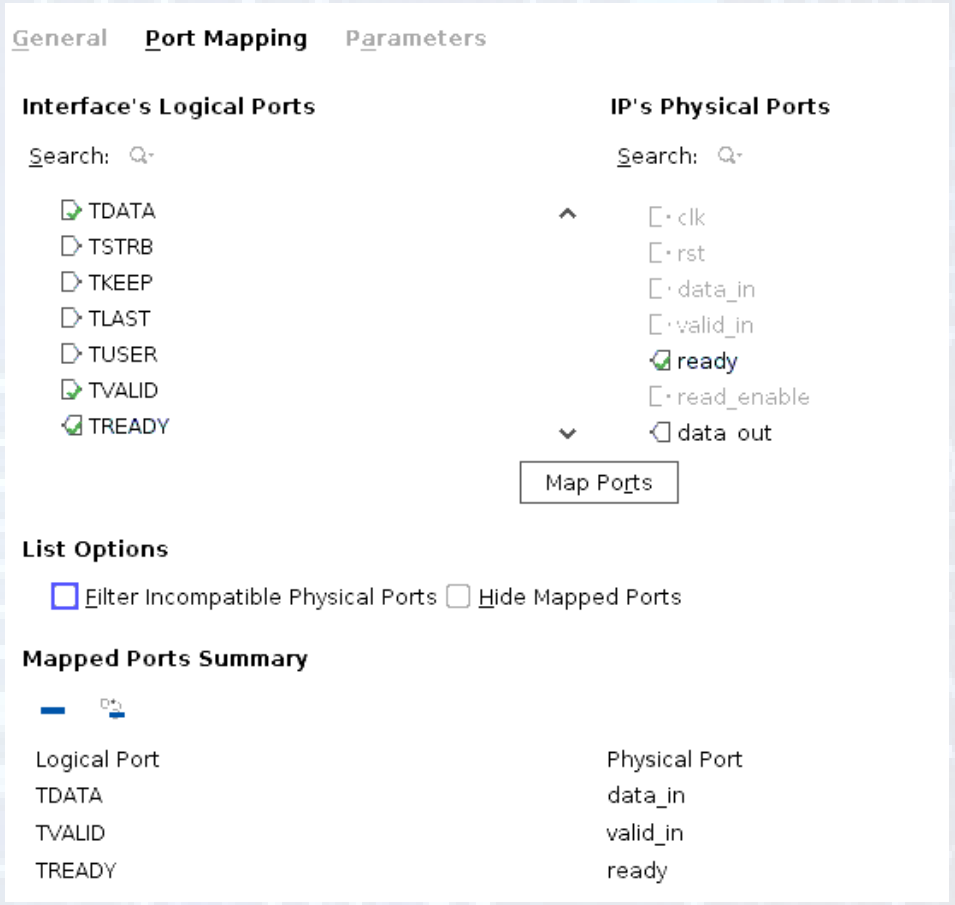
AXI stream master IF



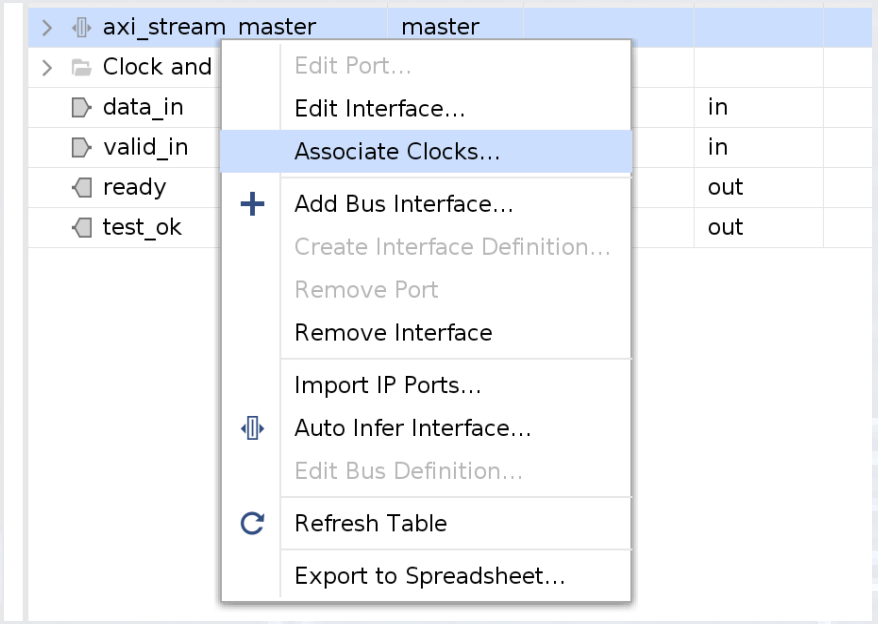


FIFO Project IP
AXI stream slave IF

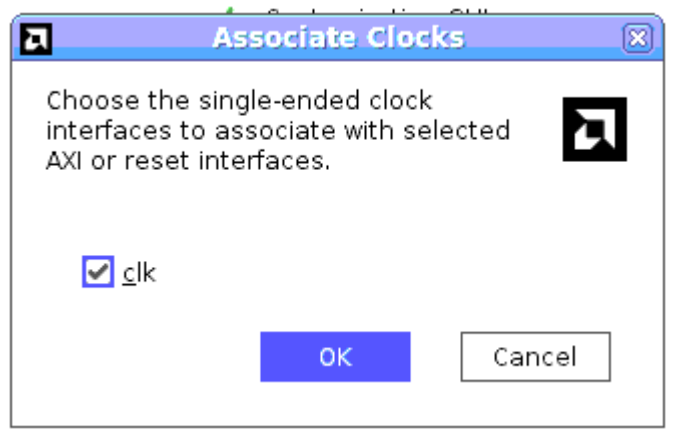




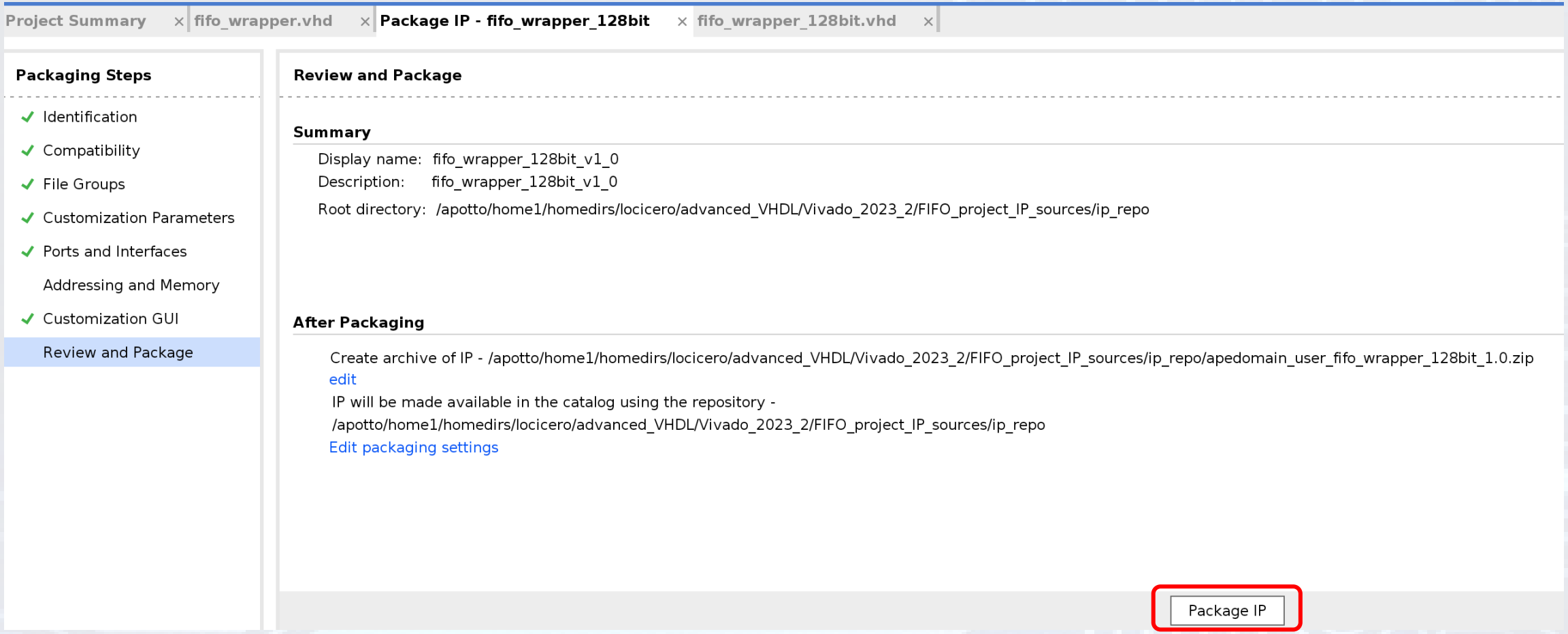
FIFO Project IP
AXI stream IFs



- Connect clock clk to both AXI stream interfaces

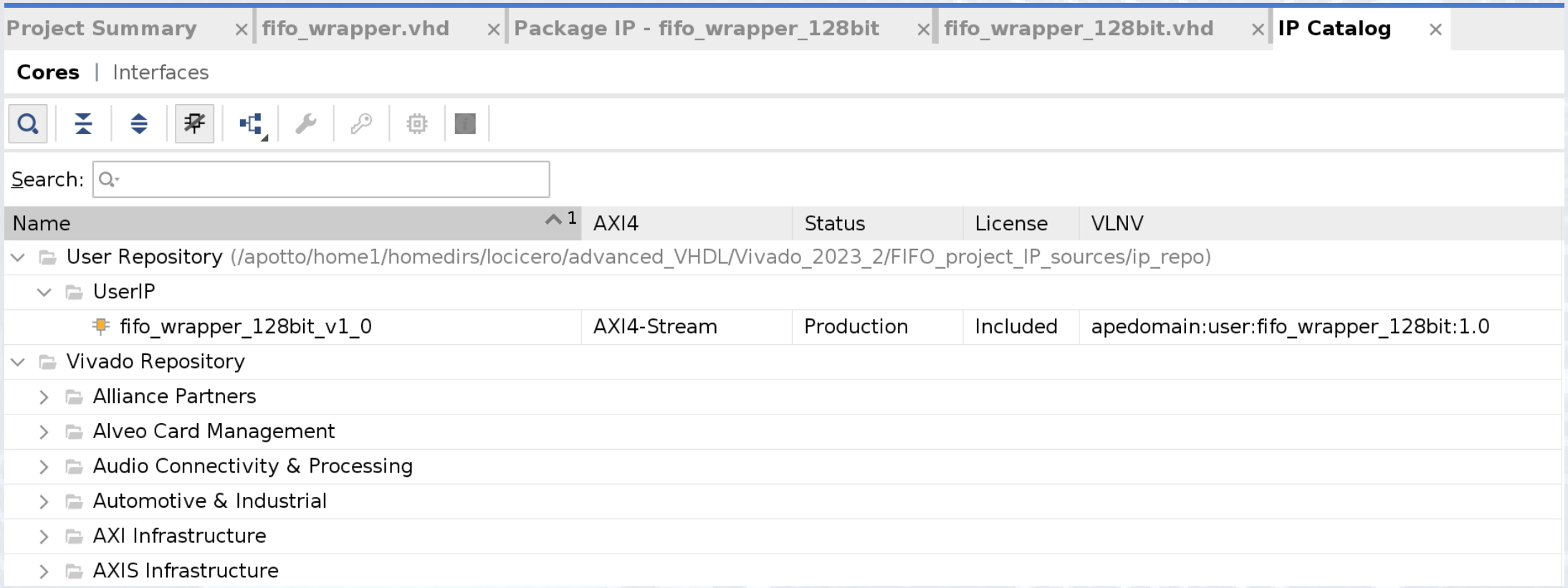


FIFO Project IP



FIFO Project IP

IP appears in the IP Catalog, ready for reuse



Course Overview

Introduction to FPGA

- Architecture
- Advantages and limitations
- Design Flow
- Simulation tool

Tools & Programming languages

- Advanced VHDL
- Vivado tool overview
  - Design Flow
    - Project creation
    - Ip core integration
    - Simulation
    - Synthesis
    - Implementation
    - Tcl scripting
  - Timing analysis
  - Custom IP Design & Integration
  - FPGA Test & Debug

FPGA interconnection

- Quick intro
- AURORA Xilinx
  - Test and Debug
  - Fine tuning
  - Timing and Resources analysis
  - Optimization

Vivado: Test and Debug

After generating the bitstream, it is possible to test and debug the design directly on the FPGA using Vivado's built-in tools:

- **Integrated Logic Analyzer (ILA) core** : perform in-system debugging of post-implemented
  - To monitor signals in the design
  - To trigger on hardware events and capture data at system speeds.

- **Virtual I/O cores (VIO)** can both monitor and drive internal FPGA.
- T his debug core needs to be instantiated in the RTL code

Integrated Bit Error Ratio Tester (IBERT) enables in-system serial I/O validation and debug.

- To measure and optimize high-speed serial I/O links in FPGA-based system. AMD recommends using the IBERT Serial Analyzer design when you are interested in addressing a range of in-system debug and validation problems from simple clocking and connectivity issues to complex margin analysis and channel optimization issues.

Vivado: ILA

The ILA core can be inserted post synthesis in the Vivado design flow or instantiated in RTL code.

Post-synthesys design flow

- **Add debug nets**
  - ❑ Mark signals for debug in the source HDL as well as the post synthesis netlist.

```
attribute mark_debug : string;
attribute mark_debug of sine     : signal is "true";
```

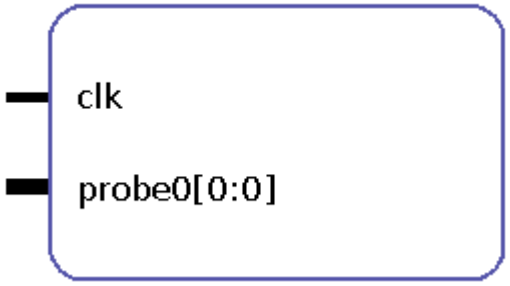
This method gives the highest probability of preserving HDL signal names after synthesis.
This can prevent optimization that might otherwise occur to that signal.

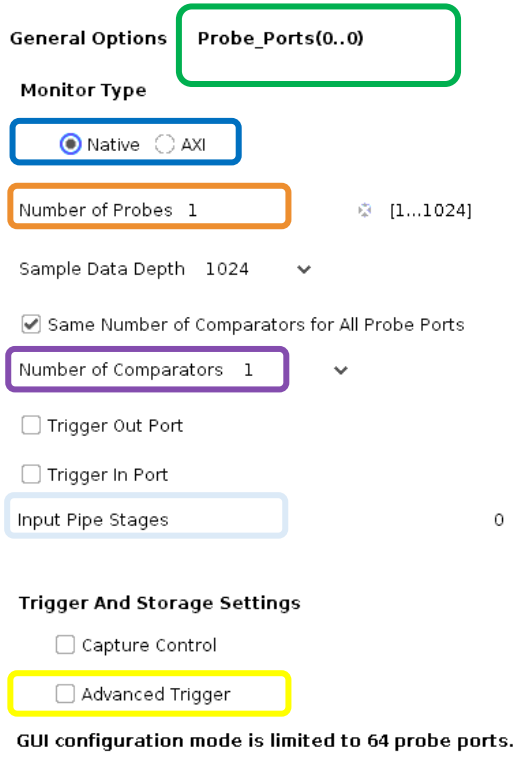
  - ❑ Right-click and select **Mark Debug** or **Unmark Debug** on a synthesized netlist.
    - This method is flexible
    - Does not require HDL source modification
    - Synthesis might not preserve the signals due to netlist optimization involving absorption or merging of design structures.
- **Run the Set Up Debug wizard**.

Vivado: ILA

RTL instantiation

- **Generate an ILA IP**





Type of interface ILA should be debugging

Number of probes port (in Native type)

Each probe input is connected to trigger comparators

Number of registers addeed to the probes

Advance trigger option enables trigger state machine to write trigger sequence in Vivado Logic Analyzer

- **Add IP to the design**

Probe Port Panels to configure width of each Probe

Hands-on

Exercise 12

FIFO Project IP

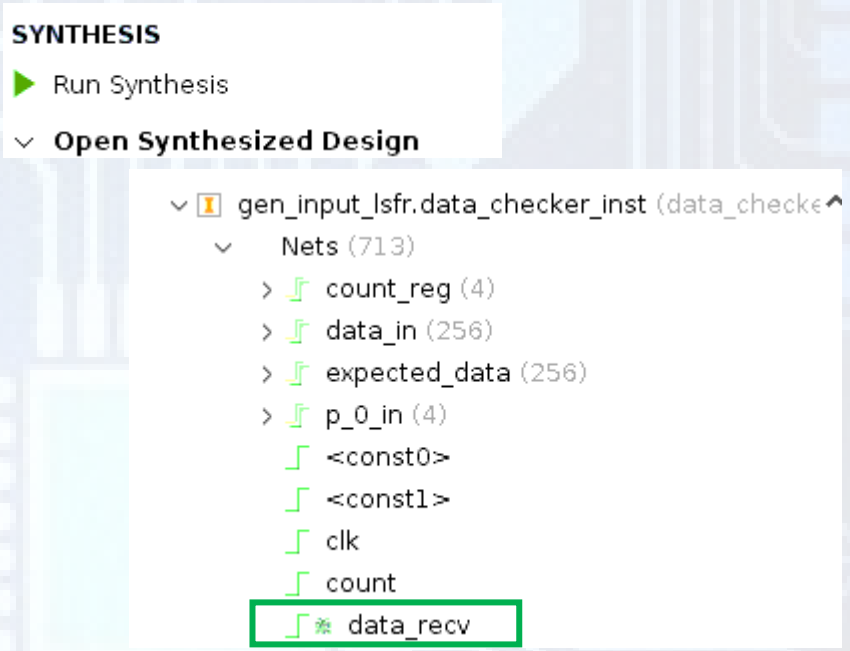
Exercise:

- Add debug net on data_recv (with mark_debug attribute in data_checker) and on test_ok (using GUI)
- Run SetUP Debug wizard to add the corresponding ILA

FIFO Project IP

Mark signals for debug in the source HDL

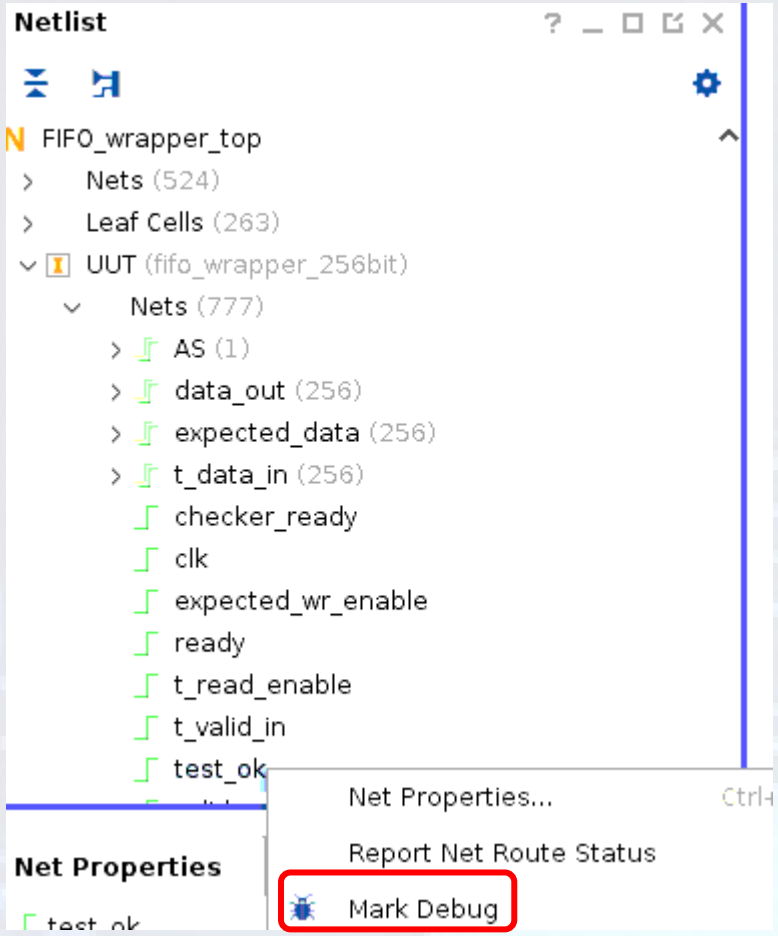
```
entity data_checker is
  …
end entity;

architecture FSM of data_checker is
….
  signal count      : integer range 0 to NUM_DATA := 0;
.
   -- Mark this signal for debug
    attribute mark_debug : string;
    attribute mark_debug of data_recv : signal is "TRUE";

begin
```



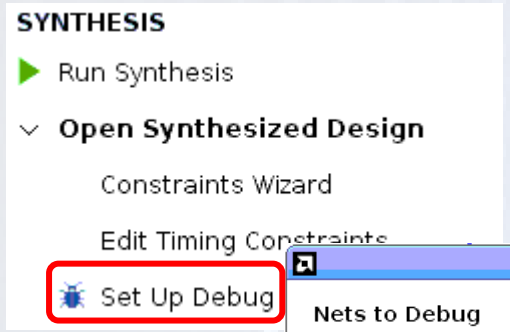
Nets corresponding to signals marked for debug in HDL are automatically listed in the Debug window under the Unassigned Debug Nets folder.

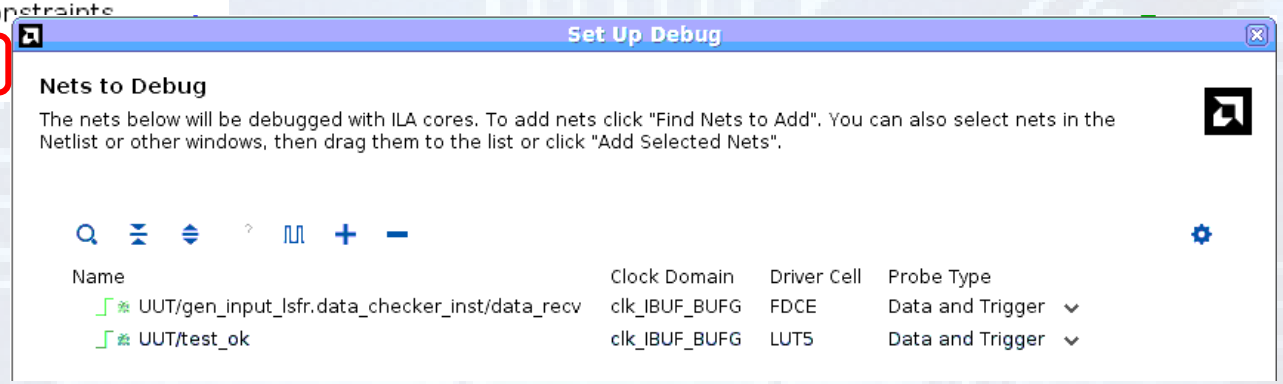
FIFO Project IP

- Right-click on test_ok net and select **Mark Debug**

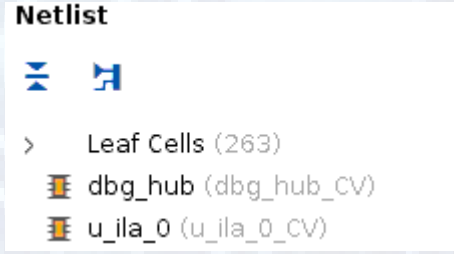


- Run **Set Up Debug Wizard**

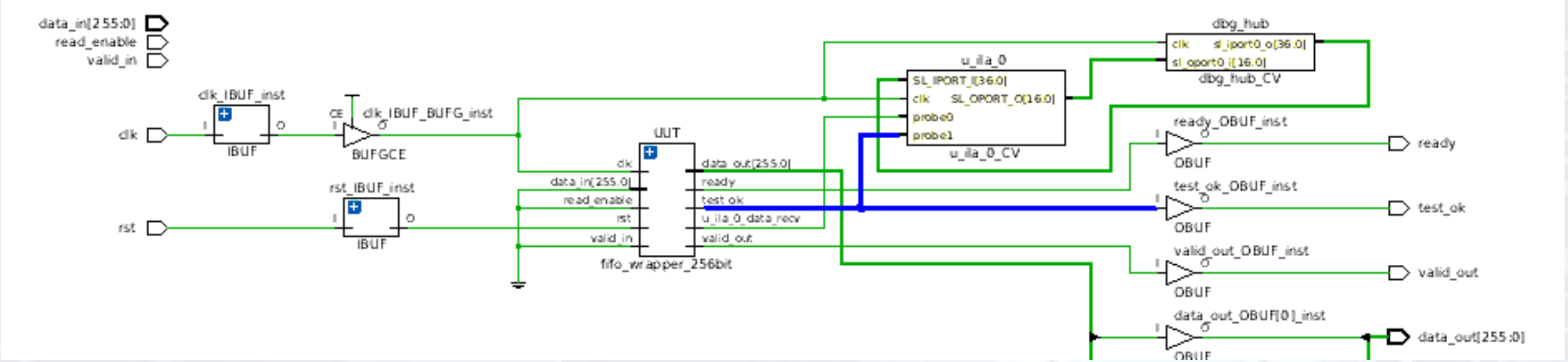




The relevant XDC commands that insert the ILA core(s) is generated.
Remember to save project!



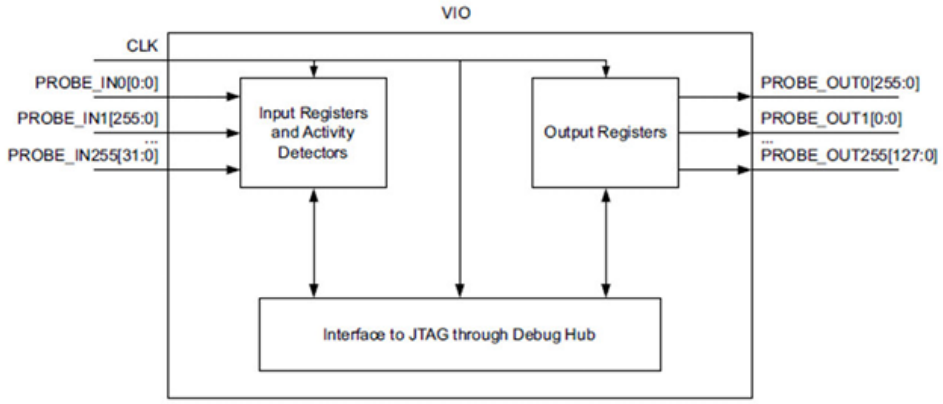
FIFO Project IP
SetUp Debug Wizard

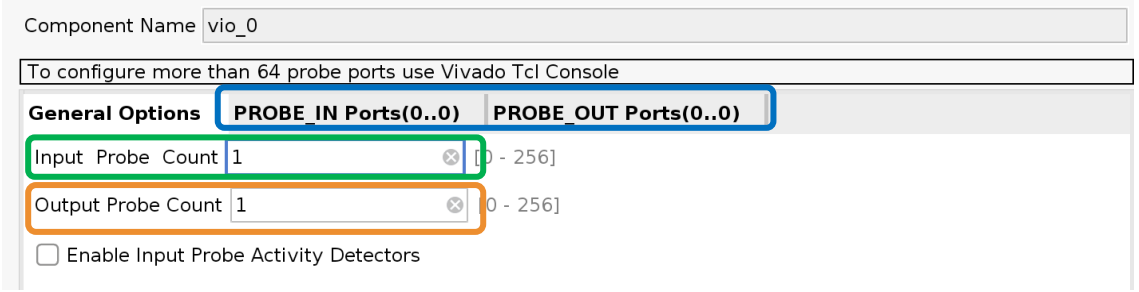


Design Schematic

Vivado: VIO

The Virtual Input/Output (VIO) core is a customizable core that can both monitor and drive internal FPGA signals in real time.





- **Generate a VIO IP**

Number of input probe ports

Number of output probe ports

Probe_in/out Ports Panels
Use the Probe Width field to set the width of each probe port

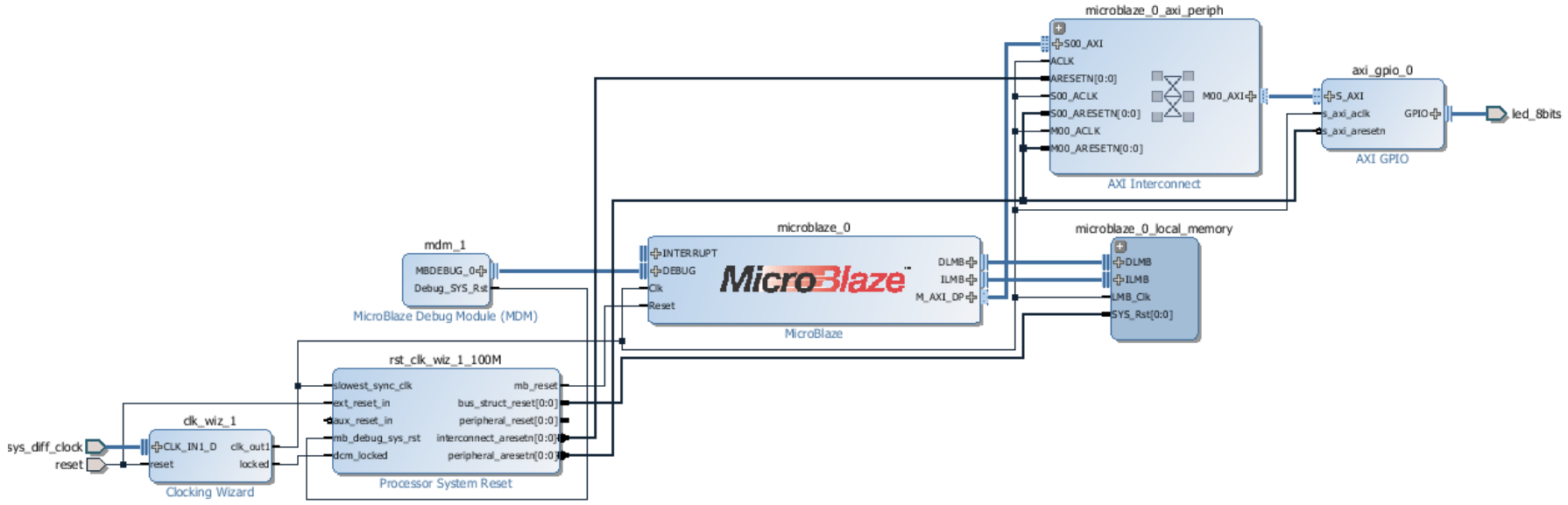
- **Add IP to the design**

Vivado IP Integrator

The **Vivado IP Integrator** enables the development of complex system designs by instantiating and interconnecting IP from the Vivado IP catalog.

The Block Design is a graphical representation of a hardware system, created and managed through the Vivado IP Integrator.

- It enables users to create modular, reusable, and scalable hardware systems efficiently within the Vivado environment.



Hands-on

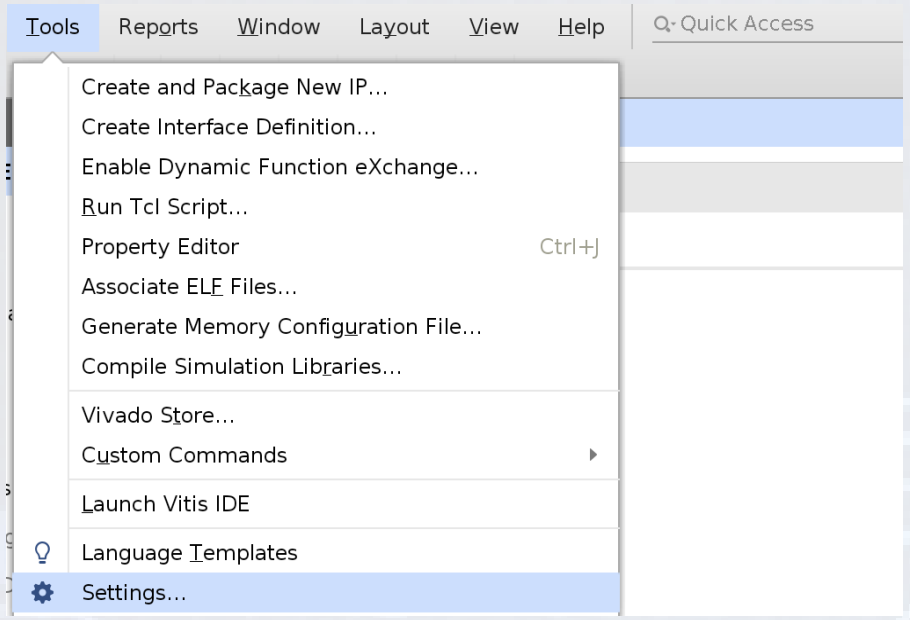
Exercise 13

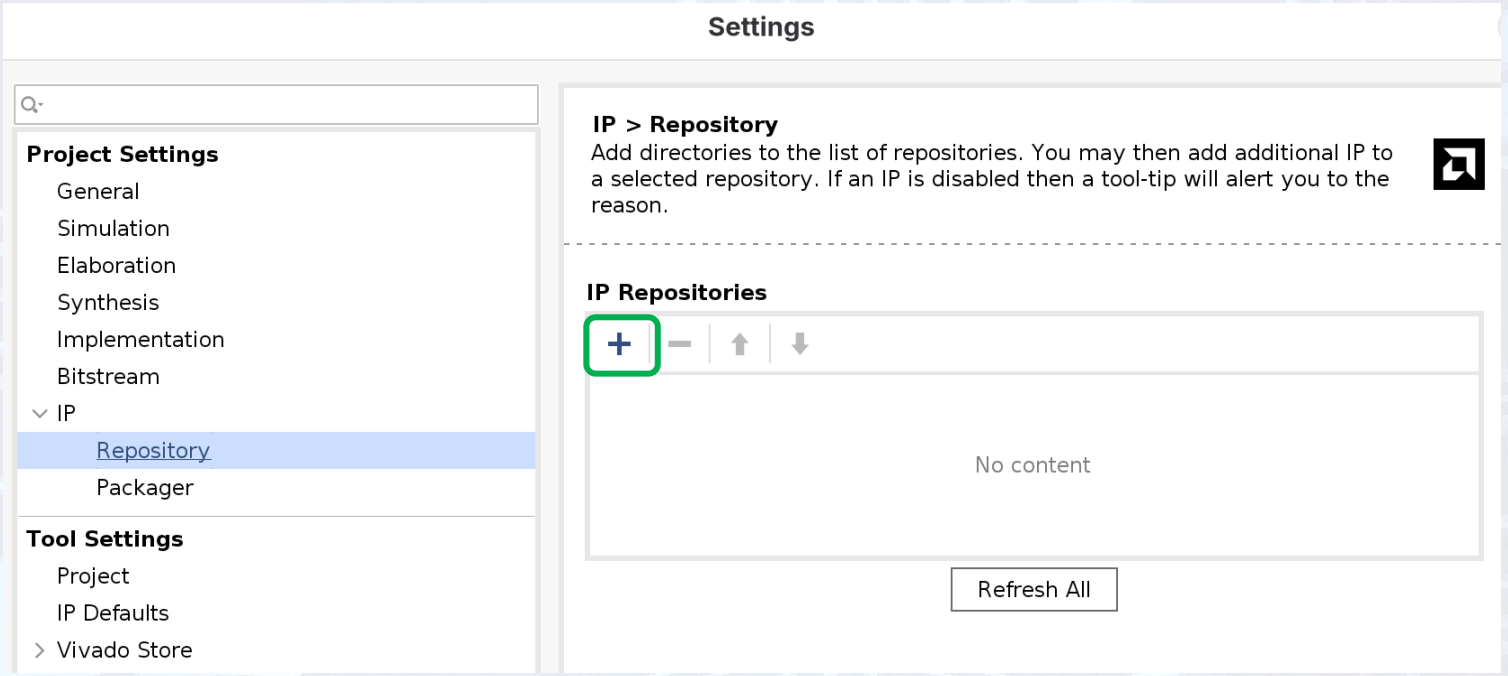
Design Project

- Create a new project for Alveo U55 board
- Create a Block Design and instantiate:
  - FIFO_wrapper
  - ILA (for test_ok signal)
  - VIO (connected to reset signal)
  - Utility_buffer (from differential to single-ended clock)
- Generate BD wrapper
- Create the constraint File modifying the port names as needed to match your design .
- Generate the bitstream

Design project

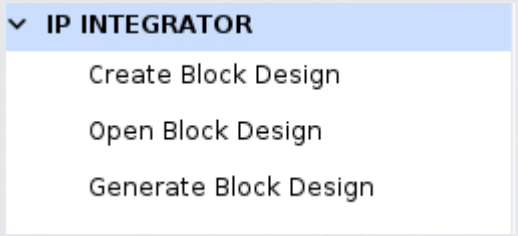
To use the custom IP FIFO_wrapper you need to add a user IP repository.



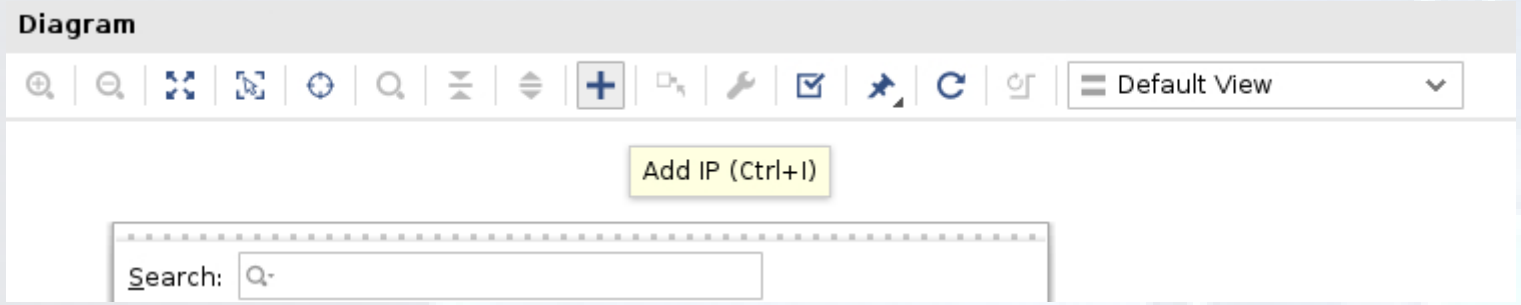


- Specify the location of the IP definition

Design project



- Create a new block design



- Add IP Blocks using the IP Catalog

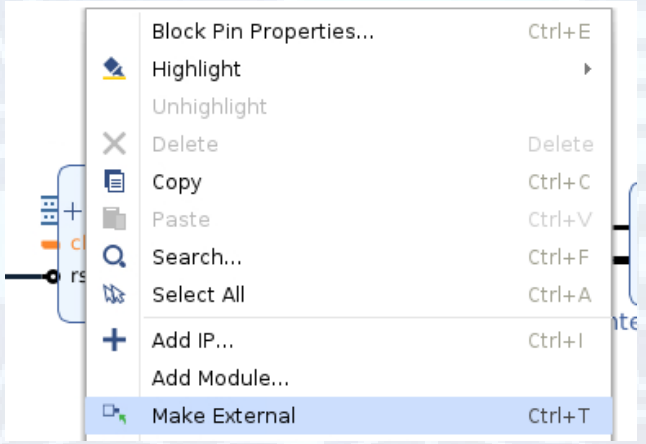
- Configure parameters for each IP block (e.g., data width, clock settings, interface types).

Design project

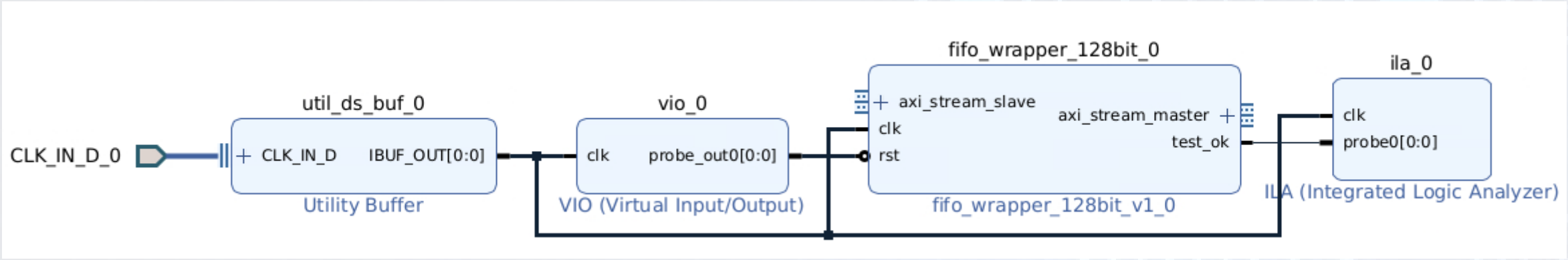
- Connect Blocks manually - click and drag from one port to another - or using "Run Connection Automation" to automatically generate necessary connections

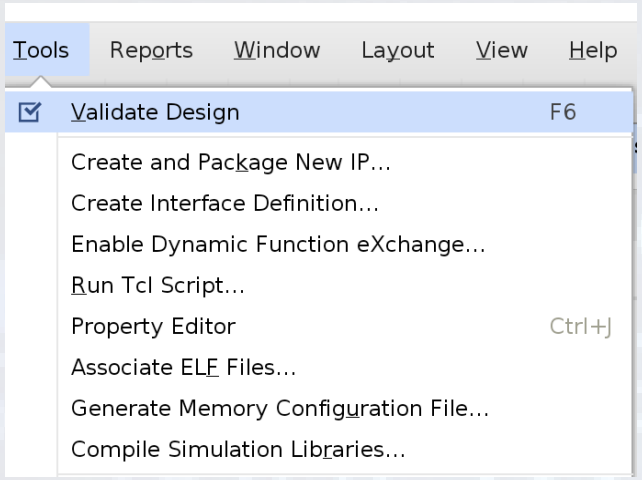
- Expose **VIO clk** as a top-level port - right-click a port and choose “Make External”

- Connect **clk** to fifo_wrapper and to ila

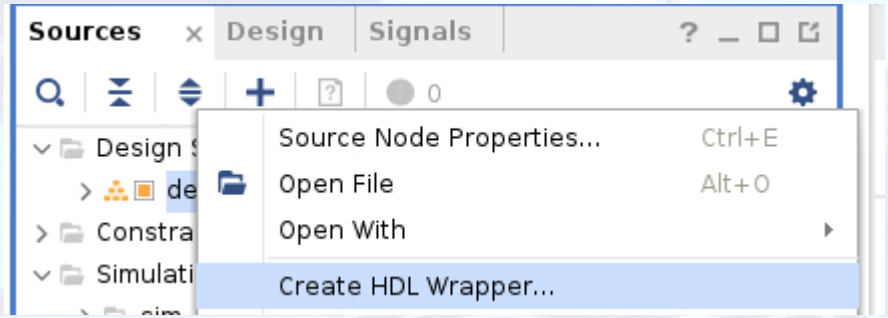


Design project





- Validate the Design to check for errors or missing connections in the block design.



- Create HDL Wrapper

Backup

Vivado Usage Modes

Vivado supports two main usage modes:

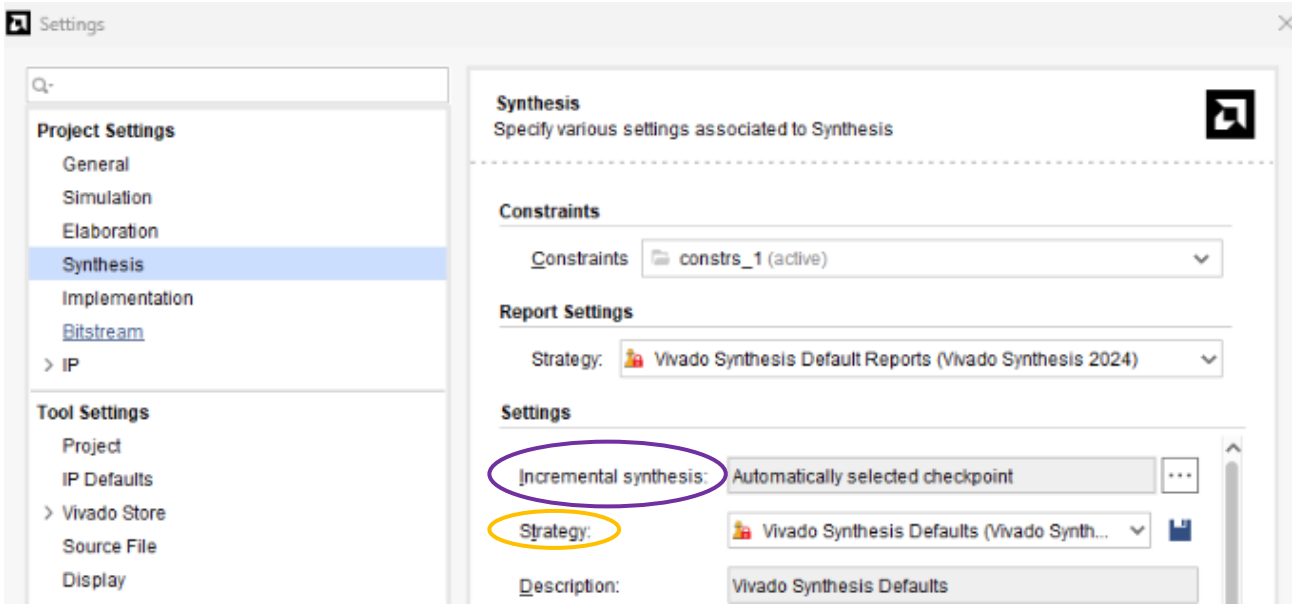
- **Project Mode**
  - Vivado manages source files, constraints, runs, and results inside a project directory.
  - Supports Incremental compilation

- **Non-Project Mode**
  - Designer has full control over the flow, deciding when to run synthesis, implementation, writing bitstream, etc.
  - No project database is created → only the specified files and results are generated.

Flow Element	Project Mode	Non Project Mode
management	automatic	manual
flow navigation	guided	manual
flow customizations	limited (through Tcl commands)	unlimited (through Tcl commands)
reporting	automatic	manual
analysis stages	automatic design checkpoints generation	manual design checkpoints generation

Incremental Synthesis

Vivado Synthesis can be run incrementally : the tool puts incremental synthesis info in the generated **DCP** (Design CkeckPoint) file that can be referenced in later runs.
It detects when the design has changed and only re-runs synthesis on sections of the design that have changed



Incremental Synthesis box to
- use a known checkpoint
- use the last checkpoint created (default)
- disable incremental synthesis

Strategy: describes how aggressive synthesis is with optimizations across partitions.
- **Quick** turns off most optimizations
- **Aggressive** turns on more and repeat synthesis on certain sections
- **Off** tells synthesis not to use the incremental synthesis information in the DCP file.
- **Default**

Incremental Synthesis

When the reference run is performed, the tool partitions out the design as it is performing synthesis. When the incremental run is started, it compares the elaborated design with the reference run and identifies the changed modules.

The information on how much of the design and what parts of the design were re-synthesized can be found in the “Incremental Synthesis Report Summary.”

Advantages:

- Faster for small changes – only modified partitions are re-synthesized.
- Preserves previous optimizations – keeps placement, timing, and routing of unchanged partitions.

Disadvantages:

- Limited benefit for large changes
- Potential QoR degradation preserving unchanged partitions may prevent global optimizations, affecting timing or area efficiency.
- Complex flow management – requires careful partitioning and checkpoint handling to avoid inconsistent results.
- Less effective for small designs – designs with few partitions may synthesize faster with a full synthesis.

Advantages of VHDL

•Enforces stricter rules, in particular strongly typed, less permissive and error-prone
•Initialization of RAM components in the HDL source code is easier (Verilog initial blocks are less convenient)
•Package support
•Custom types
•Enumerated types
•No reg versus wire confusion

VHDL allows buffer port mode when a signal is used both internally, and as an output port when there is only one internal driver. Buffer ports are a potential source of errors during synthesis, and complicate validation of post-synthesis results through simulation.

Advantages of Verilog

•C-like syntax
•More compact code
•Block commenting
•No heavy component instantiation as in VHDL